

ラピッドリリースを前提としたマイクロサービス性能障害の原因箇所絞り込み手法の検討

佐々木 裕介¹ 関口 敦二¹

概要: ラピッドリリースを前提として Microservice Architecture (MSA) で構築したシステムを複数の組織で運用する場合、複数の組織が様々な設定変更を MSA システム上で同時に実施する。しかし、複数の同時変更起因するテストの失敗、すなわち障害を解決することは困難である。本論文では、サービスマッシュの構成変更時、特に性能にかかわる障害に関して、障害原因の切り分け作業を網羅的かつ効率的に行う手法を提示する。本手法は、複数の設定変更を組み合わせた場合にのみ再現する障害について、効率的に原因箇所を特定できる。

Consideration of how to identify the cause of microservice performance failure assuming rapid release

1. 背景

Microservice Architecture (以降 MSA と略記) は、粒度の小さなサービス (マイクロサービス) 同士を互いに軽量なプロトコルで通信させることで、1つのシステムとするアーキテクチャである [1]。公になっている複数のウェブサービスでは、MSA を採用している [2][4]。それらのウェブサービスの中には特に、個々のマイクロサービスをコンテナイメージとしてテンプレート化し、Kubernetes system[5] 上で運用するという手法がとられているものがある。Kubernetes (以降 k8s と略記) とは、ウェブなどで実際のウェブサービスの提供に必要な冗長性、分散性などをもたせつつ、コンテナ化されたアプリケーションを自動的にデプロイ、スケールリング、管理するための OSS システムである。

MSA を採用することのメリットとして、高頻度なリリース (ラピッドリリース) が可能になることが挙げられる。MSA では、マイクロサービス間の結合が疎になるため、あるマイクロサービスの変更が他のマイクロサービスへ影響することが少なくなり、結果としてサービス全体での高頻度なリリースが可能になる。ある調査によると、人気のウェブサイト上位に位置する企業ほど高頻度なリリースを行っているという結果がある [6]。自社のビジネスや市場の変化に合わせてシステムを変化させることは、企業にとって重

要であり、MSA を採用する動機となる。

MSA 上のメッシュ状のサービス構造をサービスマッシュと呼ぶことがある。サービスマッシュ上のサービスディスカバリ、ルーティング、アプリケーションレベルの非機能通信要件にまつわる設定は、サービスが増えるほど複雑になり管理が難しくなる [7][8]。

そこで Istio[9] のように、サービスの通信設定をマイクロサービスの外で一元管理する OSS が開発された。Istio は特に、マイクロサービス個別の設定をせずとも、共通の方法で後述する timeout などの設定を一元管理できるというメリットをシステム全体の運用管理者へ提供してくれる。以降、サービスマッシュの設定とは Istio の設定を指すこととする。

1.1 MSA で構築したシステムで生じる障害事例

MSA で構築したシステムを開発運用する場合、複数の設定項目の変更などが互いに影響しあって複雑な障害を発生させることがある。Kubernetes Failure Stories[10] にまとめられている障害事例の中には次のようなものがある。

- OutOfMemory error, DNS 枯渇, CircuitBreaker の作動などが複合的に起き、1時間にわたって接続不能となる [11]
- CPU limit の制限を行ったことで Linux kernel の bug が発現し CPU throttling が起きてしまう [13]

¹ FUJITSU LABORATORIES LTD.

- k8s の bug として, scale-in に伴う node 削除時に contrack の routing 情報更新ミスが発現 [15]
- Logging ソフトウェアを切り替えようとしたが, 利用しているロードバランササービスの制約によって取りやめざるを得なくなった [12]

いずれの事例も最終的な問題解決には k8s や network, Linux kernel などの詳細な知識が必要になるが, その前に作業者の経験に基づきメトリックやログを眺めて異常箇所の目星をつける作業, すなわち, 問題の原因切り分け作業が発生している. そこで我々は, 障害発生時, 特に設定変更に起因する障害の問題の原因切り分け作業を, 作業者の経験に頼らずに効率化できないか検討している.

1.2 RQ

我々が興味のある問題は次のようなものである:

- 実際に我々が MSA でシステムを構築し運用するとして, 作業者の経験を元に障害要因を絞り込む方法以外に効率的な方法はあるか?

1.3 本研究の貢献

MSA で構築したシステムを開発, 運用する場合に生じる, サービスメッシュ周り, 特に性能劣化にかかわる障害解決に向けての切り分け作業を, 網羅的かつ効率的に行う手法を提示する.

- 特にラピッドリリースを前提として複数組織で MSA で構築したシステムを運用する場合に, 問題解決に関してどのような制約が生じるのか詳細に説明する
- 設定変更の組み合わせに起因する障害について, 網羅的に原因箇所を特定する手法を提示する

以降, 2 節で従来事例における MSA 採用企業と, 旧来の SI によるシステム開発を行ってきた企業とで, MSA を採用する場合にどのような差異があるのかについて, 2.4 節で作業者の経験に頼らず障害要因を絞り込む方法について, 3 節でさらにそれを効率化する手法についてそれぞれ述べる.

