Meltria：マイクロサービスにおける異常検知・原因分析のためのデータセットの動的生成システム

坪内 佑樹[†1,†2,a),b)]　青山 真也[†1,c)]

概要：クラウド上の大規模なアプリケーションの構成は，機能単位で独立して変更可能とするために，単一の巨大なアプリケーションを分解して分散協調させるマイクロサービスアーキテクチャへと変遷している．アプリケーション構成の分散化により，構成要素数が増大し，構成要素間の依存関係が複雑化することから，システム管理者の認知負荷が高まっている．認知負荷を低減するために，システム管理者の経験と直感が要求される異常検知と異常の原因分析を自動化するための研究が盛んである．これらの研究では，データ分析手法を実験により評価する際に，正常データと異常データを含む運用データが必要となる．既存の公開されているデータセットは，その静的な性質故に，データセットに含まれる異常パターンの数は限られる．本研究では，多様な異常のパターンに対して異常検知・原因分析手法を評価するために，データセットを動的に生成するためのシステムである Meltria の設計基準を提案する．我々が提案する設計基準は，(1) 運用データに異常を含めるために，多様な故障注入を実行し，データを採取するための一連の手続きを実行可能なスケジューリング，および，(2) 故障注入の影響と想定外の異常のそれぞれの有無をデータセットにラベル付けするための検証の自動化である．Meltria を用いて，故障注入の種類やパラメータを変更することにより，様々な異常のパターンを含んだデータセットを生成できる．実験の結果，生成されたデータセットに対する (2) の基準に基づいた検証手法の正解率は 85%となった．

Meltria: A Dynamic Datasets Generating System for Anomaly Detection and Root Cause Analysis in Microservices

Yuuki Tsubouchi[†1,†2,a),b)]　Masaya Aoyama[†1,c)]

Abstract: Large-scale applications in the cloud are migrating to Microservices architecture to change application features independently. Microservices architecture brings high cognitive load on system administrators because it increases the number of application components and the complexity of dependencies between components. To reduce cognitive load, there are many pieces of researches for automated anomaly detection and cause analysis, which require the experience and intuition of system administrators. These researches require operational data including normal and abnormal data to evaluate their methods by experiments. Because existing public datasets are static, the number of anomaly patterns included in the datasets is limited. In this paper, we propose new design criteria for a dynamic datasets generating system named Meltria to evaluate anomaly detection and cause analysis methods for a wide variety of anomaly patterns. The criteria we propose include (1) feasible scheduling of a scenario to perform various fault injections and picking data to include anomalies in a dataset, and (2) the automation of verification to label the dataset with the presence or absence of fault injection effects and unexpected anomalies. Meltria can generate datasets containing various patterns of anomalies by changing the type and parameters of fault injection. Experimental results indicate that the accuracy of the veritication method based on the criteria (2) is 85%.

[†1] さくらインターネット株式会社 さくらインターネット研究所 SAKURA internet Research Center, SAKURA internet Inc., Ofukatyo, Kitaku, Osaka 530-0011 Japan
[†2] 京都大学情報学研究科, Graduate School of Infomatics, Kyoto University, Kyoto 606–8501, Japan
[a)] y-tsubouchi@sakura.ad.jp
[b)] y-tsubouchi@net.ist.i.kyoto-u.ac.jp
[c)] amsy810@gmail.com

1. はじめに

SNS（Social Networking Service），メディア配信，電子商取引，および，IoT (Internet of Things) などを実現する今日の IT サービスのアプリケーションは，クラウドの分散コンピューティング環境に展開されている．クラウド上で大規模なアプリケーションを構成する際に，アプリケーション開発者が機能単位で独立して変更するために，単一の巨大なアプリケーションを独立した小さなサービスとして分解して分散協調させるマイクロサービスアーキテクチャ [20] が採用されている．マイクロサービス化により，アプリケーションの内部システムの構成要素数が増加し，構成要素間の相互作用の関係性も複雑化するため，システム管理者の認知負荷が増大する．そのため，アプリケーションの異常を検知したり，異常の原因を分析するといった，システム管理者の経験や直感が要求されるような仕事がより困難となっている．そこで，認知負荷を低減するために，統計学や機械学習によるデータ分析により，異常検知・原因分析を自動化するための研究が活発である [19]．

異常検知・原因分析の研究では，提案されるデータ分析手法の有用性を評価するために，アプリケーションの運用中に計測される運用データセットが必要となる．異常検知・原因分析は，正常なデータと異常なデータの両方を含んだデータに対して，異常の有無，または，異常の原因を予測するタスクである．したがって，データ分析手法の評価用のデータセットには，正常なデータだけでなく，異常なデータを含めなければならない．既存研究では，評価用のデータセットは独自に作成されており，それらのデータセットは研究コミュニティに対して非公開であることがほとんどである．また，データセットを作成するための手順の詳細は記載されていないことが多い．

