Vol.2022-IOT-56 No.37
2022/3/8

マイクロサービスにおけるコンポーネント間の依存関係に着目した障害原因箇所特定手法の提案

土手 貴裕[1,a)]　近堂 徹[2,b)]　前田 香織[1,c)]　今村 光良[3]　高野 知佐[1,d)]

概要：IT システムの新たなアーキテクチャとしてマイクロサービスが注目されている．マイクロサービスではシステムが持つ各機能を独立したコンポーネントとして分割し API で疎結合させることで，機能単位での頻繁な改修が容易となり，顧客の要望への迅速な対応が可能となる．しかし，マイクロサービス化によりシステム構成の複雑性とシステム状態を把握するための時系列データであるメトリックの数が増大するため，障害原因であるコンポーネント（障害原因箇所）の特定が困難となる．本研究では，マイクロサービスにはコンポーネント間で API 呼び出しによる依存関係があることに着目し，メトリックの定常時からの変化量にコンポーネント間の依存関係を組み合わせることで障害原因箇所を特定する手法を提案する．また，提案手法による障害原因箇所特定システムを開発し，EC サイトを模したマイクロサービスのベンチマークを用いた実験を行った．特定精度および特定に要する時間について評価を行い，これらの結果から提案手法の有効性を示す．

Development of a Failure Cause Identification System Focused on the Component Dependencies in Microservices

1. はじめに

EC サイトや動画配信サービス，ソーシャルゲームなどを実現する近年の IT システムは，ユーザからの様々な要求に対応するために機能の追加や改修などの変更が多く加えられ大規模化しつつある．クラウド上に構築された大規模システムは，ユーザからの新たな要求に対応する際，その各機能が密接した大きな規模に対して新たに変更を加える必要があるため，変更による他機能への影響が大きくなる．そこで，大規模システムであってもユーザからの要求に容易に対応できるアーキテクチャとして注目されているマイクロサービス [1] は，システムがもつ各機能を独立したコンポーネントに分割し，ネットワークを介してそれらを API で疎結合し協調動作させるため，機能変更によるシステム全体への影響を最小限にしつつ，システムの負荷状況に応じたスケールを機能単位で行うことが可能である．

マイクロサービス化されたシステムの監視には，各コンポーネントの応答時間（API 呼び出しによるリクエストの処理時間）や各コンポーネント内のコンテナの CPU 使用率などのメトリック（時系列データ形式で収集される，システムの健全性を把握するための定量的な指標）をグラフ化し，複数のグラフの関係が一目見て把握しやすい形に集約されたダッシュボードを用いることが一般的である [2]．また，ダッシュボードの構築方法を発展させてコンポーネント間の関係性を踏まえた監視を行う場合もある．しかし，マイクロサービスが大規模であるほどネットワーク上に分散配置されるコンポーネント数や監視メトリック数，コンポーネント間の関係の複雑性などが増大し，システム管理者がマイクロサービス全体の構成と状態を把握することが困難となる．

そこで本研究では，マイクロサービスにおける障害発生時の迅速かつ正確な障害原因箇所の特定を行うことを目的とし，既存手法における課題を踏まえた上で，メトリックの定常時からの変化量とコンポーネント間の依存関係を組

[1] 広島市立大学大学院情報科学研究科, Graduate School of Information Sciences, Hiroshima City University, Hiroshima 731-3194, Japan
[2] 広島大学情報メディア教育研究センター, Information Media Center, Hiroshima University, Hiroshima 739-8511, Japan
[3] 野村アセットマネジメント株式会社, Nomura Asset Management Co., Ltd., Toyosu, Koto-ku, Tokyo 135-0061, Japan
[a)] dote@v6.netsci.info.hiroshima-cu.ac.jp
[b)] tkondo@hiroshima-u.ac.jp
[c)] kaori@hiroshima-cu.ac.jp
[d)] takano@hiroshima-cu.ac.jp

み合わせることで障害原因箇所を特定する手法を提案する．

本稿では，まず 2 章でマイクロサービスにおける既存の障害原因箇所特定手法について述べる．次に，3 章で提案する障害原因箇所特定手法について述べ，4 章では提案手法に基づいた障害原因箇所特定システムの実装について述べる．5 章では，提案手法による特定精度と特定時間について，マイクロサービスのベンチマークと開発した障害原因箇所特定システムを用いた実験による評価を述べる．最後に，6 章でまとめと今後の展望について述べる．

