Kubernetesのリソースイベントと依存グラフとトレースによる障害の原因調査にかかる時間の短縮

小山 智之[1,a)]　串田 高幸[1,b)]　生野 壮一郎[1,c)]

概要：Web アプリケーションの障害の原因調査のプロセスでは，システム管理者はデータソース (トレース，ログ，メトリクス) を解析する．ある障害においてアプリケーションでのエラーの原因がミドルウェアにある場合，その原因の調査には複数のデータソースを解析する必要がある．こうした作業はシステム管理者が手動で行うため，原因調査に時間がかかっている．システム管理者は障害の原因特定に時間がかかっており，障害の原因調査の効率化が必要である．本稿ではミドルウェアから出力されるイベントと依存グラフ，トレースを使用した障害の原因調査の手法を提案する．提案手法ではアプリケーションから出力されたトレースに関連する依存グラフを作成する．その後，依存グラフをもとにリソースごとのイベントの時系列の順序を求め，障害原因の候補をリストアップする．インターネット上に公開したマイクロサービスアーキテクチャで設計された Web アプリケーションを対象に障害を再現し，提案手法の障害原因の候補に実際の原因が含まれるかを評価する．

1. はじめに

背景

クラウド・分散システム研究室（Cloud and Distributed Systems Laboratory, CDSL）では Web サービスの Doktor[*1]を運用している．Doktor では CDSL で作成された PDF 形式のテクニカルレポートの検索やダウンロードができる．Doktor はマイクロサービスアーキテクチャに基づいて構築されている．マイクロサービスアーキテクチャは Google や Amazon で使用されている分散型のアーキテクチャスタイルの 1 つである [1]．マイクロサービスアーキテクチャは，システム全体を複数の独立したサブシステムに分割することで，ソフトウェアの変更しやすさを向上している [2]．Doktor は Kubernetes クラスタ上に構築されている．永続ストレージには分散ストレージの Rook Ceph[*2]が使用されている [3]．

Doktor で 2025 年 6 月 21 日に Rook Ceph の障害が発生した．図 1 に Doktor で発生した障害の概要を示す．Kubernetes ノードには author マイクロサービス (author MS) が動作している．author MS は Doktor を構成するマイクロサービスの 1 つである．author MS にはアプリケーションのコンテナ (App Pod)，Mongo DB のコンテナ (Mongo Pod)，Mongo Pod 用の永続ディスク (PV) が存在する．Rook Ceph には PV のための OSD Pod が存在する．

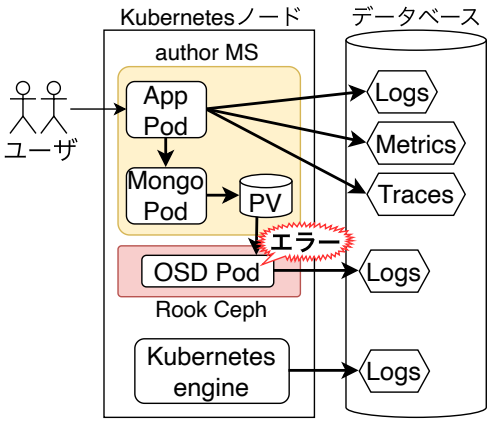

図 1　Doktor で発生した障害の概要

システム管理者は，Doktor での障害に関するアラートを受信すると，障害の原因を調査するために Kubernetes クラスタのメトリクスやログ，トレースを調査する．システム管理者が Rook Ceph を障害の原因として特定するまでのプロセスを以下に示す．

(1) 障害の発生時刻，影響しているマイクロサービスやコンテナをメトリクスやアラートから特定した．
(2) 分散トレーシングを使用して影響しているマイクロ

[1] 東京工科大学大学院 バイオ・情報メディア研究科 コンピュータサイエンス専攻
[a)] d212400159@edu.teu.ac.jp
[b)] kushida@acm.org
[c)] ikuno@stf.teu.ac.jp
[*1] https://doktor.tak-cslab.org
[*2] https://rook.io/

プログラム 1 Rook Ceph の OSD Pod のエラーの状態

```
1 $ kubectl get pod -A -o wide | grep osd
2 rook-ceph rook-ceph-osd-0-5d7d4b4648-d8tbr 2/2
   Running ...
3 rook-ceph rook-ceph-osd-1-867ffb7b64-279pp 2/2
   Running ...
4 rook-ceph rook-ceph-osd-2-5dbc5f798c-flwh4 2/2
   Running ...
5 rook-ceph rook-ceph-osd-3-6c5c8bf688-n4d5z 0/2
   Init:CrashLoopBackOff ...
```

サービスやコンテナに関連するリクエストを調査したものの原因が特定できなかった．

(3) Kubernetes クラスタの Pod の起動状態や Pod や PV のログやイベントを確認した．ノードのハードウェアリソースの使用率をメトリクスで確認した．