公開されたデータセットが不足しているという課題に対して，コミュニティから利用可能なデータセットを公開する取り組みがある [11, 13]．多様な異常のパターンを含むデータセットが公開されていれば，既存の異常検知・原因診断手法について，既存の評価実験とは異なる異常のパターンに対して，性能を再評価することが容易となる．もし既存手法の性能が低下するような異常のパターンを発見すれば，そのパターンに対応するために，新たな手法を考案する契機となりえる．しかし，作成済みの静的なデータセットに含まれる異常のパターンには限りがある．新たな異常パターンをデータセットに含めるのであれば，その異常パターンを再現させた上で，データセットを再作成する必要がある．データセット作成者にとって未知の異常パターンが存在することやデータセットが肥大化することを考慮すると，全ての異常パターンを含むデータセットを事前に作成することは難しい．

本研究では，異常検知・原因分析の評価用に，多様な異常のパターンを含むデータセットを柔軟に生成できるように，正常・異常データを含むデータセットを動的に生成するためのシステムである Meltria を提案する．まず，異常検知・原因分析の既存研究の文献から，評価実験に必要なデータセットの要件を導出する．次に，導出されたデータセットの要件を満たすことが可能な，データセット生成システムの設計基準を提示する．第一の設計基準は，Meltria の基本機能であり，運用データに異常を含めるために故障注入のスケジューリングを自動化することと，故障内容とデータを紐付けられるようにデータを管理することである．第一の設計基準にしたがってデータセットを動的に生成すると，管理するデータセットの個数が増加しやすくなることから，それらのデータセットにデータ分析者の助けとなるラベル情報を手動で付与することが困難となる．第二の設計基準は，故障注入の影響の有無と想定外の異常の有無を生成されたデータセットにラベル付けするために，データセットの検証を自動化することである．これらの設計基準に従い，コンテナ型仮想化の管理基盤である Kubernetes [12] 上に構築したマイクロサービスのデモアプリケーションである Sock Shop [6] を用いて，本システムを実装した．第一の設計基準に従い実際に生成されたデータセットに対して，第二の設計基準に従った検証手法を適用する実験を行い，検証手法の性能と，データセットの妥当性を評価した．

本論文を以下のように構成する．2 章で，既存の異常検知・原因分析の研究における評価実験を調査し，公開データセットと意図的に異常を再現するシステムの関連研究を述べる．3 章では，既存研究の調査内容から，Meltria の設計基準を導出した上で，具体的な設計とその実装方法を提示する．4 章では，実際にデータセットを生成する実験を行い，生成されたデータセットの検証手法を評価する．最後に，5 章にて，本研究の結論と今後の展望を述べる．

2. 関連研究

2.1 異常検知・原因分析のデータ生成手法

Soldani らの調査 [19] に挙げられているマイクロサービスにおける異常検知・原因分析の研究にて実施されたデータセットの生成手法を表 1 に示す．これらの研究で収集される運用データは，時系列の数値データ（メトリック），システムイベントに関する高度に構造化されたデータ（イベント），構造化されていない文字列のログ（ログ），リクエストの実行経路のグラフ（トレース）の 4 種類 [14] のうちのいずれか，あるいは複数の組み合わせとなる．イベントとログを総称してログと呼ぶこともあるため，以降ではまとめてログとする．表 1 の列のうち，タスクは異常検知（AD：Anomaly Detection）または原因分析（RCA：Root Cause Analysys），テストベッドは Sock Shop [6]（SS），Pymicro [5]（P），TrainTicket [2]（TT），データ種別はメ

表 1: マイクロサービスを対象とした異常検知・原因分析研究のデータ生成手法

文献	タスク	テストベッド	データ種別	注入される故障	注入回数	注入時間	負荷生成	データの規模や粒度
Microscope [15]	RCA	SS	M	CPU 過負荷, ネットワーク混雑, コンテナ停止	5 (240)	1 分	5000 QPS	1 分サンプリング
AutoMAP [17]	RCA	P	M	コンテナ停止,DoS 攻撃	NA	NA	NA	注入の前後 1 時間，2 秒サンプリング
CloudRanger [22]	RCA	P	M	NA	NA	NA	NA	2 時間
MicroRCA [24]	RCA	SS	M	ネットワーク遅延,CPU 過負荷, メモリリーク	5 (95)	1 分	500users, 600QPS	5 秒サンプリング
Aggarwal et al. [7]	RCA	TT	L	サービス間の HTTP 要求エラー	NA	NA	NA	注入あたり平均 164,740 行
MEPFL [26]	RCA	SS, TT	DT	NA	SS:32, TT:142	NA	NA	NA
Qiu et al. [18]	RCA	SS	M	CPU 過負荷, メモリリーク, ディスク I/O ブロック, ネットワーク混雑	20	NA	NA	NA
Wang et al. [21]	RCA	TT	L,M	計算機リソース枯渇,HTTP 要求遅延・中断, その他 16 ケースの故障	NA	NA	NA	NA
Arya et al. [8]	RCA	TT	L	NA	NA	NA	NA	NA
MicroRAS [23]	AD	SS	M	CPU 過負荷, メモリリーク	3-5 (23)	3 分	500 users, 600QPS	時間範囲 5 分
TraceAnomaly [16]	AD	TT	DT	ネットワーク遅延, コンテナ削除[*1]	1	5 分	NA	1 日分[*2]

トリック（M: Metrics），ログ（L: Logs），トレース（DT: Distributed Traces）のいずれかを示し，NA（No Answer）は文献に記載がないことを表す．

