© 2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Root Cause Analysis for Middleware Issues by Kubernetes Resource Events

Tomoyuki Koyama [*†], Takayuki Kushida [*], Soichiro Ikuno [*]
[*]Graduate School of Bionics, Computer and Media Sciences,
Tokyo University of Technology, Tokyo, Japan
[†]Mercari, Inc., Tokyo, Japan
d212400159@edu.teu.ac.jp, kushida@acm.org, ikuno@stf.teu.ac.jp

***Abstract*—A system failure caused by middleware requires system administrator to investigate multiple telemetry data such as logs, traces, and metrics across the application and middleware. Conventional methods focus on application telemetry data while these methods lack analysis of middleware's telemetry data. System administrator needs to investigate multiple telemetry data to find the root cause of the system failure. This paper proposes a method of root cause analysis for middleware issues by resource events, dependency graphs, resource definitions, and traces. The proposed method constructs dependency graphs by identifying Kubernetes resources from traces and analyzing Kubernetes resource configurations. The method identifies root cause candidates by scoring increases in Kubernetes resource events. Evaluation experiments measured Hits@k and MRR of the proposed method and baseline methods. Evaluation results show that MRR of the proposed method is approximately 0.28 greater than the baseline method on average and Hits@5 of the proposed method is approximately 0.88 greater than the baseline method on average.**

***Index Terms*—Root Cause Analysis, Distributed Tracing**

I. Introduction

Cloud and Distributed Systems Laboratory (CDSL) operates the web service called Doktor[1]. Doktor is a microservice-based application deployed on a Kubernetes cluster. Rook Ceph[2] is used for persistent storage on the cluster. Rook Ceph failure occurred in Doktor on June 21, 2025. Fig. 1 shows the overview of the failure. The author microservice (author MS) runs on a Kubernetes node. The author MS consists of an application container (App Pod), a MongoDB container (Mongo Pod), and a persistent volume (PV) for the Mongo Pod. Rook Ceph has an OSD Pod for the PV. The OSD Pod was in CrashLoopBackOff state when a Kubernetes node hung and was restarted. The system administrator received alerts about the failure and investigated metrics, logs, and traces to find the root causes. The cause of the failure was an error in the OSD Pod of Rook Ceph. Listing 1 shows the result of kubectl command. Line 1 represents kubectl get command to get the state of the OSD Pod. OSD (Object Storage Daemon) is a process for running a storage server. Line 2 represents OSD Pod "rook-ceph-osd-3-6c5c8bf688-n4d5z" that was in CrashLoopBackOff state. As a result, the Mongo Pod failed to mount the persistent volume (PV), causing an error in the author MS.

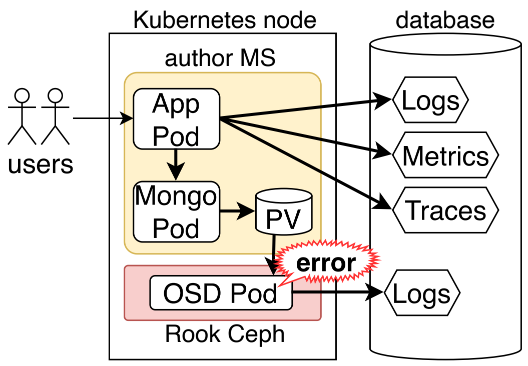

Fig. 1: Overview of the failure in Doktor

Listing 1: The result of kubectl command
```
1 $ kubectl get pod -A -o wide | grep osd
2 rook-ceph rook-ceph-osd-3-6c5c8bf688-
  n4d5z 0/2 Init:CrashLoopBackOff ...
```

Listing 2: Logs of the OSD Pod related to the error
```
1 $ kubectl logs rook-ceph-osd-3-6c5c8bf688
  -n4d5z -n rook-ceph -c expand-bluefs
2 2025-06-21T09:40:34.447+0000 7fdddf249980
   -1 rocksdb: Corruption: SST file is
  ahead of WALs in CF default
3 2025-06-21T09:40:34.447+0000 7fdddf249980
   -1 bluestore(/var/lib/ceph/osd/ceph-3)
  _open_db erroring opening db:
```

Listing 2 shows the OSD Pod logs related to the error. Line 1 shows the kubectl logs command. Line 2 indicates that the SST (Sorted String Table) file is ahead of WALs in RocksDB and an inconsistency occurred. RocksDB (rocksdb) is a database used for storing metadata in BlueStore. Line 3 indicates that an error occurred on opening RocksDB in BlueStore (bluestore). BlueStore is a storage backend component of Rook Ceph.