2. 対象とする開発体制

本稿で対象とする開発体制について述べる.

2.1 従来事例との比較

図 1 の左に示すように, 従来, MSA を採用している企業は, フラットな同一企業内のチーム間の連携で成り立っている.

- 原則としてそれぞれのチーム間の階層構造は存在しない
- それぞれの責任範囲は口頭やチャットなどによる約束であり, 変更が容易

一方, 旧来の SI によるシステム開発を行ってきた国内企業のように, フラットでない組織構造で MSA を採用しよ

うとすると, 様々な制約下での開発, 運用をしなければならない.

- それぞれ階層構造をもつ複数企業間のチーム
- それぞれの責任範囲は契約による約束で決まり, 変更が困難. 責任範囲に応じてタスクと報酬の支払方法が事前に固定されている
- 企業間の契約次第ではあるが, リリースに関する約束事として, 約束事を履行している証拠としての, 性能負荷テストをはじめとした試験の実施が必要となる

より詳細には, システムを使って業務を行う発注元の組織と, システムの開発運用保守まで行う受託開発元の組織が異なる. また受託開発元は実際には複数の組織で開発運用を行っている.

2.1.1 発注元

発注元は, ラピッドリリースを志向しており, 日に複数回のリリースがしたい. 一度最終成果物をリリースした後も, 発注側と受託側双方にサービスの価値向上を目指して高頻度にリリースを繰り返す動機が発生するよう, レベニューシェア [16] などの形式で提携を行うことがある. 日に複数回のリリースを実現するためには, 後述する高頻度の負荷テストと, テスト時に発生する問題の迅速な解決が必要となる.

2.1.2 各マイクロサービスの責任者

あるマイクロサービス単体の機能・性能負荷テストは, それぞれのマイクロサービスの責任において実施する. 個々のマイクロサービス内部の機能・性能負荷テストは, system testing 前に実施済みである. 本稿における system testing とは, 運用時に実際に受け取ることが想定される HTTP リクエスト等を, 本番環境同等の構成で処理するテストを実施することを指す.

本稿で想定するシステム全体は A, B, C, D の 4 つのマイクロサービスで成立している (図 2 右). それぞれのマイクロサービスを, A, B, C, D の 4 つの team が管理しており, それぞれに関し Atm, Btm, Ctm, Dtm の 4 つの設定を抱えている. それぞれのマイクロサービスが新たに反映したい設定変更の例を図 4 に示す. Timeout および weight (routing) の詳細については後述する.

- 左上 timeout: 外部もしくは他のサービスから, サービス A へリクエストが送られてきた場合の待機時間上限
- 左下 weight: サービス B が新旧 2 バージョンを同時稼働している場合に, 外部もしくは他のサービスからサービス B へリクエストが送られてきた場合の振り分けの比率
- 右上 timeout: 外部もしくは他のサービスから, サービス C へリクエストが送られてきた場合の待機時間上限
- 右下 timeout: 外部もしくは他のサービスから, サービス D へリクエストが送られてきた場合の待機時間

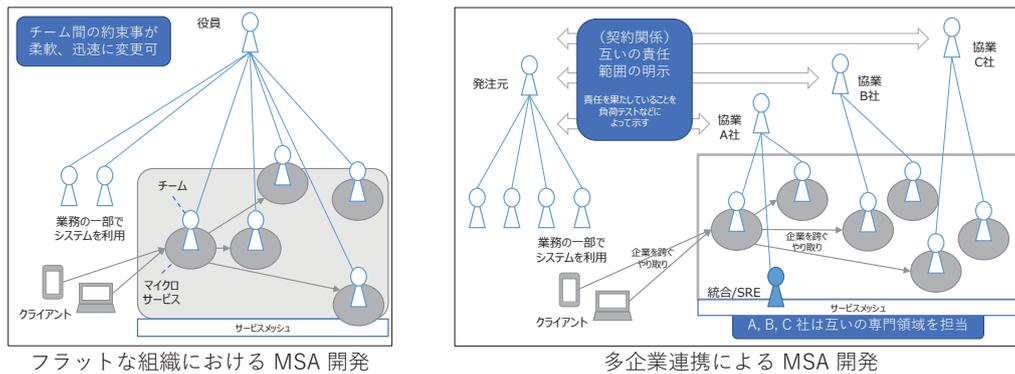


図 1 フラットな組織における MSA と多企業連携 MSA の比較

上限

2.1.3 SRE

図 1 に示すように、システム全体を管理する役職（以降では便宜的に Site Reliability Engineer, SRE と略記）がある。SRE は「モバイルからのアクセスの可用性・応答時間」などのクライアントに対するサービスレベルの維持に責任を持つ。

後述するように、個々のマイクロサービスに関する設定変更は、Git などの版管理システムで管理する。複数のチームからの変更要望が短期間で届くゆえに、本稿が対象とするシステムでは CI/CD cycle でのテスト頻度よりも commit 頻度の方が多い。このため、リリース前には各組織の変更がまとめてステージング環境に反映される。