表 1 の調査により，各研究で共通してみられるデータセット作成の手順は，(1) マイクロサービスのデモアプリケーションを構築し，(2) そのアプリケーションに対して，実験者が意図的に故障を注入し，(3) 故障が注入された前後の期間の運用データを取得する，というものであることが分かった．このように作成されたデータセットを使用して，異常検知または原因分析のタスクが評価される．異常検知では，故障が注入されていない正常期間のデータを学習しておき，学習済みのモデルに対して，正常期間または異常期間のデータを入力として与えて，異常の有無を予測する．原因分析では，故障注入ごとの前後の期間の運用データを分析手法の入力として与えて，故障注入を正解として，故障が注入された箇所や故障に対応するメトリック，ログ，トレースを予測する．

表 1 より，商用環境以外のマイクロサービス構成のアプリケーションとして，Sock Shop，TrainTicket，および，Pymicro が使用されている．Sock Shop は，8 個のマイクロサービスにより構成される小規模のアプリケーションである．TrainTicket は，41 個のマイクロサービスにより構成される中規模のアプリケーションである．Pymicro は，16 個のマイクロサービスによる構成される，実際のアプリケーション処理を含まないシミュレータである．

表 1 より，注入される故障は，(a)CPU 過負荷，メモリリーク，パケット遅延・パケット損失，コンテナの停止などの計算機リソースに関する故障，(b) サービス間の HTTP 通信の成否と遅延，および，(c)Zhou らの調査 [25] によるマイクロサービス特有の故障，の 3 つに分類される．まず (a) の故障は，CPU 過負荷やメモリリークは，仮想マシンやコンテナ上で stress-ng[*3]を実行することにより擬似的に再現し，パケット遅延と損失は Linux のトラヒック制御を行う tc[*4]を使用することにより実現されている．次に，マイクロサービス間に配置された HTTP プロキシが，HTTP 通信を中断させるか，または遅延を追加することにより，(b) の故障が実現される．最後に (c) の故障は，事前に用意された，故障の原因を含むアプリケーションコードや設定ファイルをアプリケーションに反映することにより，再現される．

表 1 に示す 11 件の文献のうち，アプリケーションに対して，どのように負荷を生成し，どのように故障を注入するかについて，一通りの手順が記載された文献は，Microscope [15]，MicroRCA [24]，および，MicroRAS [23] の 3 件のみであった．またこれらの 3 件の文献においても，ネットワーク遅延を注入するときの遅延時間などの故障注入時のパラメータは記載されていない．

2.2 公開データセット

異常検知・原因分析用の公開されたデータセットには，Loghub [11] と Exathlon [13] の 2 つがある．Loghub [11] は，OS やミドルウェアなどの 16 種類のシステムソフトウェアが出力したログを含むデータセットである．Exathlon [13] は，大規模データ向け分散処理ミドルウェアから収集された高次元時系列データを対象とした異常検知のためのデータセットである．公開データセットを使用すれば，第三者が自身でデータセットを作成する必要がないため，第三者がそのデータセットを使用した既存の実験結果を容易に再現できる．これらのデータセットは，誰でも利用できるように公開されているため，研究コミュニティ内で提案されてきた各種手法を比較評価するための標準となりえる．いずれのデータセットも，月単位の長期間にわたって収集されたデータを含むため，第三者が同等の期間のデータセットを作成するには，同等の時間を要する．

アプリケーションに発生する異常は，表 1 の列のうち，故障の種類，故障の注入時間，および，負荷生成に応じて，異なるパターンを示す．故障の種類，故障の注入時間，および，負荷生成は，データセットの作成前に設定する項目とパラメータである．したがって，静的なデータセットに含まれる異常のパターンの数は限られ，データセットを再

[*3] stress-ng: `https://kernel.ubuntu.com/~cking/stress-ng/`
[*4] tc: `https://linux.die.net/man/8/tc`

作成しないかぎりは，追加することもできない．これらの項目とパラメータの組み合わせごとにデータセットを事前に作成することも可能だが，組み合わせ数が大きいほど，データセットの生成には時間を要する．

2.3 Chaos Engineering のツール

表 1 の Qiu らの研究 [18] の実験では，異常を含むデータを採取するために，故意に故障を注入することが可能な Chaos Engineering のツール [3, 4] が使用されている．これらのツールの目的は，分散システムが予期せぬ事態に耐えられるかどうかの確証を得るための検証の規律である Chaos Engineering [10] を実践することである．これらのツールには，予期せぬ事態を故意に発生させるための故障注入の機能と，故障注入のスケジューリング実行機能が含まれる．注入可能な故障として，CPU・メモリ・ディスクなどの計算機リソース消費，パケット消失・パケット遅延増加・ネットワーク帯域制限，および，コンテナの停止などが挙げられる．ツールの利用者は，故障注入対象の構成要素と，故障の種類，故障注入間の時間間隔などの一連の手続きをワークフローとしてスケジューラに設定できる．しかし，ワークフローの中に，故障注入以外の，運用データの採取や分析のための処理を含めることが難しいという制約がある．

