2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS) | 979-8-3503-8605-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDCS60910.2024.00137

URCD: Unsupervised Root Cause Detection in Microservices Architecture with HGAN.

Harsh Borse, Utkalika Satapathy, Mainack Mondal, and Bivas Mitra
Indian Institute of Technology Kharagpur, India
harshzf2@kgpian.iitkgp.ac.in, utkalika.satapathy01@kgpian.iitkgp.ac.in, mainack@cse.iitkgp.ac.in, bivas@cse.iitkgp.ac.in

***Abstract*—The shift from monolithic services to microservices brings modularity and elasticity, but detecting faults and anomalies is challenging due to diverse data and evolving technology. The heterogeneous nature of this data complicates the analysis of anomaly signatures across various dimensions. Given the continuous evolution of this technology, exhaustively learning from historical data poses difficulties.**

To address these challenges, we present URCD, a solution designed to identify and localize faults or anomalies at the application and service level. Remarkably, URCD achieves this without explicit training on faulty data. Our approach integrates heterogeneous microservice data into a bidirectional weighted graph, leveraging a sophisticated Hyper Graph Attention Network (HGAN) model to process heterogeneous data logs generated by microservices. Our evaluation shows the optimal performance of URCD while detecting root cause of anomalies.

I. Introduction

Microservice architecture has emerged as the predominant trend in the development of cloud-native applications, leading many companies to transition from traditional monolithic architectures to the microservices paradigm. Despite concerted efforts to ensure service quality, microservices systems often demonstrate fragility, rendering failures unavoidable due to their inherent complexity and expansive scale. In the literature, a great number of efforts have been devoted to diagnosing the failure and locating the root cause.

The conventional methods for root cause detection exhibit certain limitations, like: **(i)** Incomprehensive Learning: The exhaustive learning of all potential functional (code-level) and non-functional (performance-related) faults or anomalies within a microservices architecture-based application is deemed impractical. Furthermore, supervised learning methods encounter challenges related to data imbalance, wherein the quality and quantity of historical faulty data are limited. **(ii)** Heterogeneous Data Handling: In distributed microservices applications, faults may manifest across various modalities, including performance monitoring metrics, distributed traces, and system/application logs. Traditional methods struggle to effectively harness heterogeneous data.

For example, in Figure 1 fault A and fault B can be classified as different faults if all the different information from different modalities is considered. Existing frameworks often focus on a subset of these modalities, necessitating separate pipelines for each type of data [1]–[3]. **(iii)** Additionally, representing this heterogeneous modalities effectively is essential to avoid information loss. Taking one modality example, a trace can be breakdown into the pre-processing, waiting time, and post-processing time of the request. Figure 2 represents traces for four different request with their heath state in traditional and our breakdown format, where we can observe that the detailed breakdown of latency can help us locate the faulty part efficiently compared to traditional representation (wait time fault in R4) [4], [5].

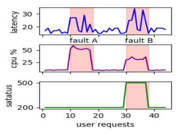


Fig. 1. Modalities behaviour with faults occurrence (red)

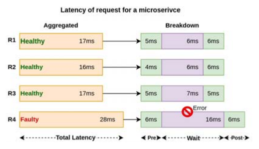


Fig. 2. Latency representation of four request

In our research paper, we present URCD, a novel framework for root cause detection in microservices architectures. URCD leverages heterogeneous data logs without explicit training on anomalous data, utilizing an unsupervised approach to detect both functional and non-functional anomalies. The features in each modality are correlated with each other and different features can be used to estimate a particular feature under specific conditions. The framework combines heterogeneous data sources into unified graph structures for each user request. These graphs are then fed into a combination of complex HGAN and feedforward networks, focusing on each modality's attributes and the relationship between them. By analyzing reconstruction errors during feature estimation with the aid of learned relationships, URCD effectively identifies root causes at the application and service levels. Our evaluation on several major faults and anomalies encountered in microservices architecture demonstrates URCD's effectiveness in root cause detection [4]–[6].