2. 関連研究

既存の障害原因箇所特定手法として，まず定常時と負荷注入時のトレースログを機械学習させる手法がある [3]．トレースログとは，ユーザからのリクエストを処理するためにマイクロサービス全体をまたいだコンポーネントの API 呼び出しによる通信のログを指す．トレースログはコンポーネント間の依存関係や応答時間など，あるリクエストを処理した際の一連の流れを追跡した情報が含まれるため，ネットワーク的に構成が複雑なマイクロサービスにおいて障害原因箇所を特定する際に切り分けるための情報として有効であるが，CPU 使用率やメモリ使用量といったリソースに関する情報は含まれない．マイクロサービスでは，コンポーネント間の接続関係が疎結合であり，またコンポーネントごとに障害発生までに耐えられる負荷の大きさも異なるため，あるコンポーネントでリソースを占有する負荷が発生してもその影響をトレースログで観測することが難しい．そのため，トレースログによる観測が難しい CPU 高負荷やメモリリークなどのリソース系の障害に対する特定精度が低下する課題が挙げられている．

次に，複数のメトリック間の因果関係からコンポーネント間の関係性を推定した手法 [4–8] がある．[4,5] は，ユーザに最も近い，すなわちコンポーネント全体の中で最前段にあるコンポーネントを起点として因果探索を行うため，最前段への影響が小さいコンポーネントが障害原因箇所となるパターンに対する特定精度が低下する課題がある．これらに対し [6] は，各コンポーネントの応答時間も利用することで精度を改善しているが，コンポーネント間の関係性の推定にリソース使用率と応答時間の相関値を利用しており，これらの相関が強いものを中心に因果探索を行うため，リソース使用率と応答時間の間の関係性が弱いパターン（CPU が高負荷でも応答時間が増加しないパターンなど）に対する特定精度が低下する課題がある．

一方，マイクロサービスでは任意のコンポーネントを処理するために複数のコンポーネントが API 呼び出しにより連携しあうため，コンポーネント間に依存関係が存在する．依存関係は様々な形で形成されるため，依存関係によっては依存関係のある別のコンポーネントの処理待ちとなることや，あるコンポーネントにおける障害の発生が他のコンポーネントのメトリック変動を生じさせることもある．よって，コンポーネント間の依存関係は，コンポーネント間の関係性のモデル化において重要な情報であり，また任意のコンポーネントが異常であってもそれが依存するコンポーネントも見ることで障害原因箇所であるかの判断の一助となるため，障害原因箇所であるかどうかを切り分ける情報としても有効である．しかし，メトリックからコンポーネント間の関係性を推定した手法 [4–8] では，このコンポーネント間の依存関係を捉えること，及び異常なメトリックがどのコンポーネントに影響されたものであるかの考慮はされていない．

他方では，学習済みの機械学習モデルから得られる異常度に対する各メトリックの貢献度を，機械学習の解釈手法として注目されている協力ゲーム理論のシャープレイ値 [9] を用いて算出することで，障害に最も関与したメトリックを特定する手法 [10] もある．一般的にシャープレイ値の計算量はその特徴量（メトリックの系列数）の数に応じて増大するが，[10] では独自のライブラリを用いて高速化を実現している．

3. 障害原因箇所特定手法の提案

マイクロサービスにおける障害原因箇所の特定を正確に行うには，2 章で述べた現行の手法の課題を解決するために以下の 3 点を考慮する必要がある．

(1) エッジ特性のメトリック（コンポーネントの応答時間，受信リクエスト数など）とコンポーネント特性のメトリック（コンテナの CPU 使用率，メモリ使用量など）の両方を使用すること．
(2) メトリック間の相関値ではなく，メトリック個々の変化に着目すること．
(3) API 呼び出しによるコンポーネント間の依存関係を直接捉えたモデル化を行うこと．

これらの点を考慮した障害原因箇所の特定手法として，エッジ特性とコンポーネント特性両者のメトリックの定常時からの変化量にコンポーネント間の依存関係を組み合わせた手法を提案する．

3.1 対象とするマイクロサービス

本手法が対象とするマイクロサービスは，コンテナのオーケストレーションツールである Kubernetes[*1] 上に展開される一般的なマイクロサービスとする．本手法ではコンポーネント間の依存関係のモデル化にトレースログを用いるが，一般的なマイクロサービスに対応するためにコンポーネント内にトレースログ収集用のプロキシコンテナを配置することでマイクロサービスへの変更の追加無くトレースログを収集できるサービスメッシュ [11] を適用す

[*1] Kubernetes: https://kubernetes.io/

る．また，マイクロサービスから収集するメトリックには，コンポーネント内のコンテナの CPU 使用率やメモリ使用量，及びトレースログから取得可能なコンポーネントの応答時間などの一般的なメトリックを採用する．

3.2 全体の流れ

提案手法は 4 つのステップで構成される．図 1 に提案手法の概要を示す．まず, SLO（サービスレベル目標：Service Level Objective）に基づいて障害検知を行う（a）．次に，コンポーネント間の依存関係を DAG（Directed Acyclic Graph）でモデル化し（b），構築した DAG のエッジへ重み付けする（c）．最後に，重み付けした複数の DAG を用いて，各コンポーネントの障害への貢献度を表す異常度を算出し，降順で上位にあるものを障害原因箇所として推定する（d）．