(4) author MS の Pod が CrashLoopBackOff 状態であることを確認した．

(5) クラスタ全体で Running でない Pod を調査した．

(6) Rook Ceph の OSD Pod が Init:CrashLoopBackOff 状態であることを確認した．

(7) Rook Ceph の OSD Pod に含まれるコンテナのログを調査した．

上記の調査から障害の原因は Rook Ceph の OSD Pod でのエラーであった．プログラム 1 に Rook Ceph の OSD Pod のエラーの状態を示す．コマンド `kubectl get pod -A -o wide` により OSD Pod の状態を取得できる．OSD（Object Storage Daemon）はストレージサーバを稼働させるプロセスである[*3]．Kubernetes クラスタには 4 つの OSD Pod が存在していた．このうち 1 つの OSD Pod の “rook-ceph-osd-3-6c5c8bf688-n4d5z” が CrashLoopBackOff であった．その結果，author MS の Mongo Pod が永続ボリューム (PV) のマウントに失敗した．その結果，Doktor の author MS でエラーが発生した．

プログラム 2 にエラーに関連する OSD Pod のログを示す．ログの 1 行目はログを取得するために実行した kubectl logs コマンドである．2 行目はログメッセージである．このログメッセージは RocksDB に関して SST（Sorted String Table）ファイルが WAL より先に進んでおり不整合が起きていることを表す．RocksDB(rocksdb) は BlueStore のメタデータ保存に利用されるデータベースである．3 行目のログメッセージは BlueStore(bluestore) で RocksDB を開く際にエラーが発生したことを表す．BlueStore は Rook Ceph のコンポーネントの 1 つでストレージのバックエンドである．

一連の障害の原因調査のプロセスでは，テレメトリ (トレースやログ，メトリクス) を組み合わせて使用していた．システム管理者がテレメトリを確認するには，それぞれ別

[*3] https://docs.ceph.com/en/reef/start/beginners-guide/

プログラム 2 エラーに関連する OSD Pod のログ

```
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:
```

の方法が必要である．例えば，Kubernetes クラスタのリソースの状態を確認するには，kubectl describe コマンドや kubectl get コマンドを実行する．また，アプリケーションのログを確認するには，ログサーバに Web UI から検索クエリを発行する．こうした複数のテレメトリから手動で障害の原因に関連する情報を探す作業は，時間のかかる作業であり間違いを起こしやすい [4], [5], [6]．22 件のシステム障害に対する調査の結果は，大半の障害の調査でシステム管理者がログの閲覧やトレースの閲覧に 1 時間以上かかることを示している [7]．

トラフィックの増加やシステムを構成するソフトウェアコンポーネント数の増加に伴い，テレメトリの件数は増加する．チャットアプリの WeChat のシステムでは，1 日あたり 16-20 ペタバイトのログが出力される [8]．動画ストリーミングサービスの Netflix のシステムでは，200 万に及ぶメトリクスが存在する [9]．こうした多くのテレメトリから必要な情報同士を手動で対応付けて原因を調査する作業は，システム管理者にとって時間のかかる作業である．

システムでの障害が複雑になるほど，障害にはアプリケーションに加えてミドルウェアが関連する [10], [11]．Microsoft 社での調査結果は，ミドルウェアやインフラストラクチャに関連する障害が全体の 26.3%であることを示している [12]．テレメトリはアプリケーションだけでなくサーバやコンテナに関連するミドルウェアからも出力される [13]．システム管理者は障害の原因調査でアプリケーションやミドルウェアのテレメトリのそれぞれの調査を行う．また，システムの運用期間が長くなるにつれ，単純な障害はテストや段階的なロールアウトにより発生しなくなり，複雑な障害が残り続ける [14]．システムでの障害が複雑になるほどアプリケーションとミドルウェアの両方に関わる障害の発生は，システム管理者の原因調査の難易度を高めている．

課題

Rook Ceph のエラーによる障害では，システム管理者はエラーの原因を特定するために，ミドルウェアのログやトレースを手動で確認する．マイクロサービスアーキテクチャに代表される分散型のアーキテクチャでは，モノリシックアーキテクチャに比べてソフトウェアコンポーネントの

数が多く，調査対象の箇所が多くなる．そのため，手動での調査には専門的な知識が必要であり，運用経験の浅いシステム管理者には難易度の高い作業である．そのため，障害の原因調査で経験の浅いシステム管理者は試行錯誤を繰り返す必要があり，時間のかかる作業になっている．障害の原因調査にかかる時間は MTTR(Mean Time to Repair) に含まれており，短時間での原因調査が必要である．

アプリケーションのテレメトリの収集と分析を行う研究が存在するが，ミドルウェア上のリソースの依存関係や内部イベントを含めた複数のデータソースを使用した障害の原因調査の手法が十分に研究されていない．そのため，システム管理者はアプリケーションで発生したエラーの原因調査のために複数のテレメトリのデータソースを手動で確認して原因調査を行う必要がある．