3. データセットの動的生成システム

データセットを用意する手間，既存文献の実験手順の不足，および，公開データセットに含まれる異常パターンの限定性を考慮すると，多様な異常パターンを含むデータセットを動的に生成するためのシステムが新たに必要である．

3.1 データセット生成システムの設計

2.1 節の調査より，各既存研究の評価実験にて，メトリックのみを収集し，かつ，(a) の計算機リソースに関する故障注入を使用した研究が最多であることから，以降では，メトリックのみを収集するものとし，かつ計算機リソースに関する故障注入のみを対象とする．そのときのデータセットの要件を以下に示す．

(1) アプリケーション内の複数の構成要素のそれぞれに故障注入されたデータを含む．
(2) アプリケーション内の各構成要素に対して，複数の異なる種類の故障が注入されたデータを含む．
(3) データ分析手法の予測結果の統計的な確からしさを高めるために，同一の構成要素に同じ種類の故障が複数回注入されたデータを含む．
(4) 複数回の故障注入の影響を分離して評価するために，同一の時間帯に，複数の異なる故障注入の影響が混在していない．
(5) 1 回の故障注入とそれに対応する正常なデータと異常なデータが両方含まれるデータを基本単位（以降，スロット）として管理されている
(6) 故障注入内容からそれに対応するスロットを発見できるように，データセット内の各スロットが管理されている

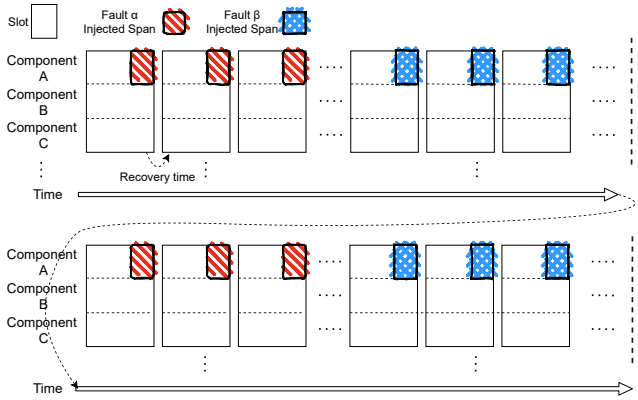


図 1: データセット生成のスケジューリング

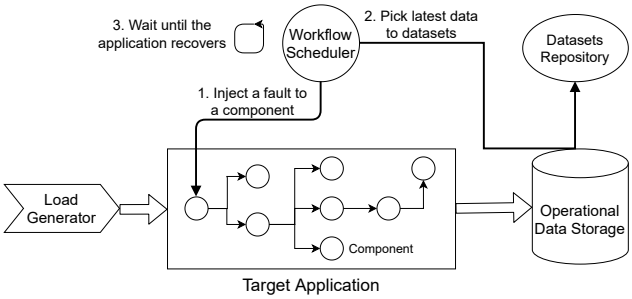


図 2: Meltria のアーキテクチャ

第一の設計基準は，データセットの要件を満たすように，故障注入をスケジュールすること，および，故障注入とスロットが紐付け可能であることである．図 1 に要件（1）〜（5）を満たすような故障注入のスケジューリングの様子を示す．図 1 は，複数のコンポーネントにそれぞれ 2 種類の故障を繰り返し注入する様子を示している．図 1 の斜線部分は故障注入期間を示している．注入される故障の種類と構成要素の組み合わせを事前に設定可能であり，同一の故障注入を複数回スケジューリング可能であることが求められる．要件（5）より，スロット（Slot）単位で正常と異常を含むデータを生成するために，故障注入後にシステムが回復するまで，指定された期間だけ待機する．要件（6）より，スロットと故障注入内容を紐付けるために，故障注入の内容をスロットのメタデータとして記録する．

第一の設計基準に基づいた，データセットの動的生成システムのアーキテクチャを図 2 に示す．負荷生成器（Load Generator）が対象アプリケーション（Target Application）に一定の負荷を与え続ける．負荷を与えられた対象アプリケーションから収集される運用データを運用データストレージ（Operational Data Storage）に継続的に保存する．第一の設計基準より，事前に与えられた故障注入対象と注

入する故障の種類の設定に基づいて，ワークフロースケジューラ（Workflow Scheduler）が必要な対象に必要な回数の故障注入を実行する．1 回の故障注入は，(1) 対象の構成要素に特定の故障を，5 分など事前に設定された時間だけ注入し続け，(2) 事前に設定された 1 時間などの時間範囲の運用データを採取して，採取されたデータをデータセットリポジトリ（Datasets Repository）に追加し，(3) アプリケーションの回復を待つために一定時間待機する，までの 3 ステップから成る．故障注入とスロットの紐付けのために，採取されたスロットを追加する際に，データセットリポジトリの各レコードのメタデータに，注入された故障の種類と故障注入された構成要素名を付与する．

第二の設計基準は，故障注入の影響の有無と想定外の異常の有無を効率よく調べるために，各スロットを自動で検証することである．データセットを検証する手順は，スロットごとに検査対象となるメトリックの系列を選択し，各系列を故障注入の影響の有無と想定外の異常の有無を表す状態のいずれかに分類する，というものである．