(a) ステップ 1：障害検知

ステップ 1 では，定期的にメトリックを取得し，SLO に基づいた障害検知を行う．SLO とはシステムが提供するサービスの品質を定量的に定めた目標値であり，本稿ではシステムの稼働率やシステムから収集されるメトリックを用いた可用性を対象とする．例として「Web サイトへのアクセスに対して，1 ヶ月のリクエストのうち最低でも 99.9% の割合で 450ms 以内にレスポンスを返す」という SLO を定めた場合，サービスは常にその SLO を満たした品質で提供する必要があるため，図 1（a）のように「1 ヶ月のリクエストのうち最低でも 99.9%の割合で 450ms 以内にレスポンスを返せているかどうか」が障害検知の基準となる．また，この基準の場合だと監視に必要なメトリックは 1 ヶ月分の応答時間（レスポンスにかかった時間）となるため，定期的に取得した応答時間のメトリック 1 ヶ月分から，この基準を満たしているかどうかの監視を行う．

(b) ステップ 2：DAG の構築

障害検知後，コンポーネント間の依存関係を DAG でモデル化する，モデル化には，サービスメッシュにより収集した一定期間のトレースログを用いる．モデル化にトレースログを使用することで，コンポーネント間の依存関係を直接捉えたモデル化が可能である（3 章の考慮点 3）．また，提案手法では，ステップ 2 により構築した DAG のエッジに対して，障害検知する直前の 5 分間分のメトリックを用いた重み計算を行う．そのため，その更に前の 5 分間分のトレースログをシステムが正常に動作しているときのデータとして扱い，モデル化に使用する．DAG においては，図 1（b）の $c_i \to c_j$ のようにコンポーネント c_i からコンポーネント c_j へ向かう有効エッジがあるとき，コンポーネント c_i はコンポーネント c_j に依存することを意味する．構築した DAG はそのままだと最前段のコンポーネントに向かうエッジが無く，最前段のコンポーネントが障害原因箇所であった場合の特定ができないため，最前段へ有効エッジを伸ばすダミーノードを再前段より前のユーザに相当する位置に配置することで対応する．

(c) ステップ 3：エッジの重みを算出

DAG を構築後，障害検知する直前の 5 分間分のメトリックを用いて，図 1（c）のように DAG のエッジへ重み付けを行う．一般的に障害原因箇所では，障害発生時に他のコンポーネントよりも定常時と障害発生時でメトリックの振る舞いが大きく異なるため，重み付けでは各メトリックが定常時からどれだけ乖離しているかを表す乖離度を算出し，重みとして利用する（3 章の考慮点 2）．このとき，エッジ特性及びコンポーネント特性のメトリックを用いて重みを算出する（3 章の考慮点 1）ため，各メトリックの重みの分布（最大値と最小値）をあわせる正規化も行う．定常時を表すしきい値は，過去のメトリックの極値分布に基づいた SPOT [12] を用いて，事前に蓄積した 1 時間分のメトリックを学習させることで生成する．

(c-i)エッジ特性からエッジの重みを算出

コンポーネントの集合を $C = c_1, c_2, \cdots, c_{|C|}$（$|C|$：コンポーネント数), 全てのエッジの集合を E としたとき, $c_i \in C$ から $c_j \in C$ へ向かう有向エッジを $(c_i \to c_j) \in E$ と表す．エッジ特性のメトリック $M = m_1, m_2, \cdots, m_{|M|}$（$|M|$：エッジ特性のメトリック数）に対し，時刻 $t = 0, 1, \cdots, T$ のエッジ $(c_i \to c_j)$ の $m_n \in M$ の値を $v^{m_n}_{(c_i \to c_j)}(t)$，SPOT のしきい値を $s^{m_n}_{(c_i \to c_j)}(t)$ とおくと，メトリックの観測区間 $[0, T]$ におけるエッジ $(c_i \to c_j) \in E$ の乖離度 $D^{m_n}_{(c_i \to c_j) \in E}$ は式 (1) で計算される．

$$D^{m_n}_{(c_i \to c_j) \in E} = \max_{t \in [0,T]} \frac{v^{m_n}_{(c_i \to c_j)}(t) - s^{m_n}_{(c_i \to c_j)}(t)}{s^{m_n}_{(c_i \to c_j)}(t)} \tag{1}$$

