マイクロサービスにおけるメトリクスによるサービスの優先順位および計算リソースの共助モデル

飯島 貴政[1]　串田 高幸[1]

概要：クラウドアプリケーションにおけるマイクロサービスでの導入時にはサービスの優先度をユーザーが構築ファイルに記述する必要があった．そのため, ノードの CPU が枯渇した際, ノードを新たに追加することでスケールしていた．本稿ではマイクロサービスの優先順位を計算し, 優先順位が高いサービスを低いサービスが共助することで処理に必要とされる CPU をクラスター内から再利用する．前段階として, マイクロサービスの相互通信のログやノードにおける CPU 使用率を取得する．取得した指標を提案する計算式に代入することでマイクロサービスが複数ある環境で優先度の順位付けを行う．このモデルでは異なるコンテナでの優先度の比較を行い, 優先度が高いマイクロサービスの処理を低いコンテナが共助する．評価方法は, マイクロサービスが 15 個稼働している環境にウェブサイトを用意した．リクエストの急増時, ここでは恒常時のリクエスト回数の 5 倍のトラフィックを処理した際のレスポンスタイムを既存のスケール手法を比較した．また，上記の優先順位付けによるノード追加頻度を評価した．リクエストの急増時のレスポンスタイムは既存スケールから最大 58%減少した．

1. はじめに

背景

クラウドを用いてアクセスより柔軟に対応できる近年の WEB アプリケーションを構築する際には以下の 3 つの構築方法がある．

- クラウドで利用できるコンテナを作成/ビルドして動作させる
- クラウドアプリケーションを単機能に分割し相互に連携させる
- クラウドでプログラムコードのみをおいて実行させる

1 番目はコンテナオーケストラレーションと呼ばれ, これは既に OS のカーネル部分を共通化し, その上に OS がインストールされたコンテナと呼ばれるものをカスタマイズすることで起動や再起動が高速化した．これにより構築ユーザーの負担が大きく軽減されたため, ビジネスにおけるアイデアから実装までの時間が高速化した．

2 番目の選択肢はマイクロサービスアーキテクチャと呼ばれ, コードベースを機能ごとに分割し, ビジネスの機能と開発チームとの高い頻度での連携が可能となった [1]．クラウドサービスが使われる背景として, ダイナミックに変化するトラフィックに対して可能な限り人間の介入なく柔軟なリソースの調整を期待されている [2]．しかしクラウドアプリケーションで用いられるマイクロサービスでは, スケーラビリティを担保するためにはマイクロサービスごとのリソース, ここでは CPU やメモリの使用量から 1 つあたりの pod のスペックを明示的に策定する必要があった．

3 番目はサーバレスと呼ばれ, マイクロサービスをさらに進化させたものである．構築ユーザーはプログラミング言語を選択するだけで自動的にプログラムの実行環境を構築できるようになった．プログラムの実行がより容易になる一方で構築ユーザーはどのようなタイミングでスケールするのか, ノードのリソースはどのように管理されているのかといった IaaS で管理できるパラメータとは縁遠くなる．

そのため, 現段階でのクラウドアプリケーションでは 3,4 番目の (マイクロサービス, サーバレス) の課題を解消する仕組みが必要である．

マイクロサービスおよびサーバレスではそれぞれのサービスのコードベースが独立しているがゆえにコンテナリソースの使いまわしができない．従来の研究及びクラウドアプリケーションにおけるマイクロサービスでの導入時にはサービスの優先度を人間が事前定義する必要があった．

そのため，ノードにおけるリソースの割り振りにおいても人間のオペレーションが発生する．

[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
〒 192–0982 東京都八王子市片倉町 1404-1

課題

本研究における課題はマイクロサービスアーキテクチャを使用している際，コンテナオーケストレーターのデフォルトでのオートスケーラの設定を用いると，ノードの CPU を有効に活用せずに，新規ノードを追加することである．図 1 はリクエストが急増した際の従来のスケール手法である．1 ノードに付き 8vCPU が搭載されている．ここでは商品ページサービスが必要 vCPU が 4, 在庫管理サービスの必要 vCPU が 3 とする．ここで，商品ページサービスに

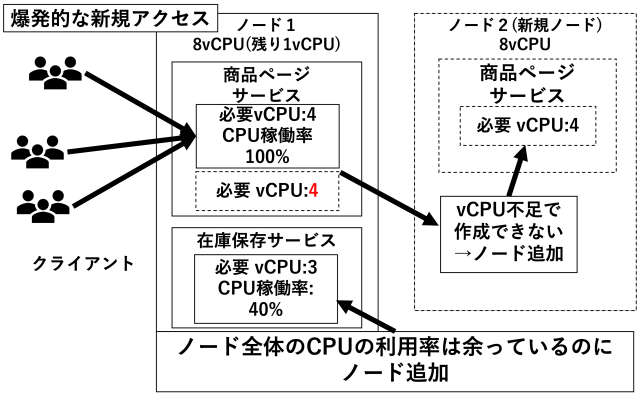

図 1 リクエストが急増した際の従来のスケール手法

対してサービスの vCPU では処理できない量のアクセスが急増した時，新規ノードを追加してデプロイしていた．ここでは vCPU1 コアと, 在庫管理サービスの利用していない 60%がノードに余っている．そのためノードにおける CPU 使用率を全て使い切っていないといえる．

各章の概要

第 2 章では本論文の関連研究について述べる．第 3 章では第 1 章で述べた課題を解決するシステムの提案について述べる．第 4 章では提案するシステムの実装と実験環境について述べる．第 5 章では提案するシステムの評価手法と分析手法について述べる．第 6 章では提案, 実験, 評価に関する議論について述べる．第 7 章では本研究の結果から得られた成果について述べる．