各章の概要

2 章では複数のデータソースを使用した障害の原因調査とミドルウェアのテレメトリの収集に関する関連研究を紹介する．3 章では提案手法を紹介する．4 章では提案手法の評価について述べる．5 章では提案に関する議論を行う．6 章では本稿のまとめを行う．

2. 関連研究

単一のデータソースを使用した障害の原因調査

障害の原因調査はシステム管理者にとって時間のかかる作業であるため，自動化の研究がされてきた．障害の原因調査では，単一のデータソースを使用した手法や，複数のデータソースを使用した手法，グラフを使用した手法が提案されてきた．単一のデータソースを使用した手法としてメトリクスを使用した手法がある [6], [15], [16], [17]．単一のデータソースにトレースを使用した手法が提案されている [18], [19]．また，単一のデータソースにログを使用した手法がある [20], [21]．これらの単一のデータソースを使用した手法は，データソースにしたテレメトリ以外に障害の兆候が含まれる場合に問題を特定できず，障害の原因特定の精度に課題がある．

複数のデータソースを使用した障害の原因調査

複数のデータソースを使用した障害の原因調査の手法が提案されている [4], [22], [23], [24]．これらの手法では障害の原因調査に複数のデータソースのテレメトリを使用している．DeepTraLog では複数のデータソースのテレメトリを使用した深層学習による異常検知の手法が提案されている [22]．ログやトレースを使いアプリケーションの依存関係を収集し，依存関係のグラフを構築した．この手法ではミドルウェアに関連するリソースのトレースやログを使用していない．この研究では異常検知に着目しており，障害の原因調査に対する手法ではない．そのため，ミドルウェアが原因で発生した障害の原因調査には適さない．Eadro では教師ありの機械学習を使い複数のデータソースのテレメトリを使った原因調査を提案している [23]．教師なしの機械学習に比べて Eadro は高い精度を示した．Eadro の使用には教師データへのラベル付与が必要であり，これはシステム管理者にとって負担の大きい作業である．こうした作業を行うことは実際の運用では現実的ではない．Nezha ではトレースとログを統合し，リクエスト単位で詳細な一連の処理プロセスの追跡を可能にした [4]．この手法ではミドルウェアのログとアプリケーションのトレースを対応付ける方法が確立されておらず，ミドルウェア原因で発生した障害の原因調査には適さない．過去に発生した障害から単純なイベントの因果グラフを求め重みを計算することで障害の原因調査を行っている [24]．この研究では過去の障害をデータセットに使用している．過去におきた障害には再発防止が行われており再発する可能性が低いため適用できるケースが限定的である．

3. 提案手法

本稿ではミドルウェアから出力されるイベントと依存グラフ，トレースを使用した障害の原因調査の手法 (Event RCA) を提案する．Event RCA はシステム管理者が障害の発生した際に，障害原因の候補をリストアップする．システム管理者はミドルウェアで発生したエラーに伴いアプリケーションのエラーが発生した場合に，Event RCA を使うことでエラーの原因箇所のリソース (例: コンテナ) を探す作業が不要になる．障害の原因調査にかかる時間や正確さは，システム管理者の技量や運用の経験に依存しやすい．Event RCA による自動化は，こうしたシステム管理者の技量や運用の経験が十分でないシステム管理者にも，熟練したシステム管理者に近い正確さで障害の原因特定を可能にする．これにより手動での障害の原因調査を削減でき，障害の原因特定にかかる時間の短縮が実現される．

図 2 に提案手法の概要を示す．図では Kubernetes[*4] クラスタ上で動作するマイクロサービスアプリケーションを対象に，障害の原因調査を行う．マイクロサービスは障害の原因調査を行う対象のアプリケーションである．マイクロサービスをまたぐ一連のリクエスト処理を追跡するために，トレースが収集されている．Kubernetes クラスタにはマイクロサービスが配置されている．提案手法では Kubernetes クラスタから Kubernetes のリソース同士の依存グラフを収集する．依存グラフは動作中のリソースの一覧を API で取得し，リソース名をもとに作成される．例えば，コンテナをあらわすリソースである Pod は，別の Pod からアクセスするためのリソースである Service に依存する．また，Pod は永続ディスクの要求をあらわすリソー

[*4] https://kubernetes.io/

スの PVC（Persistent Volume Claim）に依存する．また，PVC は永続ディスクをあらわすリソースの PV（Persistent Volume）に依存する．こうしたリソース同士の依存関係をもとに依存グラフを作成する．