また，乖離度が正であるエッジの集合 $e \in E$ において，$D^{m_n}_{(c_i \to c_j) \in e}$ の最大値を $D^{m_n}_{\max} = \max_{e \in E}(D^{m_n}_{(c_i \to c_j) \in e})$，最小値を $D^{m_n}_{\min} = \min_{e \in E}(D^{m_n}_{(c_i \to c_j) \in e})$，$m_n$ の DAG で c_j に向かうエッジの集合を $e(c^{m_n}_j)$ としたとき，c_j に向かうエッジの数は $|e(c^{m_n}_j)|$ で表され，$[0, 1]$ で正規化した乖離度 $\overline{D}^{m_n}_{(c_i \to c_j) \in e}$ は式 (2) で計算される．エッジ特性のメトリックを用いた重み付けでは，式 (2) により算出した値を重みとする．

$$\overline{D}^{m_n}_{(c_i \to c_j) \in e} = \begin{cases} 1 & (|e(c^{m_n}_j)| = 1) \\ \frac{D^{m_n}_{(c_i \to c_j)} - D^{m_n}_{\min}}{D^{m_n}_{\max} - D^{m_n}_{\min}} & (otherwise) \end{cases} \tag{2}$$

(c-ii)コンポーネント特性からエッジの重みを算出

コンポーネント特性のメトリックを $M' = m'_1, m'_2, \cdots, m'_{|M'|}$（$|M'|$：コンポーネント特性のメトリック数）としたときのコンポーネント $c_i \in C$ のメトリック $m'_n \in M'$ の値を $v^{m'_n}_{c_i}(t)$，SPOT のしきい値を $s^{m'_n}_{c_i}(t)$ とおくと，メトリックの観測区間 $[0, T]$ における m'_n，c_i の乖離度 $D^{m'_n}_{c_i}$ は式 (3) により算出される．

$$D^{m'_n}_{c_j \in C} = \max_{t \in [0,T]} \frac{v^{m'_n}_{c_j}(t) - s^{m'_n}_{c_j}(t)}{s^{m'_n}_{c_j}(t)} \tag{3}$$

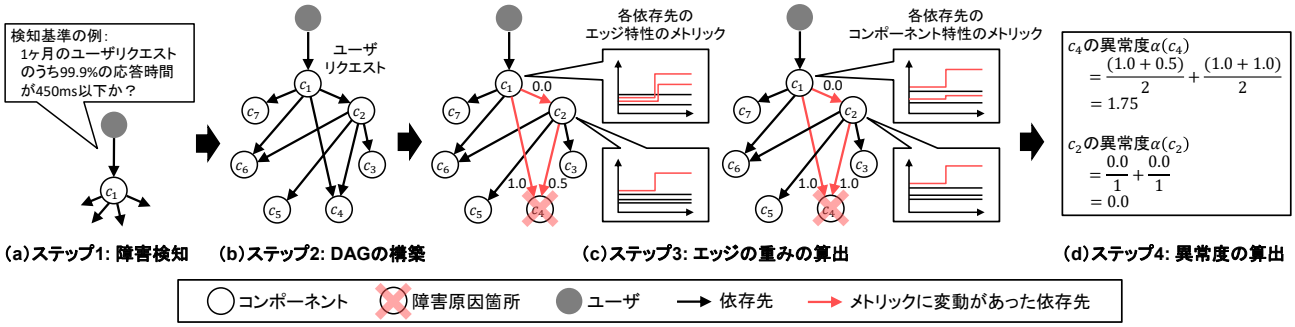

図 1 提案手法による障害原因箇所の特定の流れ

式 (3) は式 (1) の添字 m_n が m'_n に，また $(c_i \to c_j) \in E$ が $c_i \in C$ に置き換わっただけである．式 (3) で算出した乖離度が正の値であるコンポーネントの集合 $c \in C$ において，式 (2) と同様に $[0,1]$ で正規化した乖離度を $\overline{D}^{m'_n}_{c_i \in c}$ とし，式 (4) によりエッジ特性の重み $\overline{D}^{m'_n}_{(c_i \to c_j) \in e(c_j^{m'_n})}$ に変換する．

$$\overline{D}^{m'_n}_{(c_i \to c_j) \in e(c_j^{m'_n})} = \overline{D}^{m'_n}_{c_j \in C} \tag{4}$$

(d) ステップ 4：コンポーネントの異常度を算出

ステップ 3 で重み付けした DAG を用いて，図 1（d）のようにコンポーネント c_j が定常時と比べてどれだけ異常であるかを表す異常度 $\alpha(c_j)$ を式 (5) により算出する．

$$\alpha(c_j) = \sum_{n=1}^{|M|} \left\{ \frac{\sum_{(c_i \to c_j) \in e(c_j^{m_n})} \overline{D}^{m_n}_{(c_i \to c_j)}}{|e(c_j^{m_n})|} \right\} + \sum_{n=1}^{|M'|} \left\{ \frac{\sum_{(c_i \to c_j) \in e(c_j^{m'_n})} \overline{D}^{m'_n}_{(c_i \to c_j)}}{|e(c_j^{m'_n})|} \right\} \tag{5}$$