検査対象を限定するためのメトリックの系列の選択には，故障（Fault）からエラー（Error）が発生し，障害（Failure）へと至るサイクルを示す影響伝搬モデルである Fault-Error-Failure サイクル [9] を使用する．Fault-Error-Failure サイクルは，システム内部に故障が発生すると，故障がエラーを引き起こし，エラーが伝搬した結果，システム内部と外部の境界に影響が現れると，システム利用者へ影響する障害へと至る機構を示している．このサイクルに基づくと，データセット内の各スロットデータに対して，(a) 故障注入箇所に直接対応するメトリック（以降，故障メトリック），(b) 故障注入箇所を含むマイクロサービスの定常状態を示すメトリック（以降，マイクロサービスレベルメトリック），(c) 対象アプリケーション全体の定常状態を示すメトリック（以降，アプリケーションレベルメトリック）の 3 種類を検査対象の系列として選択すれば十分である．(a) は故障の種類に応じて，(b) と (c) は対象アプリケーションに応じて，それぞれ事前に定義されるものとする．

各系列の状態を分類するために，故障注入の影響の有無と想定外の異常の有無を表現可能な，次の 3 つの系列の状態を定義する．`NOT_FOUND` は入力系列に顕著な異常がないこと，`FOUND_INSIDE_ANOMALY` は入力系列内の故障注入期間に相当する時間範囲のみに異常があること，FOUNT_OUTSIDE_ANOMALY は，故障注入期間外に異常があることを表す．故障注入の結果，対象アプリケーション全体に期待どおりに顕著な影響が顕在化した状態は，(i) 単一のメトリックの系列内に正常標本と異常標本の両者が含まれている，(ii) 単一の故障から成る異常標本のみが含まれている，(iii) 異常標本が故障注入期間にあること，のすべてを満たす状態である．`FOUND_INSIDE_ANOMALY` は (i) から (iii) を同時に満たす状態と同等である．

Algorithm 1 系列の状態分類アルゴリズム

Input: A series $\boldsymbol{y} = \{y_t\}_{t=1}^{T}$ including an expected anomaly sub series $\{y_t\}_{t=a}^{T} (1 < a \leq T)$.
Output: One of `NOT_FOUND`, `FOUND_INSIDE_ANOMALY`, `FOUND_OUTSIDE_ANOMALY`

$upper\_val \leftarrow mean(\boldsymbol{y}) + 2 * std(\boldsymbol{y})$
$lower\_val \leftarrow mean(\boldsymbol{y}) - 2 * std(\boldsymbol{y})$
for $t \leftarrow 1; t \leq T; t = t + 1$ **do**
  if $y_t < lower\_val$ **or** $y_t > upper\_val$ **then**
    if $t < a$ **then**
      return `FOUND_OUTSIDE_ANOMALY`
    else
      return `FOUND_INSIDE_ANOMALY`
    end if
  end if
end for
return `NOT_FOUND`

選択された個々の系列に対して，故障注入の影響を分類するための手法をアルゴリズム 1 に示す．アルゴリズム 1 は，入力として標本数 T の系列 $\boldsymbol{y} = \{y_t\}_{t=1}^{T}$ を与えると，その系列が次の 3 つの状態のいずれかであるかの分類結果を出力する．mean 関数は算術平均値，std 関数は標準偏差を返すものとする．`NOT_FOUND` とそれ以外の状態の分類には，正規分布において 68%, 95%, 99.7%の値がそれぞれ平均の 1,2,3 標準偏差以内に収まるという経験則である 68-95-99 則 [1] を使用する．特に，99.7%の値が 3 標準偏差以内に収まる規則は 3 シグマ則とも呼ばれ，表 1 の Microscope [15] がアプリケーションの障害を検知するために使用する．(a) の故障に対応するメトリックでは，すでにリソース消費量が上限に近い状態でリソースを上限まで消費する種の故障を注入する場合，時系列の変動が小さくなる可能性がある．そのため，本研究では，3 シグマ則よりも異常検出率が高い，95%の値が 2 標準偏差以内に収まる 2 シグマ則を用いる．

3.2 データセット生成システムの実装

2.1 節にて述べた既存研究の評価実験にて使用されている Sock Shop [6] を対象アプリケーションとする．Sock Shop は，コンテナ型仮想化環境を管理するための Kubernetes [12]（version 1.20）上に展開される．Sock Shop は，靴下を販売する電子商取引 Web サイトを模したマイクロサービス構成のデモアプリケーションである．

Sock Shop に対する負荷生成には，負荷テストツールである Locust[*5]（version 1.6.0）を用いる．ワークフロースケジューラにおけるスケジューリングを実現するために，Kubernetes 上で動作するワークフローエンジンである Argo Workflows[*6]（version 3.1）を使用する．Argo Work-

[*5] Locust: `https://locust.io/`
[*6] Argo Workflows: `https://argoproj.github.io/workflows/`

flows は，Kubernetes 環境で実行したい一連の処理の各ステップが，コンテナとして動作するようなワークフローを定義できる．故障注入には，Kubernetes が管理するコンテナに対して，プリセットで用意された様々な故障を注入可能な LitmusChaos（version 2.0）[4] を用いる．故障注入以外の検証処理などをワークフローに組み込めるように，LitmusChaos が備えるスケジューラではなく，Argo Workflows を使用する．Argo Workflows のワークフローのステップとして，LitmusChaos による故障注入を定義する．