II. Dataset and Problem Definition

A. System Design and Data Collection

Applications: We developed two microservices applications, $library$ and $Ecom$ (Ecommerce), to represent varying levels of complexity. The $library$ app has five microservices, while $Ecom$ has twelve. They offer different levels of interaction between microservices, providing a comprehensive testbed for our research. A detailed Figure 3 presents a flow diagram of these micro-services for $library$ app. A sample

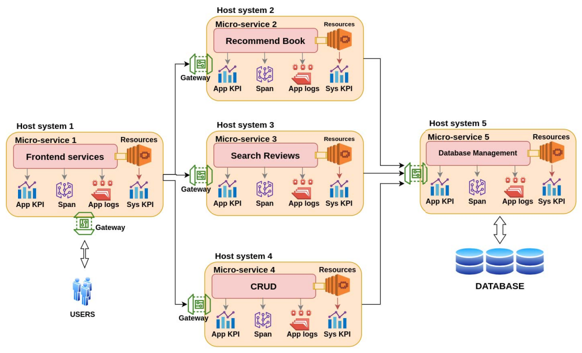
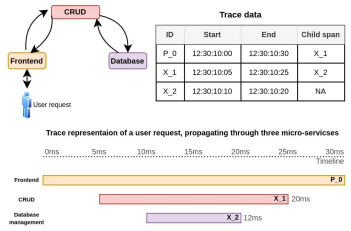


Fig. 3. $library$ system design (detailed)



Trace data

ID	Start	End	Child span
P_0	12:30:10:00	12:30:10:30	X_1
X_1	12:30:10:05	12:30:10:25	X_2
X_2	12:30:10:10	12:30:10:20	NA

Fig. 4. Example of trace and parent-child relation of a span for insert book record operation in $library$ system

user request (r_i) with $library$ app is also represented in Figure 4.

Modalities: Our objective is to gather various types of data/logs with functional and non-functional faults. Specifically, we focus on four main types of data: Traces, performance metrics, application logs and system calls.

(a) Traces: In the context of a microservice system, a trace encapsulates the execution process of a request across different service instances, forming a service invocation chain. Figure 4 illustrates an example of trace for a user request for an insert operation which involves three microservices.

(b) Performance metrics: To track resource utilization, these metrics include performance indicators and resource utilization metrics like $CPU\%, wait\%, RSS, MEM\%, iodelay$ etc. Each microservice and its respective host generate these metrics in a time-series format and store them within the host.

(c) Application Logs: Logs provide detailed information about the application's operations, and user activities in semi-structured text format. We mine categorical and numerical data from the logs. In a distributed system architecture, for a user request r_i each microservice generates a unique log entry (L_v^i $\forall_{v \in V}$, where V= set of microservices) which is stored in the host system log storage.

(d) Systemcalls: We collect all the system calls made by the services for processing a request. We counter the challenge of collecting and ensuring casual ordering of the system call in a distributed system with eBPF based tool. For each request r_i, each microservice generates set of system calls $Q_v = \{q_1, q_2....q_n\}$

Faults: Additionally, we introduce both non-functional and functional anomalies into different microservices. The specifics of these anomalies and the associated information are also described in Table II-A. We run the workload simulating real-world user interaction with each of the anomalies ($F = \{F_0, F_1, F_2, F_3, F_3, F_4, F_5, F_6\}$) injected in different microservices and collect the desired data and logs [7].

Fault	Description	#Request
F0	Normal requests with no faults	66332
F1	microservices increased the compute consumption internally	40100
F2	Zombie process eating unnecessary resources in host system m	56420
F3	Complexity increased in microservice before sending a request	50800
F4	Complexity increased in microservice after receiving a response	51200
F5	Write access to a file is revoked	50100
F6	Database queries issue with relational database	48500

TABLE I
FAULTS AND THEIR DETAILS

B. Problem Definition

Consider a micro-services system consisting of $|V|$ micro-services ($V = \{m_1, m_2, m_3..m_v\}$). A microservice m_x in the system have n features representing its attributes $K = \{k_1, k_2...k_n\}$ from different modalities. A microservice m_x may face anomalies in system attributes $Ka \subset K$. We aim to develop a framework, which can detect root cause microservices $m_x \in V$ having a fault in attributes $Ka \subset K$ and output the resultant tuple (m_x,Ka).