従来の MSA システムでは、それぞれのチームが独自の判断で、運用環境上に次版をリリースするということが行われている。しかし特定のマイクロサービス単独の変更では問題なくても、多段の呼び出し関係にあるマイクロサービス群の変更が問題を起すことがある。そのため SRE としては本番環境での直接の試行錯誤による悪影響は減らしたい。また発注元から受託元に対しても、リリース前に実際の運用環境に近い環境（以降、ステージング環境）での system testing が求められることが想定される。

上述の環境下だと、短いリリース間隔の合間に、SRE が複数の設定変更をひとつのステージング環境で試す必要がある。特に複数の組織が絡む大きなシステムの場合、ステージング環境の準備コストが高く、開発元では精々 2, 3 の環境しか同時に用意できず、並行してテスト実施をして時間を短縮するにしても限度がある。そこで本稿では、SRE が、全体のサービスメッシュの調整を、従来よりも少ないステージング環境で行えるようにしたい。すなわち、障害が生じた場合でも、原因となった変更箇所を、人手でやるよりも迅速に絞り込めるようにしたい。

2.2 ステージング環境への設定変更反映プロセス

マイクロサービスは複数の組織が開発するサービスが協調連携することで成立している。複数の組織がそれぞれで

開発するため、設定項目もそれぞれの組織に関して存在する。スピードの速い開発ではそれらの設定変更要求を一括して処理し、十分な性能が出るか確認することになる。各組織における設定変更をステージング環境へ反映するまでの流れを図 2 に例示する。対象とするシステム開発では A から D team のすべての設定変更要求を Git 上で管理する。例えば以前の version のリリース (N 回目) では十分性能が出ていたものが、次のリリース (N+1 回目) それぞれの変更により障害を引き起こすことがある。そうすると、どの組織の変更が原因なのか早急に特定し修正する必要が出てくる。

Git 上での次 version リリース前の設定変更プロセスを図 3 に示す。リリース前には各組織の変更がまとめてステージング環境に反映される。例えば N 回目のリリース前には (N は任意の自然数)、N 回目のリリース用の環境を用意し、その上で性能負荷テストなどを含めた試験を実施する。想定した動作が確認出来たらリリースへ踏み切る。性能負荷テストが常にうまくいくとは限らず、N+1 回目のリリースには失敗する可能性もある。その場合はリリースせずに、性能負荷テストがうまくいくよう適宜設定変更して、再び試験を回す必要がある。

2.3 障害の具体例

以降では説明のため、マイクロサービス A から D のそれぞれの設定項目について改変を実施したところ、性能負荷テストに失敗したと仮定する。この場合に「障害の原因であるすべての改変項目」を抽出し、障害が起こる条件としてユーザに提示したい。

本稿では主に、次に示すような性能問題に関する障害を扱うこととする。

2.3.1 Timeout 設定による障害

Timeout[17] は、主にマイクロサービスの個々の障害を他のサービスへ波及させないために使われる設定である。Timeout 設定は、障害の起きている後段のサービスへのリクエストを前段のサービスが不必要に待つ必要をなくすことで、前段サービスへの負荷及びネットワークトラフィック