最終的に算出された $\alpha(c_j)$ を降順に並び替えたときの上位のコンポーネントを障害原因箇所として推定する．例えばあるコンポーネントの異常度が上位 1 番目である場合，そのコンポーネントが最も障害原因箇所として適当であることを意味する．

4. プロトタイプシステムの実装

上記の提案手法に基づく障害原因箇所特定システムを Python で開発した．図 2 に開発したシステムの概要を示す．コンテナのリソース使用量などのコンポーネント特性のメトリックの取得には cAdvisor[*2]，コンポーネントの応答時間などのエッジ特性のメトリックを含むトレースログの取得にはサービスメッシュを実現するツールの 1 つである Istio[*3]，取得したメトリックの収集・蓄積には Prometheus[*4]を使用した．cAdvisor はコンテナとして手動で実行する方法と，Kubernetes のコンポーネントの 1 つである kubelet に統合された機能として自動で実行する方法があるが，提案システムの実装においてはその導入の容易さから後者を採用した．Istio によるトレースログの収集には，コンポーネント内にサイドカーコンテナと呼ばれるトレースログ収集用のプロキシを配置することで実現する．

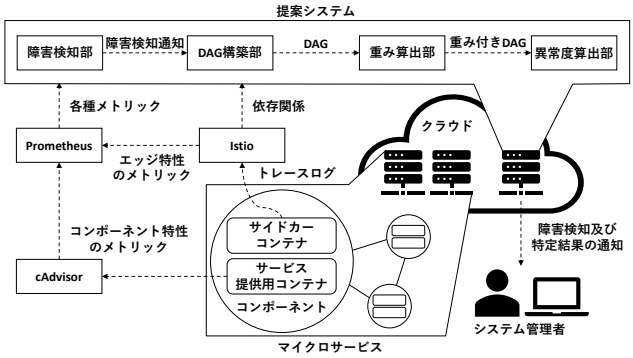

図 2 障害原因箇所特定システムの概要

システムは 4 つのモジュールで構成される．障害検知部では，定期的に Prometheus からメトリックを取得し，SLO に基づいた障害検知を行う（ステップ 1）．障害検知後，ステップ 2 として DAG 構築部が Istio からコンポーネント間の依存関係の情報を取得し，DAG の構築を行う（ステップ 2）．DAG 構築後，ステップ 2 で構築した DAG のエッジに対して，重み算出部による重み付けを行う（ステップ 3）．最後に，重み算出部で作成した複数の重み付き DAG を用いて，異常度算出部にてコンポーネントごとに異常度を算出し（ステップ 4），降順で並び替えたときの上位にあるコンポーネントを障害原因箇所として特定する．

5. 評価

提案手法の有効性を示すために，開発した障害原因箇所特定システムとマイクロサービスのベンチマークを用いた実験を行い，提案手法による障害原因箇所の特定精度と特定時間を評価する．

5.1 実験環境

実験環境に使用したサーバとツールの構成を表 1 に，ま

*2 cAdvisor: https://github.com/google/cadvisor/
*3 Istio: https://istio.io/
*4 Prometheus: https://prometheus.io/

表 1 実験環境のサーバとツールの構成

項目	仕様	
Master ノード・	OS	CentOS 7.8.2003
Worker ノード	CPU	Intel Xeon CPU 3.50GHz
（サービス提供用）	vCPU	2 core
	RAM	4 GiB
監視用サーバ・	OS	CentOS 7.9.2009
解析用サーバ	CPU	Intel Xeon CPU 2.20GHz
	vCPU	4 core
	RAM	24 GiB
Kubernetes	Version	1.21.3
Prometheus	Version	2.21.0
Istio	Version	1.7.4