The series of failure investigation processes used a combination of telemetry (traces, logs, and metrics). System administrator required different methods to check each type

[1]https://doktor.tak-cslab.org
[2]https://rook.io/

of telemetry. For example, system administrator executed "kubectl describe" commands or "kubectl get" commands to check the status of resources in the Kubernetes cluster. System administrator issued search queries from web UI to log servers to search for application logs. Manual operations for log, metrics, and traces required system administrator to have experience of system diagnostics and was time-consuming and error-prone [1]–[3]. Investigation results for 22 system failures show that system administrator spent more than one hour to investigate logs or traces in most incident handling [4].

Middleware becomes involved in failures in addition to applications as system failures become more complex [5], [6]. Investigation results at Microsoft show that middleware and infrastructure-related failures account for 26.3% of all failures [7]. Failures involving both applications and middleware increase the difficulty of root cause analysis for system administrator as system failures become more complex.

Issue

System administrator manually investigates middleware logs and traces to identify the cause of errors in failures caused by Rook Ceph errors. Distributed systems represented by microservice architecture contain more components than monolithic systems. The number of investigation targets increases as the number of components increases. Manual investigation requires professional knowledge, cross-team collaboration and operational experience [8]. System administrator lacking sufficient experience for root cause analysis is required to engage in trial-and-error processes. This requires extended investigation time. The time required for root cause analysis is included in MTTR (Mean Time to Repair). This necessitates rapid cause investigation. There are root cause analysis methods with collecting and analyzing application telemetry [9]. These methods by multiple data sources such as application latency and resource dependencies require further investigation. System administrator lacking sufficient experience for root cause analysis requires manual investigation of multiple telemetry data sources on system failure.

II. Related Study

Root cause analysis is a time-consuming task for system administrator, so that research on automation is conducted. Root cause analysis methods using single data sources and multiple data sources have been proposed. Metrics-based methods were proposed [3], [9]. Trace-based methods were proposed [10], [11]. Log-based methods were proposed [12], [13]. These single data source-based methods failed to identify problems when the data source lacked failure symptoms, resulting in insufficient accuracy of failure cause identification.

Multi modal RCA methods using multiple data sources were proposed [1], [14]–[16]. These methods used multiple data source telemetry for root cause analysis. DeepTraLog used deep learning to detect anomalies using multiple data source telemetry [14]. The method used logs and traces to collect application dependencies and build a dependency graph. This study focused on anomaly detection rather than root cause analysis. Therefore, this method addresses only anomaly detection and lacks capabilities for root cause analysis caused by middleware. Eadro used supervised machine learning for root cause analysis using multiple data source telemetry [15]. It showed higher accuracy than unsupervised machine learning. This method required labeling training data, which was a burdensome task for system administrator. The method remains to be improved without labeling training data. Nezha integrated traces and logs to enable detailed tracking of a series of processing processes on a request unit [1]. This method lacks a mechanism to align the logs of middleware and the traces of applications. Therefore, this method addresses application-level issues and lacks capabilities for root cause analysis caused by middleware. A study conducted root cause analysis by deriving simple event causal graphs from past failures and calculating weights [16]. The study used past failures in datasets. Past failures had undergone recurrence prevention measures and had low probability of recurrence, making applicable cases limited.

III. Proposed Method

This paper proposes EventRCA that is a method of root cause analysis for middleware issues by resource events, dependency graphs, and traces. EventRCA creates dependency graphs of Kubernetes resources from traces corresponding to these resources. It finds resource events for each resource based on the dependency graphs. EventRCA obtains resource events for each resource and determines root cause candidates by scoring increases in Kubernetes resource events. The main challenge is determination of root cause candidates with scoring for middleware issues. The key contribution is to identify root cause candidates for middleware issues without manually investigating multiple telemetry data sources.