Sock Shop アプリケーションからのメトリックの収集には，メトリックを収集・保存・取得するためのツールである Prometheus[*7]（version 2.20.0）を使用した．Kubernetes 上のコンテナのメトリックが，cAdvisor[*8]により公開され，Sock Shop の各マイクロサービスが受信する HTTP リクエストに関するメトリックが各マイクロサービスのアプリケーションにより公開される．Prometheus はこれらの 2 種類のメトリックを収集する．データセットリポジトリには，オブジェクトストレージである Google Cloud Storage[*9]を使用する．

以上の実装のためのソースコードは，著者らが管理する GitHub リポジトリ[*10]内に公開されている．

4. 実験と評価

本章では，3.2 節に示した実装を用いて，実際にデータセットを生成し，生成されたデータセットに第二の設計基準で述べた系列の状態分類手法を適用したときの評価を述べる．

実験環境 Kubernetes クラスタの自動管理サービスである GKE（Google Kubernetes Engine）[*11]を使用して作成した Kubernetes クラスタ（バージョン 1.16.13-gke.1）上に，5 個の Worker ノードを構成し，5 個のうち 4 個をマイクロサービスを配置するためのサービス用途，残り 1 個をデータ収集と負荷生成のための管理用とした．ノードのマシンタイプはすべて `e2-medium` とした．

実験設定 Sock Shop を構成するコンテナのうちの 8 種類のコンテナを予め指定し，各コンテナに対して，CPU 過負荷とメモリリークの故障を 5 回ずつ注入する計 90 回の故障注入を実行したときのデータセットを使用した．このときの Prometheus のデータ取得頻度は 15 秒とし，スロットの時間範囲を 30 分，スロットの時間範囲の最後の 5 分を故障注入期間とした．アプリケーションレベルメトリックとして，Sock Shop の最前段に配置される `front-end` コンテナの平均応答遅延，マイクロサービスレベルメトリックとして，当該マイクロサービスの平均応答遅延を選択した．cAdvisor が提供するメトリックのうち，CPU 過負荷の故障メトリックとして `user_cpu_user_seconds_total`，メモリリークの故障メトリックとして `user_memory_working_set_bytes` を選択した．1 種類のコンテナに対して，原則 3 つの系列が選択される．ただし，`front-end` コンテナについては，アプリケーションレベルメトリックと，マイクロサービスレベルメトリックが一致するため，2 つの系列が選択される．`carts` コンテナに対するメモリリーク故障注入時の繰り返しのうち 1 回分は，対応するスロットのデータ採取に失敗していたため，データセットには含まれていない．

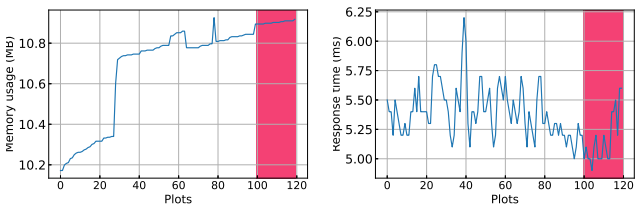
表 2: 系列の状態分類の正確度（故障別）

故障	系列数	正解系列数	正確度 (%)
合計	257	219	85
pod-cpu-hog:carts	15	5	33
pod-cpu-hog:carts-db	15	15	100
pod-cpu-hog:catalogue	15	15	100
pod-cpu-hog:catalogue-db	15	15	100
pod-cpu-hog:front-end	10	9	90
pod-cpu-hog:orders	15	15	100
pod-cpu-hog:orders-db	15	15	100
pod-cpu-hog:user	15	15	100
pod-cpu-hog:user-db	15	13	87
pod-memory-hog:carts	12	4	33
pod-memory-hog:carts-db	15	15	100
pod-memory-hog:catalogue	15	8	53
pod-memory-hog:catalogue-db	15	15	100
pod-memory-hog:front-end	10	10	100
pod-memory-hog:orders	15	11	73
pod-memory-hog:orders-db	15	11	73
pod-memory-hog:user	15	13	87
pod-memory-hog:user-db	15	15	100

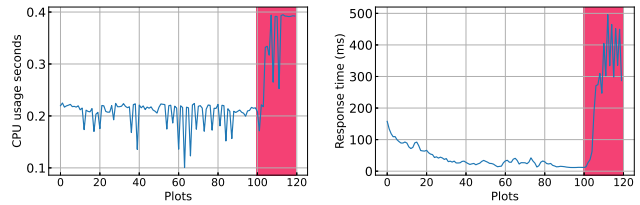
評価指標 本実験では，データセット内の全スロットの選択された全系列が `FOUND_INSIDE_ANOMALY` に分類されることを期待している．しかし，第二の設計基準より，実際に生成された系列を目視で分類したとしても，`FOUND_INSIDE_ANOMALY` となるとは限らない．そこで，実験で生成されたデータセット内の各スロットに対して，`NOT_FOUND`，`FOUND_OUTSIDE_ANOMALY`，`FOUND_INSIDE_ANOMALY` の 3 状態のうちのいずれかに 257 個の系列を目視で分類した．この目視による分類を正解としたときの系列の状態分類手法の正確度を分類性能の評価指標とする．