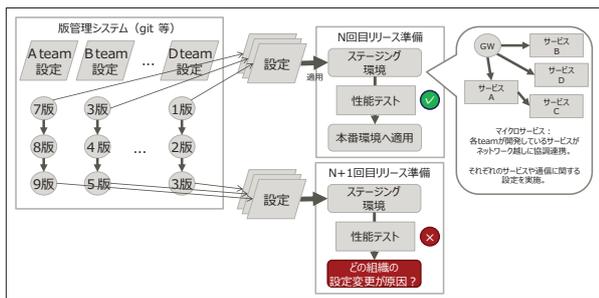


図 2 設定変更をステージングへ反映するまでの流れ

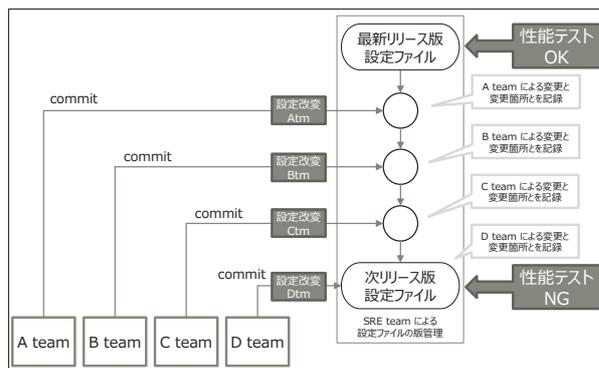


図 3 版管理システムを利用した次リリース前の設定変更プロセス

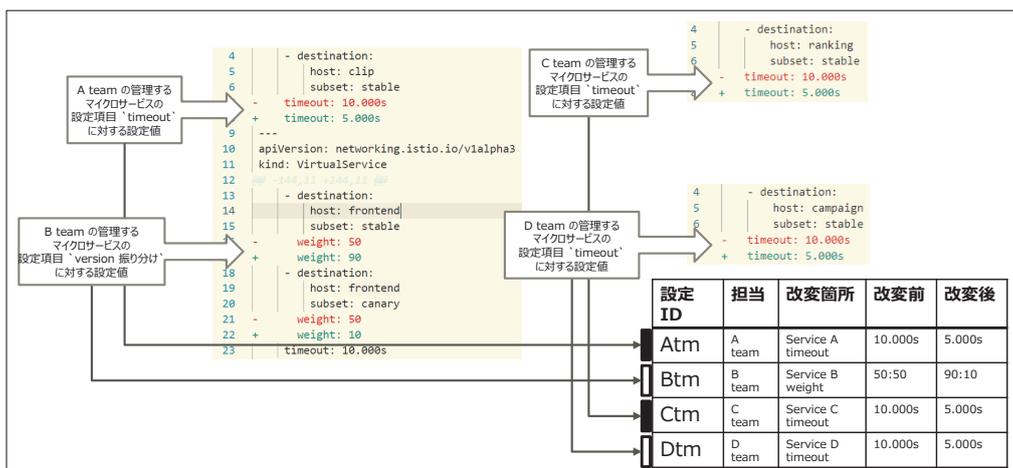


図 4 ステージング環境へ反映される設定変更の例

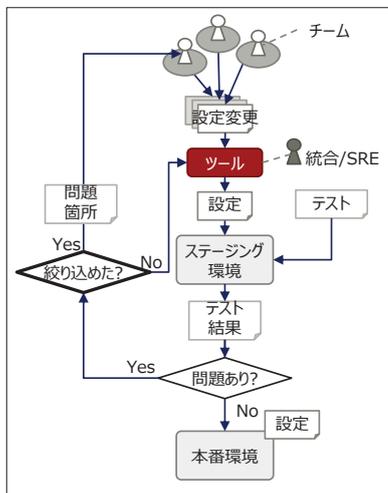


図 5 デプロイ作業におけるツールの位置づけ



図 6 論理式の簡易化

クの低減を実現する。

しかしながら timeout 時間以内に request 返答が返ってこない場合、後段のサービスが正常な場合でも接続を強制的に切ってしまうため、timeout を短くしすぎると無用なサービス障害が起きる。

2.3.2 Timeout および routing の同時変更による障害

Routing[18] は、A/B testing や blue/green deployment を目的として使われることの多い設定である。例えば、

HTTP レスポンスを返す v1 / v2 といった複数のバージョンが稼働するマイクロサービスが k8s 上に存在する場合に、v1 / v2 への routing 振り分け比率 (weight) を 8:2 などに設定すると、ユーザからのリクエストの 8 割を v1 へ、2 割を v2 へ振り分けることができるようになり、一部少数のユーザに対してのみ新規バージョン v2 のレスポンスを返すことができるようになる。このようにすることでユーザの反応を見ながら次期バージョンへの正式移行を検討す

るといったことが可能になる。

しかしながら、例えば v2 で新たな機能を追加したことで 1 リクエストを処理するのに必要なサーバリソース使用量が増え、単位時間当たりで処理可能なリクエスト数が減ってしまい、かつ v2 timeout の設定が短すぎる場合には v2 がレスポンスを返す前に毎回 timeout が来てしまい、v2 がレスポンスを返せないといった障害が生じうる。

2.4 シンプルな障害原因の探索手法

従来の障害事例では、例えば Linux kernel に関する障害は、(a) Linux kernel に関する変更を行ったこと、(b) Linux kernel に関する知識が調査者にある程度あることの 2 条件が揃えば、原因の目星をつけることができていた。SRE がすべてのマイクロサービスを熟知している場合にはこの方法で問題解決ができるが、企業が分かれているために、原因の解消に十分な知識を持っているという前提を立てにくい。特にマイクロサービスが多いほど、障害の原因特定に必要な知識は多くなる。そこで本稿では、SRE 側で、Git へ commit されている限られた情報、すなわち設定変更箇所と、その変更を行ったマイクロサービスがどれかという情報、そして性能負荷テストの実行結果の 3 種の情報だけで、障害に関与すると思しきマイクロサービスがどれかを絞り込む方法を検討する。目星がいたら、SRE が、障害に関与すると思しきマイクロサービスを管理する各チームへ、障害が解消するような設定へ再度変更できないか依頼を行う。

実際にどういう条件で障害が起きるのか、すなわちどの項目の改変によって障害が起きるのかを網羅的に調べるには、該当項目を少しずつ改変した、全設定バリエーションでの性能負荷テストの試行錯誤が必要となる。