III. Methodology

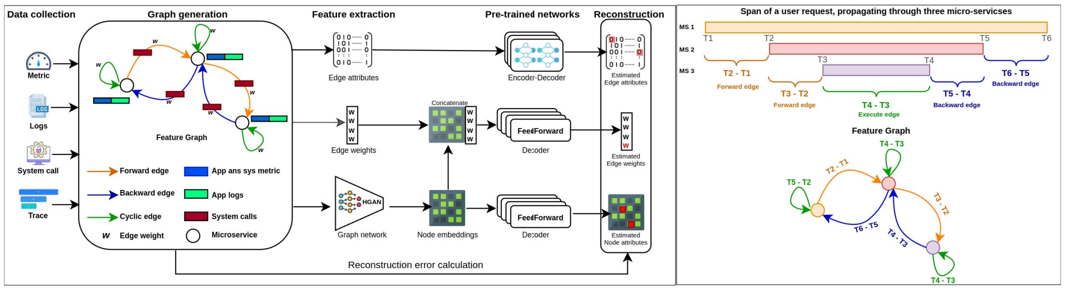
In this section, we detail the development process of URCD, which comprises several key stages. Initially, we address the challenge of managing heterogeneous microservices data by converting it into a unified graph format. Subsequently, our emphasis lies in training a combination of HyperGraph Attention Network and a set of FeedForward networks in an unsupervised setting to effectively learn the intricate dependencies among these components. The feedforward networks are responsible for reconstructing system attributes. Analyzing errors in the reconstructed values offers insight into root causes.

A. *Graph construction*

We integrate diverse data sources from microservices into a unified graph structure for comprehensive analysis. Nodes represent microservices and hosts, edge weights contain latency based properties and edge attributes represent the system calls information, and node features are derived from performance metrics and app logs.

*1) **Node-Edge formation**:* To represent each user request in the form of a graph, each node represents a microservice and a host, edges between them represent the call sequence between the microservices. For this, we use trace data collected for each user request.

Consider a user request r_i (Figure 5(b)), from which we construct the directed graph $g_i = \{V, E\}$, where V represents the micro-services as nodes which were called during the execution of request r_i, and E represents the



(a) Pipeline

(b) Formation of edges and weights from trace

Fig. 5. URCD Pipeline and graph formation from trace.

edge between two micro-services if they had communicated directly with each other for request r_i. $V = \{m_1, m_2, m_3\}$ is the set of nodes in the graph and $E = \{e_1 = (m_1, m_2), e_2 = (m_2, m_3), e_3 = (m_3, m_2), e_4 = (m_2, m_1), e_5 = (m_1, m_1), e_6 = (m_2, m_2), e_7 = (m_3, m_3)\}$ is the set of edges between the nodes. The direction of edges between the nodes is decided by established definitions of edge types in Table II. We represent the weights (latency) between two nodes as an edge weight vector as $Ea = \{e_1 = [|t_2 - t_1|], e_2 = [|t_3 - t_2|], e_3 = [|t_4 - t_3|], e_4 = [|t_6, t_5|], e_5 = [|t_5, t_2|,], e_6 = [|t_4, t_3|], e_7 = [|t_4, t_3|]\}$. At the end of this, we will obtain graph $g_i = \{V, E, Ea\}$ for request r_i.

2) ***Node feature extraction:*** To incorporate the information extracted from the application logs into our analysis, we encode the categorical values with a one-hot encoder and use numerical values as it is. We create a vector representation $L^i_v(v \in V)$ of dimension $1 \times |L|$ for each user request r_i. Next, we extract the microservice and host-level running status. To incorporate this information into the Graph, we consider the performance metrics generated by each micro-service $m_v \in V$ and its corresponding host system $s_v \in S$, represented as a vector of dimension $1 \times |A|$ (A = set of metrics). We also concatenate the log vectors with node features vectors $X^i_v = X^i_v \oplus L^i_v \forall_{v \in V}$.

We collect this concatenated vectors X^i_v from each micro-services where $v \in V$ and stack them to create node attribute matrix X^i, (where $X^i = \cup_{v \in V} X^i_v$) with dimensions $|V| \times |A|$. X^i serves as the node feature in the constructed Feature Graph for request r_i which gives us $g_i = \{V, E, Ea, X\}$.