図 2 のイベントはリソースのライフサイクルに関連するイベントを表す．Kubernetes ではリソースの作成や更新，削除が発生すると，イベントが作成される．プログラム 3 に Kubernetes のイベントの例を示す．イベントには Type, Reason, Age, From, Message の属性がある．Type にはイベントの状態が含まれる．Reason にはイベントの種類が含まれる．Age にはイベントが発生したタイムスタンプが含まれる．From にはイベントが発生したリソースを表す．Message にはイベントの詳細が含まれる．例えば，4 行目のイベントでは Pod(default/ubuntu) が Kubernetes ノードの clematis-worker2 へのスケジューリングに 69 秒前に成功したことをあらわす．

Event RCA は提案手法である．Event RCA ではトレース，依存グラフ，イベントを入力として受け取り，障害の原因調査を行い，障害の原因箇所をリストで出力する．出力には確率をあらわすスコアと対象リソースのペアがリストとして含まれる．例えば，スコアが 0.9 で対象リソースが Pod A の場合には，Pod A が障害の原因である確率が 90%である．スコアは 0 から 1 の範囲の小数として出力される．出力はスコアの降順でソートされる．

図 2 の一連の障害の原因調査の過程を述べる．一連の過程は，障害が検知された時点で開始される．(1) では次のデータを収集する．

- マイクロサービスからトレースを収集する．トレースの収集は既存の OpenTelemetry SDK[*5]や Zipkin ライブラリ[*6]に代表されるライブラリを使用する．
- Kubernetes クラスタから依存グラフを収集する．依存グラフの収集には Kubernetes API を使用し，リソース名をもとに依存グラフを収集する．エラーを含むトレースに対応する Pod を起点に依存するリソースを収集する．
- Kubernetes クラスタからイベントを収集する．イベントの収集には Kubernetes API を使用する．収集対象のイベントは，依存グラフに含まれるリソースに関連するイベントである．

(2) ではトレース，依存グラフ，イベントを Event RCA に入力し，障害の原因調査を行う．(3) で Event RCA は障害の原因調査の結果を出力する．

Event RCA

アルゴリズム 1 に Event RCA の内部処理を示す．入力にはトレースの一覧 (*traces*)，依存グラフ (*dependencyGraph*)，イベントの一覧 (*events*) をとる．出力は障害の原因箇所のスコアの一覧 (*endToEndEventCounts*) である．障害の原因箇所のスコアの一覧は，リソース名とスコアのペアの一覧である．スコアの一覧はスコアの降順でソートされている．1 行目から 8 行目では入力されたトレースの一覧からエラーを含むトレースを探す．トレースを構成するスパンにエラーが含まれていた場合に，*errorTraces* にスパンの Pod 名とタイムスタンプのペアを追加する．これによりトレースの一覧からエラーを含むトレースだけを取り出す．9 行目から 15 行目ではエラーを含むスパンに対応する Pod に関連するリソースの依存グラフを作成する．9 行目の LOADMAPPINGFILE 関数では事前に作成したリソースの種類ごとに依存関係をもつ設定項目の一覧 (依存項目リスト) を読み込む．表 1 に依存項目リストの例を示す．リソースが Pod である場合，別のリソースである ConfigMap との依存関係は，リソース定義にある.spec.volumes[].configMap.name と.spec.env.valueFrom.secretKeyRef.name に指定されている．また，PVC(Persistent Volume Claim) のリソー

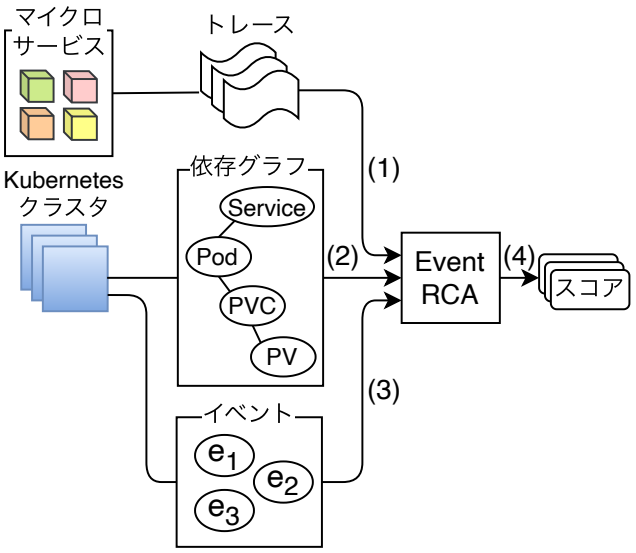

図 2　提案手法の概要

プログラム 3　Kubernetes のイベントの例

```
1  Events:
2    Type Reason Age From Message
3    ---- ------ ---- ---- -------
4    Normal Scheduled 69s default-scheduler
     Successfully assigned default/ubuntu to
     clematis-worker2
5    Normal Pulled 69s kubelet Container image "
     ubuntu:24.04" already present on machine
6    Normal Created 69s kubelet Created container
      ubuntu
7    Normal Started 69s kubelet Started container
      ubuntu
```