Fig. 2 shows the overview of the proposed method. Microservices represent a set of applications running on a Kubernetes cluster. Traces are collected from microservices to track a series of request processing across microservices. Microservices are deployed on the Kubernetes cluster as containers. Each resource running in Kubernetes cluster is defined by resource definition in YAML format. Kubernetes generates events for each resource when resources are created, updated, or deleted. EventRCA is the proposed method which collects

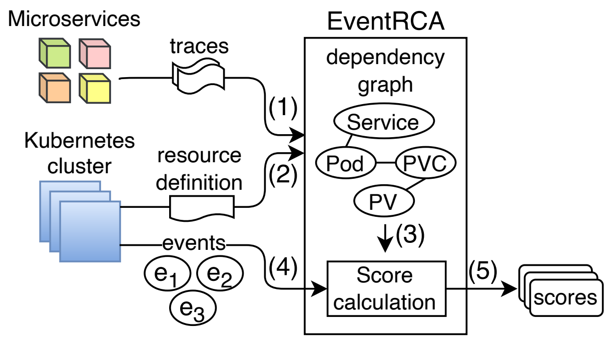

Fig. 2: Overview of the proposed method

traces, resource definitions, and events as input. It creates a dependency graph of Kubernetes resources from traces and resource definitions and finds resource events for each resource based on the dependency graph. It determines root cause candidates with scores by resource event count changes. Score calculation of EventRCA takes dependency graph and resource events as input. It calculates scores for end to end routes of dependency graph and determines root cause candidates with scores. When a system failure occurs, the following process is executed.

(1) EventRCA collects traces from Microservices.
(2) EventRCA collects resource definitions corresponding to traces from the Kubernetes cluster.
(3) EventRCA constructs a dependency graph from the traces and resource definitions.
(4) Score calculation of EventRCA collects events from the Kubernetes cluster.
(5) Score calculation of EventRCA identifies root cause candidates by scoring increases in Kubernetes resource events and outputs scores.

The proposed method uses static analysis approach with resource definition to create the dependency graph. Static analysis approach is a conventional approach. This approach allows for a light footprint and has low resource consumption as compared with dynamic analysis approach. The dependency graph is created by running resource definition through Kubernetes API. Resources in Kubernetes have dependencies to other resources. Fig. 3 shows dependency graph creation process. The trace represents a trace. The span represents a span that is a part of the trace. Kubernetes resource names are obtained from each span using trace metadata. For example, Pod2 is obtained from span2. The metadata is collected on trace creation in the application. The resource definition corresponding to the resource name is obtained from API server of Kubernetes cluster. Kubernetes cluster represents a group of Kubernetes nodes. The resource definition contains dependency parameters which represent relationships to other resources. For example, the resource definition of Pod2 includes “author: app”. The resources that include the dependency parameters in the resource definition are collected. For example, a resource definition of “kind: Service” includes “app: author”, the resource has a dependency to the resource “kind: Pod”. These processes are applied to all traces.

Listing 3 shows an example of dependency list. The from field represents source resource and the to field represents destination resource. For example, Pod resource named “front-app-deploy-6b4dbf8d84-rbwvx” in front namespace has a dependency to ConfigMap resource named “istio-ca-root-cert” in front namespace.

Listing 4 shows an example of Kubernetes events that are collected from a Kubernetes cluster. Events have Type, Reason, Age, From, and Message attributes. Type field includes the state of the event. Reason field includes the type of event. Age field includes the timestamp when the event occurred. From field represents the resource that caused the event.

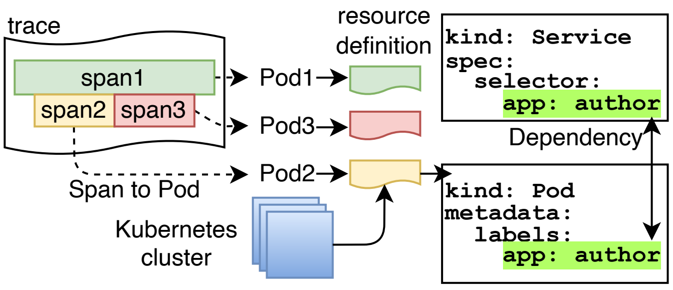

Fig. 3: Dependency graph creation process

Listing 3: An example of dependency list

```
{
"from":"front/Pod/front-app-deploy-6
b4dbf8d84-rbwvx",
"to":"front/ConfigMap/istio-ca-root-cert"
}
```

Listing 4: An example of Kubernetes events

```
Type Reason Age From Message
---- ------ ---- ---- -------
Normal Scheduled 69s default-scheduler
Successfully assigned default/ubuntu to
clematis-worker2
```

Message includes the details of the event. For example, the event in the line 3 represents that the Pod (default/ubuntu) was scheduled 69 seconds ago to the Kubernetes node clematis-worker2.