例えば改変項目が A から D の 4 つあるとすると、図 7 に示すすべての設定バリエーションを調べる必要がある。4 項目未改変時の結果は前回リリースで既に結果が OK と分かっており、4 項目改変時の結果は今回のリリース前のテストで障害が生じており NG とわかっているため、試行から外すことができる。このため、改変項目が 4 つの場合、全部で $2^4 - 2$ 通りを調べる必要がある。特に C2-4 に示す通り、複数の項目改変で初めて NG となるバリエーションが存在することもある。このケースでは設定 C2-4 でのテストしなければこの 2 項目の変更が原因であるとは判別しにくくなっている。

仮に改変項目数を n とすると、バリエーション数は $2^n - 2$ 通りとなる。

3. 提案手法

以降に示す提案手法では、改変箇所の少ないバリエーションから先に試行錯誤し、性能負荷テスト結果が NG となるバリエーションの内のいくつかを、試行せずとも発見可能

とする。SRE の作業効率化のため、網羅的な設定バリエーションの生成及び試行不要なバリエーションを自動スキップするツール（以降、提案ツール）を開発した。ツールの入力から出力までの大まかな流れを図 8 に示す。

提案手法の大まかな流れは以下のようになっている（図 5）。

- S1. SRE: Istio 変更なし+性能負荷テスト (OK)
- S2. SRE: Istio 変更全て適用+性能負荷テスト (NG)
- 障害の原因箇所の絞り込みを開始
- S3. SRE: 提案ツールを実行
 - 提案ツール：図 7 に従い設定バリエーションの生成
 - 提案ツール：図 9,10,11,12 に従いスキップ可能な設定バリエーションを特定
- S4. SRE: まだ性能負荷テストを実行すべき設定ファイルが生成されたら S5 へ。されなければ S8 へ。
- S5. SRE: 生成された Istio 設定を Istio に適用し、性能負荷テスト実施
 - 今回は一部のみ自動化を実施
- S6. SRE: 性能負荷テストの結果を提案ツールの入力へ加える
 - 生成された設定バリエーションに対し、実際に性能負荷テストをして OK/NG どちらか結果を埋める
- S7. S3 に戻る
- S8. SRE: 提案ツールが提示する疑わしい変更箇所（図 4 を参照）を見て確認

3.1 具体例

例えば先に例示したケース（改変項目数が 4）の場合、図 9 の改変項目の組み合わせ表の「結果」を更新していく。

3.1.1 初期状態

最初は以下 2 つについて結果がわかっている。

- C0: 一つも改変していない状態（前回リリース時の状態）で性能負荷テストが OK であること
- C4: すべての項目を改変した状態（今回リリースしようとした状態）で性能負荷テストが NG であること

3.1.2 ループ 1

次に、改変箇所の少ないバリエーション、すなわち 1 つのみを改変したバリエーション C1-1, C1-2, ..., C1-4 を最も優先度の高い設定バリエーションと位置づけ、順に性能負荷テストを実施する。それぞれのバリエーションについて OK/NG を記録する（図 10）。

3.1.3 ループ 2

次に 2 つのみを改変したバリエーション C2-1, C2-2, ..., C2-6 を最も優先度の高い設定バリエーションと位置づけ、性能負荷テストを実施する（図 11）。

このとき、Atm と Dtm を同時改変した C2-4 について結果が NG と分かったとすると、本稿で開発したツールは、この改変を含む C3-3 および C3-4 も NG であると推定す

4項目未変更時		4項目変更時 (m=4)		「2項目変更で障害発生するか」を調査するバリエーション群 m = 2 (バリエーション数 C(4,2) = 4*3/2*1=6)						
結果	C0	結果	C4	結果	C2-1	C2-2	C2-3	C2-4	C2-5	C2-6
Atm	OK	Atm	NG	Atm	OK	OK	OK	NG	OK	OK
Btm		Btm	✓	Btm	✓			✓	✓	
Ctm		Ctm	✓	Ctm	✓	✓			✓	
Dtm		Dtm	✓	Dtm			✓	✓		✓

「単独変更で障害発生するか」を調査するバリエーション群
m = 1 (バリエーション数 C(4,1) = 4*1/1)

結果	C1-1	C1-2	C1-3	C1-4
Atm	✓			
Btm		✓		
Ctm			✓	
Dtm				✓

「3項目変更で障害発生するか」を調査するバリエーション群
m = 3 (バリエーション数 C(4,3) = 4*3*2/3*2*1=4)

結果	C3-1	C3-2	C3-3	C3-4
Atm	✓		NG	NG
Btm	✓	✓		✓
Ctm	✓	✓	✓	
Dtm		✓	✓	✓

複数項目の同時変更で初めてNGとなるバリエーション
A, D 単独ではNGとならない(左下表)