*5　https://opentelemetry.io/docs/languages/
*6　https://zipkin.io/pages/tracers_instrumentation

表 1 リソースの種類ごとに依存関係をもつ設定項目の一覧の例

依存元	依存先	抽出方法
Pod	ConfigMap	.spec.volumes[].configMap.name, .spec.env.valueFrom.secretKeyRef.name
PVC	PV	.spec.volumeName, .spec.storageClassName

スが永続ディスクをあらわすリソースである PV(Persistent Volume) との依存関係は，リソース定義にある.spec.volumeName と.spec.storageClassName に指定されている．10 行目では List 型 *routeList* を定義しており，この変数にはリソース間の依存関係をエンドツーエンド (始点から終点) が記録される．11 行目から 13 行目ではエラーを含むトレース *errorTraces* の中から Pod 名とタイムスタンプのペアを取り出し，BUILDDEPGRAPH 関数を呼び出す．BUILDDEPGRAPH 関数はリソース名を引数とする．BUILDDEPGRAPH 関数はリソース名をもとに依存関係のあるグラフを LOADMAPPINGFILE 関数から読み込んだ依存項目リストをもとに作成する．14 行目から 26 行目では，BUILDDEPGRAPH 関数を定義している．15 行目の GETRESOURCEDEF ではリソース定義を取得する．16 行目ではリソース定義をパースし，設定項目であるフィールドに分割する．17 行目から 25 行目では，フィールドの要素が依存項目リストに含まれているか確認する．含まれている場合には BUILDDEPGRAPH 関数を呼び出す．含まれない場合には，リソース間の依存関係を *routeList* に追加する．29 行目から 39 行目では依存グラフから原因を特定する．33 行目の GETERREVENTWITHIN5MIN では，トレースの時刻をもとにエラーの発生した前後 5 分のイベントのうちエラーのものの一覧を取得している．40 行目から 45 行目では，依存グラフからイベントの数からスコアを計算している．イベントの数は，依存グラフからエンドツーエンドのリソースの数を取得し，その数をトレースの数で割ることで計算される．

ユースケース・シナリオ

図 3 にユースケース・シナリオを示す．図 3 はシステム管理者が分散トレーシングを使いシステム障害の原因を調査する状況をあらわす．分散トレーシング Web UI は分散トレーシングのためのユーザーインターフェースである．システム管理者はシステム障害が発生した際にシステム障害の原因箇所を見つけるために Web UI の個別トレース画面にアクセスする．Web UI にはタイムスタンプ順に並べられた一連のリクエスト処理が表示される．1 つのトレースは複数のスパンで構成されており，システム管理者はエラーを含むスパンを Web UI を使い見つける．エラーがスパン間で伝搬されている場合には，最初にエラーの発生したスパンを探す．個別トレース画面では，エラーの発生している原因のスパンやマイクロサービスの特定が可能である．一方で，エラーの原因がミドルウェアにある場合には，原因の調査が分散トレーシング Web UI だけでは行えない．

分散トレーシング Web UI に提案手法を表示するため

アルゴリズム 1 Event RCA の内部処理

Require: $traces, dependencyGraph, events$
Ensure: $endToEndEventCounts$ ▷ Map[str, float]

/* エラーを含むスパンを探す */
1: $errorTraces \leftarrow \emptyset$ ▷ map[str, str]
2: **for** $trace \leftarrow traces$ **do**
3: **for** $span \leftarrow trace$ **do**
4: **if** $span$ contains error **then**
5: $errorTraces[span.podName] \leftarrow span.timestamp$
6: **end if**
7: **end for**
8: **end for**

/* エラーを含むスパンに対応する Pod に関連するリソースの依存グラフを作成する */
9: $fieldMap \leftarrow$ LoadMappingFile ▷ Map[str, str]
10: $routeList \leftarrow \emptyset$ ▷ List[str]
11: **for** $podName, timestamp \leftarrow errorTraces$ **do**
12: BuildDepGraph($podName$)
13: **end for**
14: **function** BuildDepGraph($rsrcName, rsrcChain$)
15: $resourceDef \leftarrow$ GetResourceDef($rsrcName$)
16: $parsedDef \leftarrow$ ParseResourceDef($resourceDef$)
17: **for** $fieldName \in parsedDef$ **do**
18: $childName \leftarrow fieldMap[fieldName]$
19: **if** $childName \neq \emptyset$ **then** ▷ has child resource
20: $rsrcChain \leftarrow rsrcChain + " > " + childName$
21: BuildDepGraph($childName, rsrcChain$)
22: **else** ▷ has no child resource
23: $routeList \leftarrow routeList \cup \{rsrcChain\}$
24: **end if**
25: **end for**
26: **end function**