EventRCA outputs failure cause candidates as a list. The output includes a score representing the probability and a pair of target resources. The score is a decimal number between 0 and 1. The output is sorted by score in ascending order. For example, if the score is 0.9 and the target resource is Pod A, the probability that Pod A is the cause of the error is 90%.

Algorithm 1 shows the score calculation process. The algorithm takes a dependency list as input and outputs scores for each end-to-end path. The dependency list contains pairs of source and destination resources. Line 1 calls GET_PATHS which obtains all end-to-end paths from the dependency list. An end-to-end path is a sequence of resources from the source resource to the destination resource. For example, the end-to-end path of the dependency list in Listing 3 is “(front/Pod/front-app-deploy-6b4dbf8d84-rbwvx, front/ConfigMap/istio-ca-root-cert)”. Line 2 initializes the variable of *counts* as an empty list. It represents the number of events for each end-to-end path. Line 3 to 9 calculate the number of events for each end-to-end path. Line 4 initializes the variable of *count* as 0. It represents the number of events for the current end-to-end path. Lines 5 to 7 count the number of events for each resource in the current end-to-end path. Line 6 calls GET_EVENTS which obtains event counts for the resource using Kubernetes API and updates the variable of *count*. Line 8 combines the variable of *counts* with a pair of the end-to-end path and the number of events for the current end-to-end path. For example, if *counts* is

Algorithm 1 Score calculation process

```
Input: $dependencyList$: list of (src, dest) pairs
Output: $result$: list of (path, score) pairs
 1: $paths \leftarrow$ GET_PATHS($dependencyList$)
 2: $counts \leftarrow []$
 3: for each $path \in paths$ do
 4:     $count \leftarrow 0$
 5:     for each $resource \in path$ do
 6:         $count \leftarrow count +$ GET_EVENTS($resource$)
 7:     end for
 8:     $counts \leftarrow counts \cup (path, count)$
 9: end for
10: $total \leftarrow$ sum of all $count$ in $counts$
11: $result \leftarrow []$
12: for each $(path, count) \in counts$ do
13:     $score \leftarrow count/total$
14:     $result \leftarrow result \cup$ (LAST($path$), $score$)
15: end for
16: return $result$ sorted by $score$ in descending order
```

empty and the end-to-end path is "(front/Pod/front-app-deploy-6b4dbf8d84-rbwvx, front/ConfigMap/istio-ca-root-cert)" and the number of events is 10, the variable of *counts* is updated to "((front/Pod/front-app-deploy-6b4dbf8d84-rbwvx, front/ConfigMap/istio-ca-root-cert), 10)". Line 10 calculates the total number of events for all end-to-end paths. Line 11 initializes the variable of *result* as an empty list. It represents the list of pairs of the end-to-end path and the score. Lines 12 to 15 calculate the score for each path and add the pair of the path and the score to the variable of *result*. Line 13 divides the number of events for the current path by the total number of events for all end-to-end paths and updates the variable of *score*. Line 14 adds the pair of the last resource in the path and the score to the variable of *result*. Line 16 returns the variable of *result* sorted by score in descending order.

Use case scenario

Fig. 4 represents a situation where system administrator investigates the causes of system failures using distributed tracing. The distributed tracing Web UI is a user interface for distributed tracing. System administrator accesses individual trace screens in the Web UI to find the root causes of system failures when system failures occur. The Web UI displays a series of request processing arranged in timestamp order. One trace consists of multiple spans, and system administrator finds spans containing errors using the Web UI. When errors propagate between spans, system administrator searches for the span where the error first occurred. Individual trace screens enable identification of spans and microservices causing errors. However, when error causes lie in middleware, cause investigation cannot be performed using only the distributed tracing Web UI. A score list screen is added to display the proposed method in the distributed tracing Web UI. The score list screen displays scores representing cause probabilities, resources assumed to be causes, and events related to resources. Higher scores are