※mは同時変更する項目数

図 7 変更項目が4つの場合に4項目を少しずつ変更した全設定バリエーション

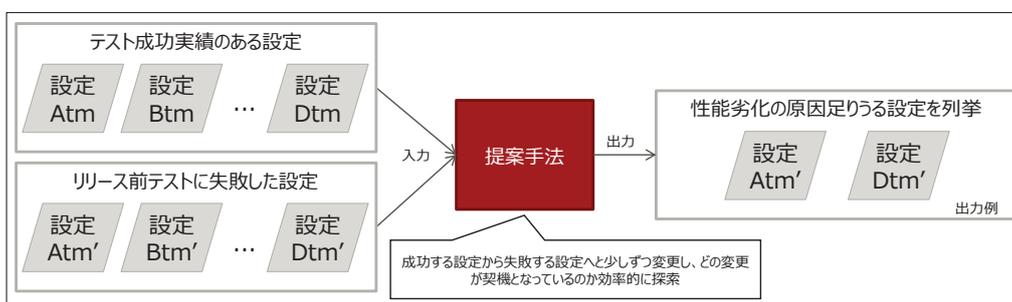


図 8 ツールの入出力概要

4項目未変更時		4項目変更時 (m=4)		2項目変更で初めて障害, m = 2 (バリエーション数 C(4,2) = 4*3/2*1=6)						
結果	C0	結果	C4	結果	C2-1	C2-2	C2-3	C2-4	C2-5	C2-6
Atm	OK	Atm	NG	Atm	Skipped					
Btm		Btm	✓	Btm	✓			✓	✓	
Ctm		Ctm	✓	Ctm	✓	✓			✓	✓
Dtm		Dtm	✓	Dtm			✓	✓		✓

「単独変更で障害発生するか」を調査するバリエーション群
m = 1 (バリエーション数 C(4,1) = 4*1/1)

結果	C1-1	C1-2	C1-3	C1-4
Atm	✓			
Btm		✓		
Ctm			✓	
Dtm				✓

3項目変更で初めて障害
m = 3 (バリエーション数 C(4,3) = 4*3*2/3*2*1=4)

結果	C3-1	C3-2	C3-3	C3-4
Atm	Skipped			
Btm	✓	✓		✓
Ctm	✓	✓	✓	
Dtm		✓	✓	✓

※mは同時変更する項目数

図 9 初期状態

4項目未変更時		4項目変更時 (m=4)		2項目変更で初めて障害, m = 2 (バリエーション数 C(4,2) = 4*3/2*1=6)						
結果	C0	結果	C4	結果	C2-1	C2-2	C2-3	C2-4	C2-5	C2-6
Atm	OK	Atm	NG	Atm	Skipped					
Btm		Btm	✓	Btm	✓			✓	✓	
Ctm		Ctm	✓	Ctm	✓	✓			✓	✓
Dtm		Dtm	✓	Dtm			✓	✓		✓

「単独変更で障害発生するか」を調査するバリエーション群
m = 1 (バリエーション数 C(4,1) = 4*1/1)

結果	C1-1	C1-2	C1-3	C1-4
Atm	Skipped			
Btm	✓			
Ctm		✓		
Dtm			✓	

3項目変更で初めて障害
m = 3 (バリエーション数 C(4,3) = 4*3*2/3*2*1=4)

結果	C3-1	C3-2	C3-3	C3-4
Atm	Skipped			
Btm	✓	✓		✓
Ctm	✓	✓	✓	
Dtm		✓	✓	✓

最初にテストするバリエーション群

※mは同時変更する項目数

図 10 ループ1

る(図12)。

3.1.4 ループ3

最後に3つのみを変更したバリエーション C3-1, C3-2, ..., C3-4 について、結果が埋まっていない部分について性能負荷テストを実施する。

3.2 最終出力

すべての結果を埋めたなら、ツールは性能負荷テストがNGとなる条件を論理式として出力する。出力の理解を容易にするため、ツールは論理式を簡易化した上でユーザに提示する。簡易化の方法として、例えば次に示すカルノー図に落とし込む方法などがある(図6)。最終的に導かれる論理式は「A 変更 and D 変更」となる。この例では A