/* 依存グラフから原因を特定する */
27: $endToEndEventCounts \leftarrow \emptyset$ ▷ Map[str, float]
28: $totalEventCount \leftarrow 0$
29: **for** $routeChain \leftarrow routeList$ **do**
30: $rsrcNames \leftarrow routeChain.split(" > ")$
31: $eventCount \leftarrow 0$
32: **for** $rsrcName \leftarrow rsrcNames$ **do**
33: $events \leftarrow$ GetErrEventsWithin5Min($rsrcName$)
34: $eventCount \leftarrow eventCount +$ length(events)
35: **end for**
36: $lastRsrcName \leftarrow rsrcNames[length(rsrcNames) - 1]$
37: $endToEndEventCounts[routeChain] \leftarrow eventCount$
38: $totalEventCount \leftarrow totalEventCount + eventCount$
39: **end for**

/* スコアの計算を行う */
40: **for** $routeChain, eventCount \leftarrow endToEndEventCounts$ **do**
41: $rsrcNames \leftarrow routeChain.split(" > ")$
42: $score \leftarrow eventCount/totalEventCount$
43: $lastRsrcName \leftarrow rsrcNames[length(rsrcNames) - 1]$
44: $endToEndEventCounts[lastRsrcName] \leftarrow score$
45: **end for**
46: **return** SortByScore($endToEndEventCounts$)

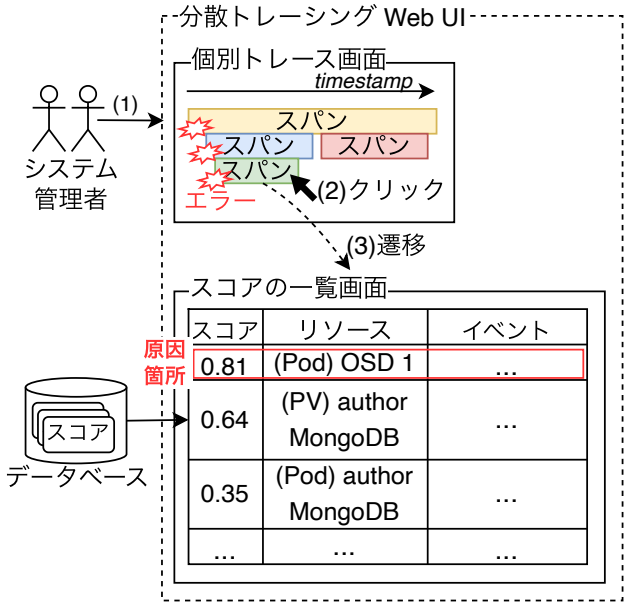

図 3 ユースケース・シナリオ

に，スコアの一覧画面を追加する．スコアの一覧画面では，原因の確率をあらわすスコア，原因と想定したリソース，リソースに関連したイベントが表示される．スコアが高いほど上位の行に表示される．スコアの一覧画面に表示するスコアは，データベースに保存されている．データベースに保存されているスコアは，Event RCA からの出力されたものである．

4. 評価

評価方法

提案手法による障害の原因調査の出力を評価するために，上位 5 件の適合率 (Precision) と再現率 (Recall) を計測する．これらの指標は原因調査の研究で評価指標として使用されている [17]．出力された障害箇所の上位 5 件の中に原因が含まれる割合を計測する．エンジニア 386 人への調査結果によると，障害の原因調査の出力のうち重視する結果は上位 5 件であった [25]．そのため，提案手法により出力された原因箇所の上位 5 件の中に原因が含まれる割合を評価指標に使用する．

一連の障害対応のプロセスには，障害の原因特定と対処が含まれる．障害の原因調査にかかる時間が提案手法により短縮されたかを評価するために，実際にシステム管理者により障害の原因調査にかかる時間を計測する．比較対象は手動での障害の原因調査，Event RCA を使用した原因調査，関連研究の手法を使用した障害の原因調査である．

図 4 に評価方法を示す．障害シナリオをシステムで発生した障害を記録したシナリオである．Chaos Mesh はシナリオをもとに障害を再現するツールである．Kubernetes クラスタでは Web アプリケーションの Doktor が動作している．Doktor に Chaos Mesh で障害を発生させ，依存グラフやイベントを収集する．Doktor のトレースやメトリクスはデータベースの SigNoz に保存する．Event RCA は提案ソフトウェアであり，Event RCA は依存グラフ，イベント，トレースをもとに障害の原因調査を行う．Event RCA はスコアと原因箇所のペアのリストであるスコアリストを出力する．Evaluator は評価を行うソフトウェアであり，障害シナリオに含まれる障害の原因箇所，スコアリストに含まれる障害の原因箇所を比較する．