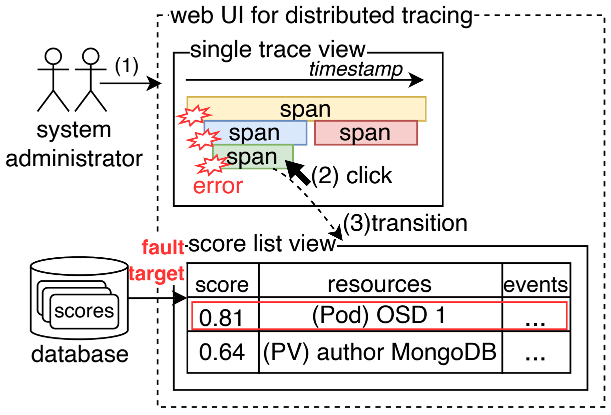

Fig. 4: Use case scenario

displayed in upper rows. Scores displayed on the score list screen are stored in a database. Scores stored in the database are output from EventRCA.

IV. EVALUATION

Evaluation metrics and Evaluation method

Execution time is measured to evaluate the time required to identify root cause candidates. It represents the duration from the start to the end of program execution. Mean Reciprocal Rank (MRR) is measured to evaluate the output of root cause analysis by the proposed method. Top-k accuracy does not represent the order of root cause analysis results. MRR represents the order of root cause analysis results. The value is used as an evaluation KPI in RCA studies [17]–[19]. Top-k Hits (Hits@k) is measured to evaluate a list of root cause candidates. Hits@k represents the number of correct root cause candidates in the top-k results. The value of k is set to $\{1, 2, 3, 4, 5\}$. Survey results from 386 engineers showed that they wanted top-5 results as root cause candidates [20].

Fig. 5 shows the evaluation method. Failure scenario represents a scenario of system failure. Doktor is deployed on the Kubernetes cluster. Traces are stored in SigNoz. EventRCA collects events from Kubernetes cluster and traces from SigNoz. Evaluator is a software that calculates evaluation metrics (MRR and Hits@k). It compares the fault targets and the score list. The fault targets contain the correct root cause candidates. The score list contains pairs of scores and resource names as root cause candidates.

Failure scenarios

Scenario 1 involves OSD Pod errors in Rook Ceph. Rook Ceph operates in a Kubernetes cluster. One node in the Kubernetes cluster becomes unresponsive and requires a restart. The Pod fails to terminate properly during the restart. Data corruption results from this failure. This scenario is reproduced by performing a restart of one node in the Kubernetes cluster.

Scenario 2 actually occurred in Doktor's development environment on November 21, 2025. One of the worker nodes in the Kubernetes cluster experienced high memory usage and OOM kill. The OOM kill caused MinIO object storage in

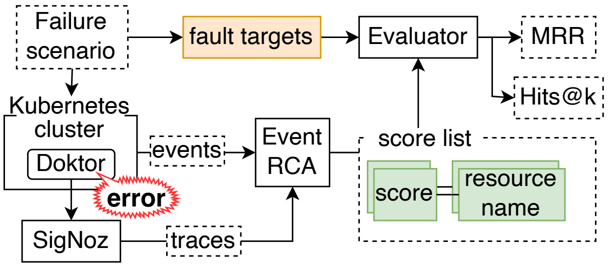

Fig. 5: Evaluation method

paper microservice to be unavailable. MinIO Pod was restarted on the other node when the OOM kill occurred. The restarted MinIO Pod failed to read and write objects because multiple volume attachments occurred. As a result, paper download requests to the paper microservice failed.

Experimental environment and Software implementation

The experiment uses Doktor as a web application. Doktor consists of multiple microservices (stats, front, paper, fulltext, author, thumbnail). These microservices operate as containers. OpenTelemetry SDK is installed in the application containers of each microservice, and logs, traces, and metrics are collected. Application containers are deployed on a Kubernetes cluster. The Kubernetes cluster consists of 1 master node and 3 worker nodes. All nodes constituting Kubernetes cluster have the same hardware configuration (vCPU: 8 cores, Memory: 8GB, Disk: 40GB). K3s v1.31.5 is used as the Kubernetes engine. SigNoz is used for monitoring[3]. SigNoz is a monitoring system capable of storing and retrieving telemetry. Telemetry generated by each microservice of Doktor is collected by OpenTelemetry and transferred and stored to SigNoz.