3) ***Edge-feature extraction:*** To integrate the edge feature, capturing the sequence of events leading to edge formation, we leverage system call logs associated with a specific user request r_i. These logs for r_i, denoted as $Q^* = \cup_{v \in V} Q_v$, where Q_v are divided into two subsets: system calls made prior to sending the request to the subsequent microservice, denoted as Q^f_v, and system calls made subsequent to receiving the response, denoted as Q^b_v. We map these subsets to forward (Q^f_v) and backward (Q^b_v) edges corresponding to a node. We depict system calls using a vector representation. Each column of this vector corresponds to a system call, and the value within each column corresponds to the return value of that system call. Importantly, the order of system calls in the vector mirrors their causal ordering. This gives us $g_i = \{V, E, Ea, X, Q^*\}$

Edge Type	Definition	Example
Forward Edge	For a request r_i, MS_x calls MS_y. It is the time taken by MS_x to process request r_i before calling MS_y.	(T2-T1) (T3-T2)
Backward Edge	For a request r_i, MS_y responds to MS_x. It is the time taken by MS_x to process request r_i after receiving response from calling MS_y.	(T6-T5) (T5-T4)
Wait Edge	For a request r_i, MS_x waiting. It is the waiting time for MS_x for request r_i between calling and receiving response from MS_y.	(T5-T2) (T4-T3)
Execution Edge	For a request r_i, MS_z is final execution. It is the execution time for MS_z for request r_i and no further MS are called	(T4-T3)

TABLE II
EDGE TYPES AND DEFINITIONS (REFERENCE WITH FIG. 5(B)

B. *Node embedding from Hypergraph*

To efficiently learn embeddings on the high-order graph-structured data, we utilise state of the art Hypergraph Convolution network to generate the node embeddings. Where each node embedding represents a unique signature of a microservice. This hyper Graph Convolutional network has the ability to generate node embeddings while aggregating the Egde Features and Egde weights along with Node features for better learning of signatures [8].

For a graph g, Q^* is the Edge features and A is the adjacency matrix derived from E. H is the aggregated Edge features matrix (Step 1: Aggregation of Edge Features). Then H is updated with Ea, which is the edge weight vector (Step 2: Weighted Aggregation), representing the weight associated with each edge. X is the Node feature matrix and Z is the concatenated feature matrix(Step 3: Concatenation of Node Features and Aggregated Edge Features). Z is passed through *HyperGraphConv()* operation to generate the Node embeddings (Step 4: Convolution Operation). The following represents the operations for generating node embedding:

$$H = Q^* \cdot A \qquad \text{(Step 1)}$$
$$H_{ij} = H_{ij} \times Ea_j \qquad \text{(Step 2)}$$
$$Z = [X, H] \qquad \text{(Step 3)}$$
$$X' = \text{HypergraphConv}(Z) \qquad \text{(Step 4)}$$

C. Graph Reconstruction

Here, our aim is to reconstruct the original graph. Hence we break this down into three parallel phases to reconstruct the Node feature, Egde weights and Edge features. For this, we utilise the conventional encoder-decoder technique where we replace the encoder with the Hypergraph convolutional network and decoder with a set of feed-forward neural networks.

Node features: From the node features generated with the *HyperGraphConv()* X', we train the nodes feed-forward networks to reconstruct the node features X. Specifically, we train the FFN_i^{Node} for each node with corresponding node embedding X'_i and reconstruct node feature X_i.

Edge weights: Edge weights are correlated with node features in microservices architecture, hence we utilise the rich node embedding and concatenate them with edge weights, $\mathbb{W}_i = X_i \oplus W_i$. For each edge, we train the feed-forward network $FFN_i^{Edge\_weights}$ with $\mathbb{W}_i$ to reconstruct edge weight W_i.

Edge features: To reconstruct the edge features, we train the edge feed-forward networks to reconstruct the edge features Q^*. Specifically, we train the $FFN_i^{Edge\_feature}$ in a simple encoder-decoder style to reconstruct the original edge feature Q^*.