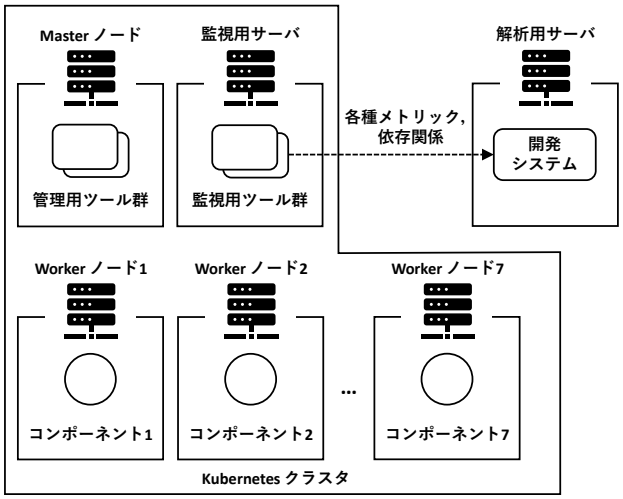

図 3 実験環境の全体構成

た実験環境の全体構成を図 3 に示す．1 台の Master ノード・7 台の Worker ノードと 1 台の監視用サーバからなる Kubernetes クラスタを構築し，Worker ノード上に EC サイトを模したマイクロサービスのベンチマークである Sock Shop[*5]を展開した．Sock Shop は 7 個のコンポーネントと 9 本のエッジで構成され，コンポーネント間で CPU やメモリ等のリソース共有が起こらないように各 Worker ノードに対して 1 コンポーネントとなるよう展開した．Prometheus と Istio は監視用サーバ，開発した障害原因箇所特定システムは解析用サーバで動作させた．ネットワーク構成としては，全てのサーバを同一ネットワーク上に配置した．

5.2 特定精度の評価

提案手法による障害原因箇所の特定精度を評価するために，予め障害検知基準を設定し，Sock Shop に対してユーザからのサービス利用を模したリクエストを Locust[*6]により生成し始めた 1 時間後に，任意のコンポーネント内のサービス提供用コンテナに対して Stress-ng[*7]による CPU 高負荷と，Chaos Mesh[*8]による 300ms の通信遅延を異なる時間軸で注入し障害を再現した．障害検知基準には「過去 5 分間のうち CPU 使用率が 140%を超えた期間が 4 分を超過したとき」または「直近 1 時間のリクエストの 99.9%の応答時間が 450ms を上回ったとき」を設定した．各コンポーネント及び注入する負荷の試行回数は 3 回であり，コンポーネント数 7 個×負荷 2 種類×試行回数 3 回の計 42 回実験を行った．特定に使用するメトリックは，表 2 に示す 7 種類を採用した．ここで，応答時間はエッジ特性，その他の 6 種類はコンポーネント特性のメトリックである．測定指標には，障害原因箇所として推定した上位 k 番目以内に障害原因箇所が含まれる確率 Top-k を採用した．

表 2 特定に使用したメトリック

メトリック	説明
istio_request_duration_milliseconds_sum / istio_request_duration_milliseconds_count	応答時間
container_cpu_usage_seconds_total	CPU 使用率
container_memory_working_set_bytes	メモリ使用量
container_network_transmit_bytes_total	送信バイト数
container_network_receive_bytes_total	受信バイト数
container_fs_reads_bytes_total	ファイルシステムへの読み込み数
container_fs_writes_bytes_total	ファイルシステムへの書き込み数

実験の結果，いずれのパターンでも負荷を注入して 5 分以内に障害検知し，実験 42 回分の特定精度 Top-1 は 81.0%，Top-2 は 100.0%であった．本研究では，障害による機会損失を可能な限り少なくするには，障害原因箇所として推定した最初のコンポーネントが障害原因箇所であること，すなわち Top-1 が 100.0%であることを目標として考えている．対して提案手法による Top-1 は 81.0%であるため，提案手法は目標に近い精度で特定できているといえる．また Top-2 が 100.0%であることから，最低でも障害原因箇所として推定したコンポーネントの上位 2 番目以内であれば必ず正しい特定が行えるため，十分精度の高い特定が行えている．

実験により任意のコンポーネントの障害が他のコンポーネントに伝搬した一例として，Sock Shop のコンポーネントの 1 つである shipping に通信遅延を注入した場合の応答時間の DAG を図 4 に示す．shipping に通信遅延を注入したことで shipping の応答時間（図 4 の orders から shipping へ向かうエッジのメトリック）が増大し，その影響により shipping に依存する orders の応答時間（図 4 の front-end から orders へ向かうエッジのメトリック）と orders に依存する front-end の応答時間（図 4 のユーザから front-end へ向かうエッジのメトリック）も増大した．定常時の shipping の応答時間は数 ms で安定するのに対

*5 Sock Shop: https://microservices-demo.github.io/
*6 Locust: https://locust.io/
*7 Stress-ng: https://github.com/ColinIanKing/stress-ng
*8 Chaos Mesh: https://github.com/chaos-mesh/chaos-mesh