EventRCA consists of four functions. The function of *get_trace* loads traces from SigNoz and saves them into traces.json. The function of *get_graph* loads resource definition from Kubernetes API server and traces from traces.json. It creates a dependency graph of Kubernetes resources and saves it into dependencies.json. The function of *get_event* gets events from a Kubernetes API server and creates a list of resource events and saves it into events.json. The function of *get_score* loads dependency graph from dependencies.json and resource events from events.json. It calculates scores for end to end routes of dependency graph and determines root cause candidates with scores. Listing 5 shows an example of score list. The score list is a part of the output of EventRCA. The score list contains pairs of scores and resource names as root cause candidates. The resource names are formatted as “namespace/resource type/resource name”.

Listing 5: An example of score list

```
0.0495 stats/ConfigMap/istio-ca-root-cert
0.0495 /StorageProvisioner/rook-ceph.rbd.
csi.ceph.com
```

[3]https://signoz.io/

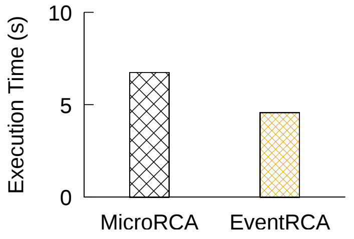

Fig. 6: Baseline comparison by execution time

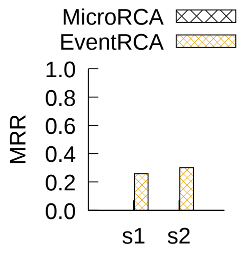

Fig. 7: Baseline comparison by MRR

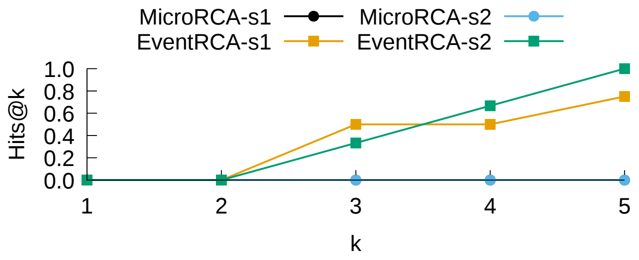

Fig. 8: Baseline comparison by Hits@k

Line 1 represents that the score of “stats/ConfigMap/istio-ca-root-cert” is 0.0495. Line 2 represents that the score of “StorageProvisioner/rook-ceph.rbd.csi.ceph.com” is 0.0495.

Experimental results

Fig. 6 shows baseline comparison by execution time. X-axis represents method. MicroRCA is a baseline method and it is a conventional RCA method based on metrics [9]. EventRCA is the proposed method. Y-axis represents execution time in seconds. Each bar represents the average execution time of 10 executions. The execution time of EventRCA was 4.56 seconds while that of MicroRCA was 6.74 seconds. The average execution time of EventRCA is approximately 2.18 seconds shorter than that of MicroRCA.

Fig. 7 shows baseline comparison by MRR. X-axis represents experimental scenarios. Y-axis represents MRR. Legend represents methods. MicroRCA is a baseline method [9]. EventRCA is the proposed method. MRR of MicroRCA was 0.00 and MRR of EventRCA was 0.26 for Scenario 1 (s1). MRR of EventRCA is 0.26 greater than that of MicroRCA in s1. MRR of MicroRCA was 0.00 and that of EventRCA was 0.30 for Scenario 2 (s2). MRR of EventRCA is 0.30 greater than that of MicroRCA in s2. The results suggest that EventRCA achieves higher accuracy than MicroRCA in Rook Ceph failure scenarios. The average MRR of EventRCA is approximately $(0.30+0.26)/2 = 0.28$ greater than the baseline method.

Fig. 8 shows baseline comparison by Hits@k. X-axis represents k which is the number of top results. Y-axis represents Hits@k. Legend represents pairs of methods and scenarios. For example, MicroRCA-s1 represents MicroRCA in Scenario 1. MicroRCA is a baseline method [9]. EventRCA is the proposed method. The result shows that the top-5 candidates of EventRCA include the correct root cause candidates in Scenario 1 and Scenario 2. MicroRCA fails to identify the correct root cause within the top-5 candidates in Scenario 1 and Scenario 2. The average Hits@5 of EventRCA is approximately $(1.00+0.75)/2 \approx 0.88$ greater than the baseline method.