分類結果 表 2 に故障ケース別の状態分類手法の正確度を示す．256 個の全体の系列に対して，正確度は 85%であった．正解の状態別に仕分けると，197 個の `FOUND_INSIDE_ANOMALY` の系列に対して正確度が 98%，22 個の `FOUND_OUTSIDE_ANOMALY` に対して正確度が 68%，36 個の `NOT_FOUND` に対して 28%の正確度であった．した

*7 Prometheus: `https://prometheus.io/`
*8 cAdvisor: `https://github.com/google/cadvisor`
*9 Google Cloud Storage: `https://cloud.google.com/storage`
*10 `https://github.com/ai4sre/meltria/`
*11 Google Kubernetes Engine: `https://cloud.google.com/kubernetes-engine`



(a) メモリ使用量の微増　**(b)** ホワイトノイズ



(c) CPU 時間のスパイク変動　**(d)** 前回の故障注入影響の混入

図 3: 誤分類された時系列データの例

がって，NOT_FOUND が正解となる系列に対して，特に低い分類性能を示したと言える．低い分類性能を示したケースは，carts，orders，orders-db，catalogue の 4 コンテナについて，繰り返しの故障注入の一部が何らかの理由で実行されなかったため，異常が含まれておらず，正解の分類が NOT_FOUND となったケースである．

低い分類性能を示した要因を詳細に分析するために，図 3 に誤分類された時系列データの 4 つの例を示す．横軸はデータ点のプロット順であり，100 から 120 プロットの間の赤い影は故障注入期間である．図 3(a) では，メモリ使用量が 10.2MB に対して微増しているにすぎないため，正解は NOT_FOUND であるが，FOUND_OUTSIDE_ANOMALY と分類された．図 3(b) では，応答時間が広範囲に同程度の強度のノイズを含むホワイトノイズとなっているため，正解は NOT_FOUND だが，FOUND_OUTSIDE_ANOMALY と分類された．図 3(c) では，CPU 過負荷の故障注入により，故障注入期間に CPU が大幅に増加しているため，正解は FOUND_INSIDE_ANOMALY だが，FOUND_OUTSIDE_ANOMALY と分類された．図 3(d) では，直前の故障注入の影響により，左側に回復の変動が見られるため，正解は FOUND_OUTSIDE_ANOMALY だが，FOUND_INSIDE_ANOMALY と分類された．

分類結果に対する考察 誤分類されたケースのうち，図 3(a) と図 3(b) の例のように NOT_FOUND が正解のケースは，故障注入に失敗していたケースと一致した．2 シグマ則では，故障注入の成功時ほど大きな変動を示さない注入失敗時の系列が，FOUND_OUTSIDE_ANOMALY や FOUND_INSIDE_ANOMALY に分類された．仮に，故障注入に成功したとしても，故障注入の影響がもし小さければ，2 シグマ則では誤分類されると予想される．その一方で，3 シグマ則を用いると，正しく FOUND_INSIDE_ANOMALY と分類できていたケースで FOUND_OUTSIDE_ANOMALY と誤分類されるケースが増加することがわかっている．このような変動が小さなケースで正確に異常を分類するためには，類似の変動傾向であっても，メトリックの種類に依ることが障壁となる．そこで，メトリックごとの知識を教師データとすることが可能な，教師ありの機械学習モデルを適用することを今後検討する．本システムでは，注入される故障の種類，故障注入先の構成要素，検証対象のメトリックが予め定まっているため，教師データを作成しやすい．

データセット生成の課題 分類手法が正答したとしても，目視による分類が期待される FOUND_INSIDE_ANOMALY でなかったケースを考察する．第一に，あるスロットで注入された故障の影響がその直後のスロットまで残留していたため，FOUND_OUTSIDE_ANOMALY に正確に分類されたケースがあった．現在のワークフロースケジューラの設計は，故障の影響からアプリケーションが回復したのちに，スロットが開始されることを保証していない．したがって，アプリケーションの回復が遅れると，スロットに 2 つの故障の影響が混在することになる．スロット間の故障の影響の独立性を保証することは今後の課題とする．第二に，故障注入期間外の系列にごく短時間のスパイクが発生したため，FOUND_OUTSIDE_ANOMALY に正確に分類されたケースがあった．ごく短時間のスパイクはランダムで発生し，その要因がクラウド事業者が管理するシステムにある可能性もあるため，突出自体を発生させないようにすることは難しい．また，短時間の突出を無視して，FOUND_INSIDE_ANOMALY に分類すると，データ分析する際に，その短時間の突出の存在に気づかず分析結果に影響を及ぼす可能性がある．これらの第一と第二のケースを一括して FOUND_OUTSIDE_ANOMALY に分類するのではなく，より詳細の状態を定義し，分類が可能となるような分類手法を今後検討する．