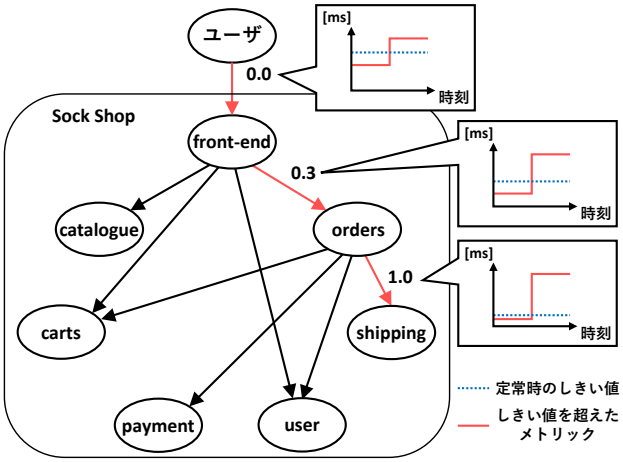

図 4 Sock Shop の shipping に通信遅延を注入した場合の応答時間の DAG

して，orders の応答時間は約 30ms，front-end の応答時間は 100ms 前後であり，また orders と front-end は自身への API 呼び出しの処理に加えて自身が依存するコンポーネントへの API 呼び出しも行うことから応答時間が数十 ms 単位で変動する．そのため，SPOT により生成されるしきい値はこれら 3 つの中で shipping が最も小さくなり，shipping の応答時間の乖離度が最も大きくなったことで，そのメトリックの収集元である shipping に向かうエッジの重みが DAG の中で最大となった．このように，メトリックの定常時からの変化量（乖離度）とコンポーネント間の依存関係の DAG を組み合わせることで，どのエッジのメトリックがどのコンポーネントに影響されたかを加味した重み付けが可能となるため，障害原因箇所の特定に依存関係を利用することは有効であるといえる．

CPU 高負荷と通信遅延それぞれの実験結果を分析したところ，CPU 高負荷による障害に対しては Top-1 が 100.0%（21/21 の精度）であったが，通信遅延による障害に対しては Top-1 が 66.7%（14/21 の精度）であった．通信遅延に対する特定精度が低下した原因は，今回使用した 7 つのメトリックのうち 6 つが通信遅延の障害に対する特定に貢献できるメトリックではなく，重み計算に利用した 5 分間のメトリックのうち定常時のしきい値から偶発的に外れた値をもったものが存在したためである．これは，元々提案手法でメトリックの定常時のしきい値生成に使用している SPOT では全てのメトリックの定常時を網羅したしきい値生成が難しく，障害が発生していなくともしきい値から外れた値をもつメトリックが日常的に存在することに起因する．特定精度を改善するには全てのメトリックの定常時を正確に網羅したしきい値生成を行う必要があるが，この点については今後の課題とする．今回使用したコンポーネント特性のメトリックへの影響が大きい CPU 高負荷に対しては，CPU 高負荷が発生したコンポーネントのメトリックの多くが定常時から乖離した値を持つために，そのコンポーネントの異常度は他のものより大きくなる．一方でコンポーネント特性のメトリックへの影響が小さい通信遅延に対しては，障害原因箇所とは異なるコンポーネントのメトリックでも定常時のしきい値から偶発的に外れたものが多く存在するようになり，障害原因箇所以外の異常度も大きくなる．このことから，本手法による障害原因箇所の特定では，どのメトリックを特定に使用するかどうかが特定精度に大きく影響を与えると考えられる．

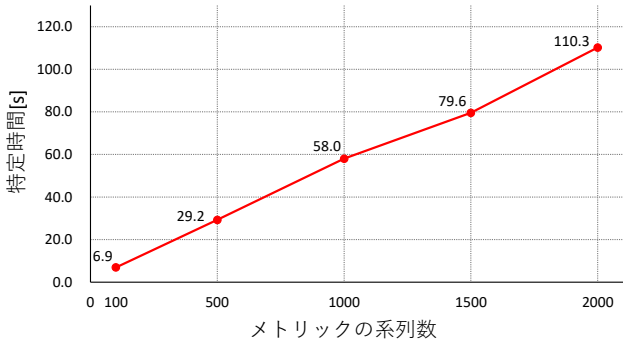

図 5 メトリックの系列数に対する特定時間

5.3 特定時間の評価

ステップ 1 の障害検知後に行う 3 つのステップのうち，ステップ 2（DAG 構築）とステップ 4（異常度計算）の所要時間に関する実験には大規模なベンチマークも必要となる．特にステップ 3 の重み計算は，コンポーネント数及びコンテナ数が増加して大規模になるほど重み計算に使用するメトリックの系列数は増大することで所要時間が大きくなる可能性があり，これが特定までの時間に大きく影響する．そこで，5.2 節で使用したメトリックを複製し擬似的にメトリックの系列数を増加させることで特定時間の変化を調査した．実験では，ステップ 2 とステップ 4 の所要時間を計 2.05 秒（5.2 節で使用したベンチマークでの 5 回の測定値の合計）固定とし，メトリックの系列数は 100，500，1000，1500，2000 とした．