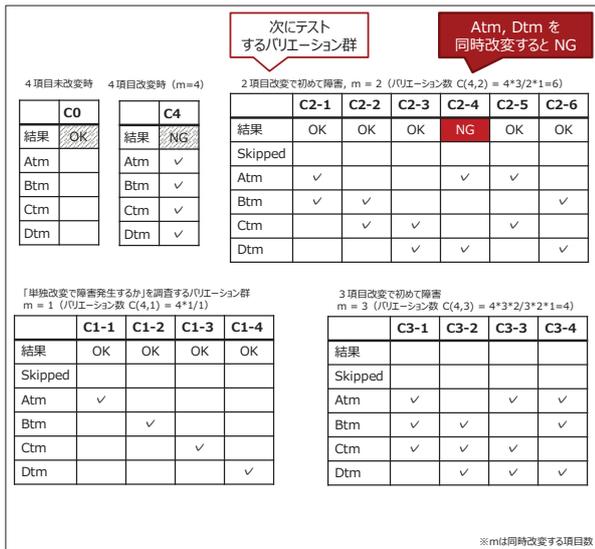


図 11 ループ2

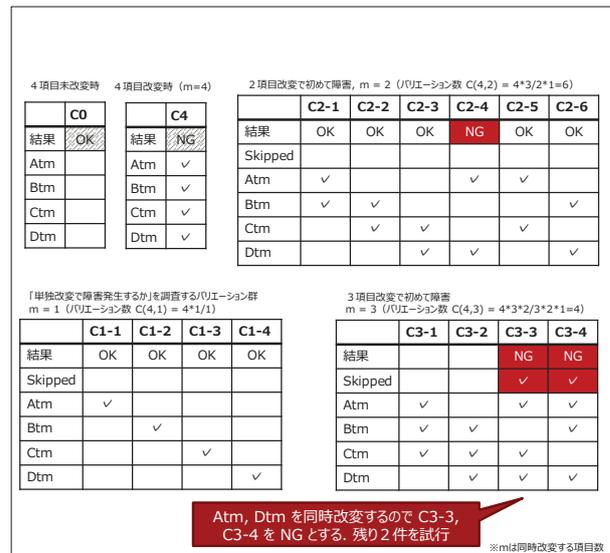


図 12 ループ3

team と D team の同時改変が原因であるので、A team と D team に対し「障害が起きるから、新規に commit された設定改変部分を協調して修正してほしい」と通知する。

3.3 手法の効果

3 節に示す例では、障害原因となる改変項目を絞り込むために、スキップなしの探索で $2^4 - 2$ 回、すなわち 14 回の試行錯誤が必要であった。本稿で示す効率化により計 2 回 (約 14%) のスキップができるため、試行錯誤の回数は 12 回に減らせる。上記では C2-4 が NG となる例を挙げたが、例えば C1-1 が NG であったなら、

- C2-1, C2-2, ..., C2-6 のうち ${}_3C_1$ バリエーションが削減できる。これは C1-1 の変更箇所に加えて 1 か所を変更したものが重複関係となるためである
 - C3-1, C3-2, C3-3, C3-4 のうち ${}_3C_2$ バリエーションが削減できる。これは C1-1 の変更箇所に加えて 2 か所を変更したものが重複関係となるためである
- よって計 6 回 (約 43%) スキップできる。

他の例として、例えば入力となる設定の改変項目数 $n = 10$ であると仮定する。スキップなしだと試行錯誤の回数は $2^{10} - 2$ 回、すなわち 1022 回だが、上記と同様に考えると、1 つのみ改変したバリエーションで結果が NG となる場合は ${}_9C_1 + {}_9C_2 + \dots + {}_9C_8 = 510$ 回 (約 50%) スキップできる。一方、1 つを改変するバリエーションと、2, 3, ..., 4 つを同時改変するバリエーションについては結果が NG とならず、5 つを同時改変するバリエーションで初めて結果が NG となる場合、 ${}_5C_1 + {}_5C_2 + \dots + {}_5C_4 = 30$ 回 (約 3%) のスキップにとどまる。このように本手法は、入力となる設定の改変項目数 n が大きく、かつ 3 節に示すループにおいてより早期のタイミングで NG が見つかるほど、多くのバリエーションをスキップできる。

4. 関連文献

坪内らは、性能異常の診断にメトリック、テキストログ、実行トレースの利用を挙げ、メトリックに着目して因果関係を推定する手法 TSifter[19] を提案している。また Zhou らは、system trace log と fault injection および fault localization を用いて、MSA アプリケーションの潜在エラー有無およびその欠陥がどのマイクロサービスにあるのか予測を行っている [20]。System trace の利用については未だ検討の余地があるが、学習対象のデータを集めるための運用期間を必要とすることから、今回は結果がすぐ出せる、より単純な手法を検討している。

また Zhou らは、Hierarchical Delta Debugging (HDD)[21] を使い、環境構築の設定を少しずつ変えながら、複数テストケースを実行させた時のテスト結果の差分から、問題のある設定の検出を行った [22]。HDD は入力の構造を考慮した枝刈りを行うことで、バリエーションの数を減らすアプローチである。実験では、実装した MSA system である TrainTicket を対象に、3 つの問題のある設定を、各々 18~30 分で検出している。原因箇所を特定した障害として、例えば JVM と Docker のメモリ上限に矛盾があった際に JVM process が Docker process により強制停止される現象を挙げている。Zhou らの手法は、Node, Instance の種類, resource 割り当て Configuration, 非同期呼び出しの実行・戻り順序, MSA システムへの input に起因する障害を特定するため、これら 5 種類の情報を入力として必要とする。入力が多く複雑であると、利用者である SRE がツールに慣れるまで時間がかかるため、本稿ではより単純な手法を検討している。

Hodován らは HDD の試行回数を低減する工夫を行っている [23]。一つ一つのマイクロサービスの設定項目の構

造を考慮する HDD のような手法は本研究でも有用と思われる。

5. まとめと今後の方針

本稿では、まず多企業連携 MSA の特徴及び制約について述べた。次にサービスメッシュの設定変更時の性能にかかわる障害に関して、特に特定の設定変更を組み合わせたときのみ再現する障害であっても、作業者の経験に頼らず障害原因の切り分け作業を網羅的かつ効率的に行う手法を検討した。