D. Root cause detection

We utilize the reconstructed graph obtained from the previous step to pinpoint the root cause of the anomaly. The attributes across modalities exhibit correlations, indicating hidden relationships between different attribute sets. Employing feed-forward networks, we learn these relationships, whether causal or otherwise. By analyzing the reconstructed values of each feature, we detect deviations from their original values. Specifically, we gather the reconstructed error for each attribute within the reconstructed graph and identify the top-k values with the highest reconstruction error. If the root cause is among the top-k results returned, we classify it as a true positive case in the root cause detection process.

IV. Evaluation

In this section, we conduct a performance evaluation of URCD.

(i) Table III, shows the accuracy of URCD on functional and non-functional anomalies. Our framework can detect faults manifesting in different modalities, with minimal variations among them (top k, where $k = 3$), due to URCD's capability to harness the diverse heterogeneous data logs simultaneously produced by microservices.

Anomalies	$library$	$Ecom$
F_0	99.8%	99.1%
F_1	99.4%	95.2%
F_2	99.5%	91.8%
F_3	99.1%	90.5%
F_4	99.5%	94.6%
F_5	99.6%	92.4%
F_6	99.3%	95.3%
Avg.	**99.4%**	**94.1%**

TABLE III
ANOMALY-WISE ACCURACY.

Microservices type	**#services**	**Accuracy**
User facing	4	96%
Intermediate	6	95%
Data management	2	91%

TABLE IV
MICROSERVICES VISE PERFORMANCE (ECOM).

(ii) In Table IV, we show the root cause detection accuracy of different types of microservices categories for $library$ app . URCD struggles with data management services as these services are usually called by many services simultaneously hence its health status becomes difficult to generalise.

(iii) We conducted ablation study of performance URCD of with varying modalities and feature. Through Figure 6 we can see that URCD is able to perform even with subset of the modalities, but using only a subset of modalities URCD performance decreases as the faults can be manifested in any modality.

(iv) In Figure 7, we also show the advantage of our unique graph construction method comparing to traditional graph formation method, i.e. latency based, where edges are represented with aggregated latency and causality based, where edges are based on call sequence.

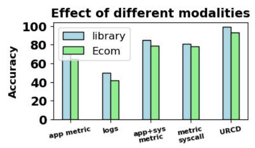


Fig. 6. Modality effect on URCD

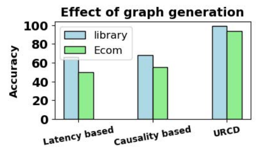


Fig. 7. Graph generation effect

V. Conclusion

In conclusion, URCD proves its efficacy as a robust framework for anomaly detection and root cause analysis within microservices systems. Through the integration of diverse data sources, including traces, performance metrics, application logs, and system calls, URCD offers a comprehensive insight into system behavior and facilitates the identification of potential anomalies without explicitly being trained with anomalous data. Its adeptness in addressing both non-functional and functional anomalies across various heterogeneous modalities renders URCD suitable for a broad spectrum of applications.

References

[1] O. Kalinagac, W. Soussi, and G. Gür, "Graph based liability analysis for the microservice architecture," in *2022 18th International Conference on Network and Service Management (CNSM)*, 2022, pp. 364–366.

[2] Y. Chen, N. Chen, W. Xu, L. Lian, and H. Tu, "Mfrl-ca: Microservice fault root cause location based on correlation analysis," in *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, 2021, pp. 90–101.

[3] M. Ma, W. Lin, D. Pan, and P. Wang, "Servicerank: Root cause identification of anomaly in large-scale microservice architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp. 3087–3100, 2022.

[4] H. X. Nguyen, S. Zhu, and M. Liu, "A survey on graph neural networks for microservice-based cloud applications," *Sensors*, vol. 22, no. 23, 2022.

[5] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," *ACM Comput. Surv.*, feb 2022.

[6] J. Soldani and D. Andrew, "The pains and gains of microservices: A systematic grey literature review," *Journal of Systems and Software*, 2018.

[7] X. Zhou, X. Peng, and Xie, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, 2021.

[8] S. Bai, F. Zhang, and P. H. S. Torr, "Hypergraph convolution and hypergraph attention," 2020.