図 5 は 5 回試行した実験結果であり，特定時間はメトリックの系列数に応じて線形的に増加した．また提案手法は，メトリックの系列数が 2000 個までであれば 2 分以内に特定できることが分かった．実験ではエッジ特性のメトリック 1 種類とコンポーネント特性のメトリック 6 種類を特定に使用したため，メトリックの系列数 2000 個はコンテナ約 300 個のシステムに相当する．システムの迅速な復旧においては秒単位で特定を終える必要は無く，また Google Cloud Plaform がサービス利用ユーザとの間で定めたサービス品質の保証契約 [13] によると，5 分以内であれば障害とみなしていないため，本研究では 5 分以内に障害原因箇所の特定を終えることを目標として考えている．そのため，コンテナ数約 300 個までであれば 2 分以内に障害原因

箇所を特定できる提案手法は，一定の規模に対して目標を満たした十分高速な特定が行えているといえる．

この実験では 5.2 節で使用した 7 種類のメトリックを複製したが，5.2 節で述べた考察を踏まえると，特定精度の都合で 7 種類より多くのメトリックを使用する場合は，メトリックの系列数 2000 個に対する特定時間は 2 分より大きくなると予想する．マイクロサービスから収集できるメトリックは数多くあるが，ネットワークの送信バイト数と送信リクエスト数のように非常に類似した傾向を示すメトリックも存在する．そのため，提案手法による迅速かつ正確な特定を実現するには，このような冗長なメトリックを除外しつつ，システムの健全性を監視するのに重要なメトリックを選択することが重要であると考える．

6. おわりに

マイクロサービスにおける障害原因箇所の迅速な特定を行うために，メトリック値の定常時からの変化量にコンポーネント間の依存関係を組み合わせた特定手法を提案した．提案手法に基づいた障害原因箇所特定システムを開発し，EC サイトを模したマイクロサービスで障害を再現する実験を行った結果，全コンポーネントのうち障害原因箇所を 81.0%で上位 1 番目に，100.0%の精度で上位 2 番目に特定でき，またコンテナ数が約 300 個のマイクロサービスであれば 2 分以内に特定できることを示した．このことから，提案手法は上位 1 番目に 100.0 の精度で，かつ一定の規模であれば 5 分以内に障害原因箇所を特定するという目標を概ね達成することを確認した．

今後は提案手法の特定精度の改善方法を検討する．また，既存の障害原因箇所特定システムとの比較などにより，提案手法の更なる有効性を検証していく．

謝辞 本研究の一部は JSPS 科研費 21K11846，19K11929，21H03432，及び 18K11271 の支援を受けて実施しました．

参考文献

[1] “Microservices a definition of this new architectural term”, `https://martinfowler.com/articles/microservices.html` (accessed Jan.25, 2022)
[2] Mayer, B., and Weinreich, R., “A dashboard for microservice monitoring and management,” 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp.66–69, 2017.
[3] Zhou, X., Peng, X., Xie, T., et al., “Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs,” 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp.683–694, 2019.
[4] Lin, J., Chen, P. and Zheng, Z., “Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-Service Environments,” International Conference on ServiceOriented Computing (ICSOC), pp.3–20, 2018.
[5] Wang, P., Xu, J., Ma, M., Lin, W., Pan, D., Wang, Y. and Chen, P., “CloudRanger: Root Cause Identification for Cloud Native Systems,” IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp.492–502, 2018.
[6] Wu, L., Tordsson, J., Elmroth, E., and Kao, O., “MicroRCA: Root Cause Localization of Performance Issues in Microservices,” NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium, pp.1–9, 2020.
[7] Ma, M., Lin, W., Pan, D., and Wang, P., “Self-Adaptive Root Cause Diagnosis for Large-Scale Microservice Architecture,” IEEE Transactions on Services Computing, 2020.
[8] Ma, M., Xu, J., Wang, Y., Chen, P., “Zhang, Z. and Wang, P., “AutoMAP: Diagnose Your Microservice-based Web Applications Automatically,” The Web Conference, pp.246–258, 2020.
[9] Molnar, C., “Interpretable machine learning. A Guide for Making Black Box Models Explainable,” `https://christophm.github.io/interpretable-ml-book/` (accessed Jan.25, 2022)
[10] 鶴田博文, 坪内祐樹, “分散システムの性能異常に対する機械学習の解釈性に基づく原因診断手法,” インターネットと運用技術シンポジウム論文集, pp.24–31, 2021.
[11] Li, W., et al., “Service mesh: Challenges, state of the art, and future research opportunities,” 2019 IEEE International Conference on Service-Oriented System Engineering(SOSE), pp.122–1225, 2019.
[12] Siffer, A., Fouque, P.-A., Termier, A. and Largouet, C., “Anomaly detection in streams with extreme value theory,” 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.1067–1075, 2017.
[13] “Google Cloud Platform Service Level Agreements”, `https://cloud.google.com/terms/sla` (accessed Jan.25, 2022)