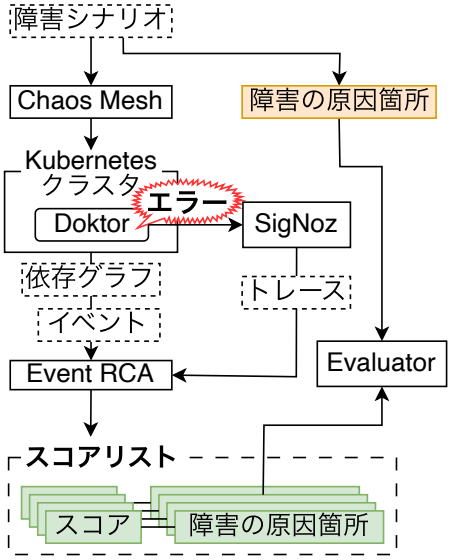

図 4 評価方法

障害シナリオ

実際に発生したシステム障害をもとに障害シナリオを 3 つ作成する．1 つ目のシナリオは Rook Ceph での OSD Pod のエラーである．このシナリオでは Rook Ceph が Kubernetes クラスタで動作しており，Kubernetes クラスタの 1 ノードが応答しないため再起動を行った．再起動の際に Pod が正常に終了せずデータの破損が発生した．このシナリオの再現には Kubernetes クラスタの 1 ノードをハードウェア再起動を行う．

2 つ目のシナリオは Kubernetes のコントロールプレーンに対する過剰なリクエスト送信である．このシナリオは OpenAI 社の Kubernetes クラスタで発生した障害のポストモーテム[*7]から作成した．Kubernetes クラスタに配置された監視用のソフトウェアが過剰に Kubernetes クラスタのコントロールプレーンにリクエストを送信し，コントロールプレーンが過負荷になった．このシナリオでは Kubernetes クラスタに cAdvisor を配置し，スクレイピングの間隔を数秒ごとに行いコントロールプレーンに対する過剰なリクエストを送信する．

3 つ目のシナリオでは DoorDash 社の Kubernetes クラスタで発生した障害のポストモーテム[*8]から作成した．こ

*7 https://status.openai.com/incidents/ctrsv3lwd797
*8 https://careersatdoordash.com/blog/how-to-handle-

の障害が発生した際には Kubernetes クラスタに配置されたアプリケーションコンテナの Readiness Probe が 1 秒に設定されていた．また，Readiness Probe でチェックされる API エンドポイントには Redis へのアクセスが含まれていた．Redis サーバのレイテンシが 1 秒を超えたときに，アプリケーションサーバの Readiness Probe が 1 秒を超え，ヘルスチェックに失敗した．その結果，新たなアプリケーションコンテナの起動に失敗した．このシナリオでは Readiness Probe のレイテンシを 1 秒に設定した状況で，Chaos Mesh でアプリケーションサーバのレイテンシに 2 秒を追加で付与する．

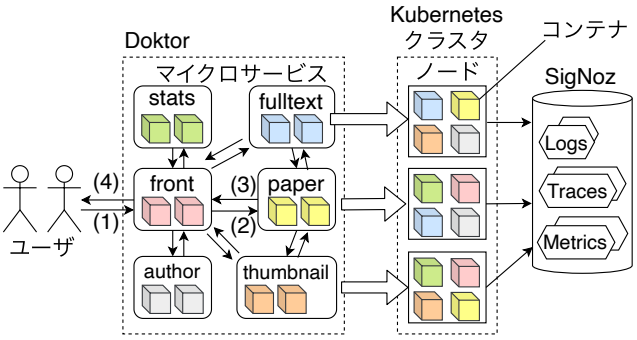

図 5 Doktor のシステム構成

実験環境

実験にはマイクロサービスアーキテクチャで設計された Web サービスである Doktor を使用する．図 5 に実験環境を示す．Doktor は複数のマイクロサービス (stats, front，paper，fulltext，author，thumbnail) から構成される．これらのマイクロサービスはコンテナとして動作している．各マイクロサービスのアプリケーションコンテナには OpenTelemetry SDK が導入されており，ログやトレース, メトリクスが収集されている．アプリケーションコンテナは Kubernetes クラスタに配置されている．Kubernetes クラスタは 4 台の仮想マシンで構成されている．マスターノードは 1 台の仮想マシンで構築されている．ワーカノードは 3 台の仮想マシンで構成されている．Kubernetes クラスタのノードを構成する全てのマシンは同種のハードウェア構成 (vCPU: 8 コア，メモリ: 8GB，ディスク: 40GB) である．Kubernetes エンジンには K3s(バージョン: 1.31.5) を使用した．監視には SigNoz を使用している[*9]．SigNoz はテレメトリの保存および取得が可能な監視システムである．Doktor の各マイクロサービスが生成するテレメトリは OpenTelemetry により収集され，SigNoz に転送・保存される．