今後は OSS 公開されているマイクロサービス、あるいは社内のマイクロサービスを対象とし、本手法が有効といえるか評価を行いたい。

参考文献

- [1] Microsoft: Microservices architecture, Microsoft (online), available from <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture> (accessed 2021/02/01).
- [2] Netflix: Netflix Conductor: A microservices orchestrator, Netflix (online), available from <https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40> (accessed 2021/02/01).
- [3] Ma, X.: Microservice Architecture at Medium, Medium (online), available from <https://medium.engineering/microservice-architecture-at-medium-9c33805eb74f> (accessed 2021/02/01).
- [4] Xia, D.: Keynote: How Spotify Accidentally Deleted All its Kube Clusters with No User Impact, Spotify (online), available from <https://www.youtube.com/watch?v=ix0Tw8uinWs> (accessed 2020/12/24).
- [5] The Linux Foundation: Kubernetes, The Linux Foundation (online), available from <https://kubernetes.io/> (accessed 2020/12/24).
- [6] CircleCI: Three Critical Development Metrics for Engineering Velocity, CircleCI (online), available from <https://circleci.com/ja/resources/velocity-report/> (accessed 2020/12/24).
- [7] Red Hat: What's a service mesh?, Red Hat (online), available from <https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh> (accessed 2020/12/24).
- [8] Bryant, D.: Service Mesh Ultimate Guide: Managing Service-to-Service Communications in the Era of Microservices, InfoQ (online), available from <https://www.infoq.com/articles/service-mesh-ultimate-guide/> (accessed 2020/12/24).
- [9] Istio: What is Istio?, Istio (online), available from <https://istio.io/latest/docs/concepts/what-is-istio/> (accessed 2020/12/24).
- [10] Jacobs, H.: Kubernetes Failure Stories, Codeberg (online), available from <https://codeberg.org/hjacobs/kubernetes-failure-stories> (accessed 2020/12/24).
- [11] Jacobs, H.: Total DNS outage in Kubernetes cluster, GitHub (online), available from <https://github.com/zalando-incubator/kubernetes-on-aws/blob/dev/docs/postmortems/jan-2019-dns-outage.md> (accessed 2020/12/24).
- [12] PrometheusKube: Why we switched from fluent-bit to Fluentd in 2 hours, PrometheusKube (online), available from <https://prometheuskube.com/why-we-switched-from-fluent-bit-to-fluentd-in-2-hours> (accessed 2020/12/24).
- [13] Khun, E.: Kubernetes: Make your services faster by removing CPU limits, Eric Khun (online), available from <https://erickhun.com/posts/kubernetes-faster-services-no-cpu-limits/> (accessed 2020/12/24).
- [14] Lerko, D.: Kubernetes Networking Problems Due to the Contrack, Dmitri Lerko (online), available from <https://deploy.live/blog/kubernetes-networking-problems-due-to-the-contrack/> (accessed 2020/12/24).
- [15] Umerov, A.: DNS issues in Kubernetes. Public postmortem, Preply (online), available from <https://medium.com/preply-engineering/dns-postmortem-e169efd45afd> (accessed 2020/12/24).
- [16] Ross, S.: How Revenue Sharing Works in Practice, Investopedia (online), available from <https://www.investopedia.com/ask/answers/010915/how-does-revenue-sharing-work-practice.asp> (accessed 2020/12/24).
- [17] Istio: Traffic Management, Istio (online), available from <https://istio.io/latest/docs/concepts/traffic-management/> (accessed 2020/12/24).
- [18] Istio: Request Routing, Istio (online), available from <https://istio.io/latest/docs/tasks/traffic-management/request-routing/> (accessed 2020/12/24).
- [19] 佐樹坪内, 博文鶴田, 雅大古川: TSifter: マイクロサービスにおける性能異常の迅速な診断に向けた時系列データの次元削減手法, インターネットと運用技術シンポジウム論文集, Vol. 2020, pp. 9–16 (2020).
- [20] 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*, ESEC/FSE 2019, New York, NY, USA, Association for Computing Machinery, p. 683–694 (online), DOI: 10.1145/3338906.3338961 (2019).
- [21] Misherghi, G. and Su, Z.: HDD: hierarchical delta debugging, *Software Engineering, International Conference on*, Los Alamitos, CA, USA, IEEE Computer Society, pp. 142–151 (online), DOI: 10.1145/1134285.1134307 (2006).
- [22] Zhou, X., Peng, X., Xie, T., Sun, J., Li, W., Ji, C. and Ding, D.: Delta Debugging Microservice Systems, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, New York, NY, USA, Association for Computing Machinery, p. 802–807 (online), DOI: 10.1145/3238147.3240730 (2018).
- [23] Hodován, R., Á. Kiss and Gyimóthy, T.: Coarse Hierarchical Delta Debugging, *2017 IEEE International Conference on Software Maintenance and Evolution (ICSM)*, pp. 194–203 (online), DOI: 10.1109/IC-SME.2017.26 (2017).