5. まとめ

本研究では，マイクロサービスの異常検知・原因分析を対象とした既存研究の実験手法を調査することにより，実験に必要なデータセットに求められる要件を導出した．そして，導出された要件に基づき，故障注入のスケジューリング，データセットのラベル管理，および，統計学の経験則によるデータセットの自動検証を行う，データセットの自動生成システムである Meltria の設計基準を提案した．自動検証手法を評価するために，実際に生成されたデータセットに含まれる個々の時系列データを異常の状態別に分類したところ，85%の正確度となった．正確度をより高めるための手がかりは，故障注入に失敗するかあるいはその影響が小さかったケースにあることを示した．

今後の展望は，メトリック以外のデータ種別のデータセット生成，複数の異なる対象アプリケーションのサポート，多様な種類の故障のサポート，および，データセットの生成時間の短縮，を実現できるように，Meltria を拡張

することである．また，異種のアプリケーションや異種の故障のサポートに加えて，スロット内の故障注入期間の配置を変更するなどの，異常パターンを増加させるための条件を設定可能にしていく．最後に，データ種別ごとに，異常パターンを分類した上で，既存の異常検知・原因分析手法による分析精度が不十分となるような異常パターンを発見していく予定である．

参考文献

[1] 68-95-99.7 rule, https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule.

[2] A Benchmark Microservice System, https://github.com/FudanSELab/train-ticket/.

[3] Chaos Mesh: A Powerful Chaos Engineering Platform for Kubernetes, https://chaos-mesh.org/.

[4] LitmusChaos: Open Source Chaos Engineering platform, https://litmuschaos.io/.

[5] Pymicro: Microservice-based Sample Application Written in Python, https://github.com/rshriram/pymicro.

[6] Sock Shop: A Microservices Demo Application, https://microservices-demo.github.io/.

[7] Aggarwal, P., Gupta, A., Mohapatra, P., Nagar, S., Mandal, A., Wang, Q. and Paradkar, A., Localization of Operational Faults in Cloud Applications by Mining Causal Dependencies in Logs using Golden Signals, *International Conference on Service-Oriented Computing (ICSOC)*, pp. 137–149 2020.

[8] Arya, V., Shanmugam, K., Aggarwal, P., Wang, Q., Mohapatra, P. and Nagar, S., Evaluation of Causal Inference Techniques for AIOps, *8th ACM IKDD CODS and 26th COMAD*, pp. 188–192 2021.

[9] Avizienis, A., Laprie, J.-C., Randell, B. and Landwehr, C., Basic Concepts and Taxonomy of Dependable and Secure Computing, *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, pp. 11–33 2004.

[10] Basiri, A., Behnam, N., De Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J. and Rosenthal, C., Chaos Engineering, *IEEE Software*, Vol. 33, No. 3, pp. 35–41 2016.

[11] He, S., Zhu, J., He, P. and Lyu, M. R., Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics, *arXiv preprint arXiv:2008.06448* 2020.

[12] Hightower, K., Burns, B. and Beda, J., *Kubernetes: Up and Running: Dive into the Future of Infrastructure*, "O'Reilly Media, Inc." 2017.

[13] Jacob, V., Song, F., Stiegler, A., Rad, B., Diao, Y. and Tatbul, N., Exathlon: A Benchmark for Explainable Anomaly Detection over Time Series, *the VLDB Endowment (PVLDB)* 2021.

[14] Karumuri, S., Solleza, F., Zdonik, S. and Tatbul, N., Towards Observability Data Management at Scale, *ACM SIGMOD Record*, Vol. 49, No. 4, pp. 18–23 2021.

[15] Lin, J., Chen, P. and Zheng, Z., Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-Service Environments, *International Conference on Service-Oriented Computing (ICSOC)*, pp. 3–20 2018.

[16] Liu, P., Xu, H., Ouyang, Q., Jiao, R., Chen, Z., Zhang, S., Yang, J., Mo, L., Zeng, J., Xue, W. et al., Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks, *IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 48–58 2020.

[17] Ma, M., Xu, J., Wang, Y., Chen, P., Zhang, Z. and Wang, P., AutoMAP: Diagnose Your Microservice-based Web Applications Automatically, *The Web Conference (WWW)*, pp. 246–258 2020.

[18] Qiu, J., Du, Q., Yin, K., Zhang, S.-L. and Qian, C., A Causality Mining and Knowledge Graph Based Method of Root Cause Diagnosis for Performance Anomaly in Cloud Applications, *Applied Sciences*, Vol. 10, No. 6, p. 2166 2020.

[19] Soldani, J. and Brogi, A., Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey, *arXiv preprint arXiv:2105.12378* 2021.

[20] Soldani, J., Tamburri, D. A. and Van Den Heuvel, W.-J., The pains and gains of Microservices: A Systematic Grey Literature Review, *Journal of Systems and Software*, Vol. 146, pp. 215–232 2018.

[21] Wang, L., Zhao, N., Chen, J., Li, P., Zhang, W. and Sui, K., Root-Cause Metric Location for Microservice Systems via Log Anomaly Detection, *IEEE International Conference on Web Services (ICWS)*, pp. 142–150 2020.

[22] 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.

[23] Wu, L., Tordsson, J., Acker, A. and Kao, O., MicroRAS: Automatic Recovery in the Absence of Historical Failure Data for Microservice Systems, *IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pp. 227–236 2020.

[24] 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)*, pp. 1–9 2020.

[25] 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* 2018.

[26] 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, *the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 683–694 2019.