リクエスト処理フローは以下のように動作する．(1) ユーザがウェブブラウザからリクエストを送信すると，front サービスがリクエストを受け付ける．(2)front サービスはアクセスされた URL に応じて各マイクロサービスに HTTP リクエストを送信する．(3) 各マイクロサービスは HTTP リクエストを受け取るとリクエストに応じた情報を front マイクロサービスに返す．(4) これらの情報は front サービスに統合され，最終的に HTML を含む HTTP レスポンスとして利用者に返される．

5. 議論

提案手法では同一のトレースに複数の異なる原因のエラーが含まれているとそれらを分けて障害の原因調査ができない．大規模な商用環境のシステムでは周期的に繰り返し発生するエラーがある [21]．システム障害に関連したトレースの中に周期的に繰り返し発生するエラーと障害原因のエラーが含まれていると，提案手法ではそれらを識別できない．そのため，周期的に繰り返し発生するエラーを含めた原因調査が行われる．解決策の 1 つは，周期的に繰り返し発生するエラーとそれ以外のエラーの識別によるフィルタリングである．平常時のトレースを収集し，トレースに含まれる宛先 URL とトレースに含まれるスパンのエラーメッセージをデータベースに記録しておく．障害発生時にはデータベースに保存された宛先 URL とスパンのエラーメッセージを比較し，周期的に繰り返し発生するエラーであるか比較する．これにより同一のトレースに複数のエラーが含まれる場合に，それらを分けた障害の原因調査が可能になる．

6. おわりに

マイクロサービスアーキテクチャで設計されたシステムでの障害の原因調査では，システム管理者は複数のテレメトリを組み合わせて使用する．ある障害が起きたときに，アプリケーションでのエラーの原因がミドルウェアにあると，その原因の調査には複数のデータソースを解析する必要がある．こうした作業はシステム管理者が手動で行っており，システム管理者にとって時間のかかる作業になっている．本稿ではミドルウェアから出力されるイベントと依存グラフ，トレースを使用した障害の原因調査の手法を提案する．提案手法ではアプリケーションから出力されたトレースに関連する依存グラフを作成する．依存グラフをもとにリソースごとのイベントの時系列の順序を求め，障害原因の候補をリストアップする．インターネット上に公開したマイクロサービスアーキテクチャで設計された Web アプリケーションを対象に障害を再現し，提案手法の障害原因の候補に実際の原因が含まれるかを評価する．

謝辞 本研究は, JSPS 科研費 JP23K11073, JP23K11087 の助成を受けたものである．

kubernetes-health-checks/

*9 https://signoz.io/

参考文献

[1] Mendonça, N. C., Box, C., Manolache, C. and Ryan, L.: The monolith strikes back: Why istio migrated from microservices to a monolithic architecture, *IEEE Software*, Vol. 38, No. 05, pp. 17–22 (2021).

[2] Thönes, J.: Microservices, *IEEE software*, Vol. 32, No. 1, pp. 116–116 (2015).

[3] Weil, S., Brandt, S. A., Miller, E. L., Long, D. D. and Maltzahn, C.: Ceph: A scalable, high-performance distributed file system, *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, pp. 307–320 (2006).

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

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

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

[7] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W. and Ding, D.: 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).

[8] Yu, G., Chen, P., Li, P., Weng, T., Zheng, H., Deng, Y. and Zheng, Z.: Logreducer: Identify and reduce log hotspots in kernel on the fly, *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, pp. 1763–1775 (2023).

[9] Thalheim, J., Rodrigues, A., Akkus, I. E., Bhatotia, P., Chen, R., Viswanath, B., Jiao, L. and Fetzer, C.: Sieve: Actionable insights from monitored metrics in distributed systems, *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*, pp. 14–27 (2017).

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

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

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

[13] Gu, S., Rong, G., Ren, T., Zhang, H., Shen, H., Yu, Y., Li, X., Ouyang, J. and Chen, C.: Trinityrcl: Multigranular and code-level root cause localization using multiple types of telemetry data in microservice systems, *IEEE Transactions on Software Engineering*, Vol. 49, No. 5, pp. 3071–3088 (2023).

[14] Huang, P., Guo, C., Lorch, J. R., Zhou, L. and Dang, Y.: Capturing and enhancing in situ system observability for failure detection, *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 1–16 (2018).

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

[16] Pham, L., Ha, H. and Zhang, H.: Baro: Robust root cause analysis for microservices via multivariate bayesian online change point detection, *Proceedings of the ACM on Software Engineering*, Vol. 1, No. FSE, pp. 2214–2237 (2024).

[17] Lin, J., Chen, P. and Zheng, Z.: Microscope: Pinpoint performance issues with causal graphs in micro-service environments, *International Conference on Service-Oriented Computing*, Springer, pp. 3–20 (2018).

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

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

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

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

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

[23] Lee, C., Yang, T., Chen, Z., Su, Y. and Lyu, M. R.: Eadro: An end-to-end troubleshooting framework for microservices on multi-source data, *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, pp. 1750–1762 (2023).

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

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