V. Discussion

The proposed method uses Kubernetes resource events for root cause analysis. These events contain resource lifecycle information such as creation, update, and deletion, representing the state of resources at specific points in time without their internal states. Log messages provide detailed internal state information, enabling code-level root cause investigation while applications generate large volumes of logs (e.g., tens of millions per second in Tencent [21]), requiring production environments to apply log sampling [22]. Standard sampling may discard logs necessary for investigation. Violation likelihood sampling addresses this by assigning sampling probabilities based on the likelihood of violations [23].

VI. Conclusion

This paper proposed EventRCA which is a method of root cause analysis for middleware issues by resource events, dependency graphs, and traces. EventRCA created dependency graphs of Kubernetes resources and found resource events for each resource based on the dependency graphs. Microservices were deployed on the Kubernetes cluster for evaluation experiments. The experimental results show that the average MRR of the proposed method is approximately 0.28 greater than the baseline method. The average Hits@5 of the proposed method is approximately 0.88 greater than the baseline method.

Acknowledgement

This work was supported by JSPS KAKENHI Grant Number JP23K11073 and JP23K11087. Generative AI (Claude Opus 4.5) was used for all sections of proofreading.

References

[1] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 553–565.

[2] Á. Brandón, M. Solé, A. Huélamo, D. Solans, M. S. Pérez, and V. Muntés-Mulero, "Graph-based root cause analysis for service-oriented and microservice architectures," *Journal of Systems and Software*, vol. 159, p. 110432, 2020.

[3] L. Wu, J. Tordsson, J. Bogatinovski, E. Elmroth, and O. Kao, "Microdiag: Fine-grained performance diagnosis for microservice systems," in *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*. IEEE, 2021, pp. 31–36.

[4] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

[5] N. Zhao, J. Chen, Z. Yu, H. Wang, J. Li, B. Qiu, H. Xu, W. Zhang, K. Sui, and D. Pei, "Identifying bad software changes via multimodal anomaly detection for online service systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 527–539.

[6] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 683–694.

[7] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, "How to fight production incidents? an empirical study on a large-scale cloud service," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 126–141.

[8] L. Li, X. Zhang, X. Zhao, H. Zhang, Y. Kang, P. Zhao, B. Qiao, S. He, P. Lee, J. Sun *et al.*, "Fighting the fog of war: Automated incident detection for cloud systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 131–146.

[9] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Microrca: Root cause localization of performance issues in microservices," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2020.

[10] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *Proceedings of the Web Conference 2021*, 2021, pp. 3087–3098.

[11] Z. Li, J. Chen, R. Jiao, N. Zhao, Z. Wang, S. Zhang, Y. Wu, L. Jiang, L. Yan, Z. Wang *et al.*, "Practical root cause localization for microservice systems via trace analysis," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 2021, pp. 1–10.

[12] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 60–70.

[13] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 102–111.

[14] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang, "Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 623–634.

[15] 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)*. IEEE, 2023, pp. 1750–1762.

[16] Z. Yao, C. Pei, W. Chen, H. Wang, L. Su, H. Jiang, Z. Xie, X. Nie, and D. Pei, "Chain-of-event: Interpretable root cause analysis for microservices through automatically learning weighted event causal graph," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 50–61.

[17] Z. Xie, S. Zhang, Y. Geng, Y. Zhang, M. Ma, X. Nie, Z. Yao, L. Xu, Y. Sun, W. Li *et al.*, "Microservice root cause analysis with limited observability through intervention recognition in the latent space," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 6049–6060.

[18] L. Zheng, Z. Chen, J. He, and H. Chen, "Mulan: multi-modal causal structure learning and root cause analysis for microservice systems," in *Proceedings of the ACM Web Conference 2024*, 2024, pp. 4107–4116.

[19] D. Wang, Z. Chen, J. Ni, L. Tong, Z. Wang, Y. Fu, and H. Chen, "Interdependent causal networks for root cause localization," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5051–5060.

[20] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 165–176.

[21] M. Yu, Z. Lin, J. Sun, R. Zhou, G. Jiang, H. Huang, and S. Zhang, "Tencentcls: the cloud log service with high query performances," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3472–3482, 2022.

[22] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: an industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, no. 1, p. 25, 2022.

[23] S. Meng and L. Liu, "Enhanced monitoring-as-a-service for effective cloud management," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1705–1720, 2012.