2. 関連研究

クラウドにおけるキューへの優先度付けによるタスクスケジューリングではクラウド環境におけるタスクをキューに分割し，キューに GA(遺伝的アルゴリズム) を用いて優先度を付与した [3]．この手法ではクラウド環境における厳格なタスクの優先順位を算出した．しかし，この手法をマイクロサービスに適用した時にはそれぞれの機能がネットワークを介しているため，このモデル通りにタスクをスケジューリングすることは困難である．

クラウドコンピューティングにおける負荷分散のためのユーザー優先度ガイド付き Min-Min スケジューリングアルゴリズムではヘテロジニアスな環境が多いクラウド全体で見た CPU, メモリといったクラスターリソースの負荷分散のために，Min-Min スケジューリングアルゴリズムを用いることでクラウド全体 (以降クラスター) のリソースよりユーザーあたりが使用できるリソースの優先順位を求めている [4]．この手法ではクラスターにおいてユーザー単位, 多くは企業やグループといった単位でのリソースの管理が実現できているが，やはりこれもマイクロサービスのようなネットワークを通じて高度に機能と機能が相互通信している場合に課題が残っているとしている．また，この手法では VIP レベルと呼ばれる優先度をユーザーが手動でシステムに教えることでタスクごとの優先度を考慮した負荷分散ができるとしている．しかし，タスクやサービスの数が爆発的に増加するマイクロサービスではこれらの優先度を自動で設定すべきである．

マイクロサービスバックエンドシステムのクラウドリソースプロビジョニングのためのオンライン機械学習ではマイクロサービスに特化したアプリケーションのスケール手法を提案している [5]．この提案では機械学習を用いることでサービスごとの適切なプロビジョニングを行い，スケーリングを行っている．しかし彼らが議論で述べているように，現状彼らは CPU 使用率という一つのメトリックのみを用いている．そのため，CPU の使用量よりストレージの帯域使用量が多い時の方がメトリックとして価値のあるデータベースには対応できていない．

3. 提案方式

全体構成図

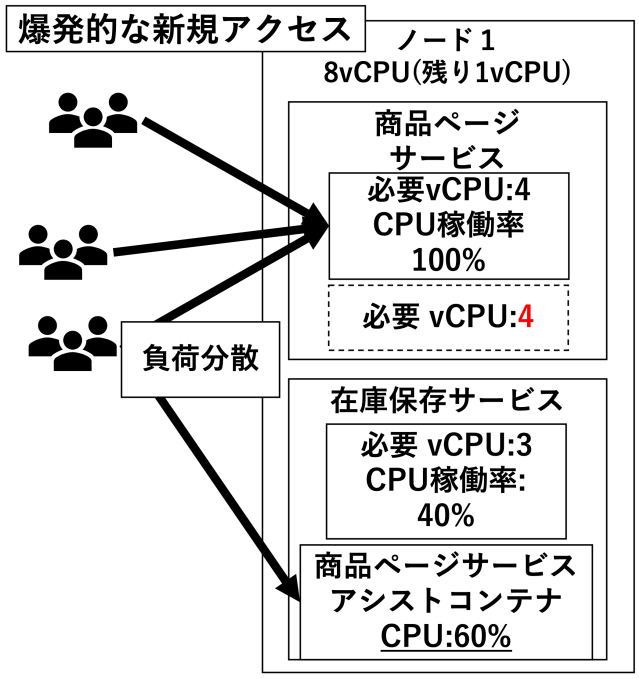

図 2 同一ノード内の使用していないリソースを用いることでノードの使用率を向上させる．

アルゴリズム 1 リクエストレート順にメトリクスの値を整理する

Input:
ML: メトリクスの種類が格納されたリスト
R:(時間 t(time_t): リクエストレート (req_rate) のハッシュマップ,
M:(時間 t(time_t): 各メトリクスの取得値) のハッシュマップが全てのメトリクス分格納されているハッシュマップ
Output: I : {}: メトリクス名ごとにリクエストレート (req_rate) に紐付いたメトリクスの取得値を格納するハッシュマップ

```
1: function METRICS_MAPPING_BY_REQ_RATE(ML, R, M)
2:     ▷ R に格納されている時間をキーにメトリクスの取得値を
3:     ▷ リクエストレートと紐付ける
4:     for all time_t, req_rate ← R do
5:         for all metrics ← ML do
6:             I.insert(metrics, {req_rate: M[metrics][time_t]})
7:         end for
8:     end for
9:     return I
```

マイクロサービスを実際に動作させている環境，ここでは Kubernetes の Pod のメトリクスを可能な限り取得し，メトリクスの優先度を取得する．

3.1 取得できるメトリクスの値とリクエストレートのマッピング

取得したメトリクスをもとにリクエストが増加したときに Pod 内で影響を受けるリソースを特定し，各マイクロサービスに上位 3 リソースのラベル付けを行う．アルゴリズム 1 はメトリクス及びリクエストレートを取得した時間 time_t を用いてマイクロサービスから取得した複数メトリクスをそれぞれレクエストレートに紐付ける処理である．

2 入力としてメトリクスの種類, 例えば CPU 使用率,RAM 使用率が格納されたリスト ML, システム上での時間とリクエストレートのハッシュマップである R, システム上での時間と個別メトリクスの値をハッシュマップである M が用意されている．出力はメトリクスごとに計測されたリクエストレートと同一時刻に取得されたメトリクスを格納したハッシュマップ I とする．関数 $METRICS\_MAPPING\_BY\_REQ\_RATE$ ではじめに R から時刻 $time\_t$ とリクエストレートである req_rate を R の要素分取得する．その後, 各メトリクスごとに $time\_t$ をキーに M からリクエストレートごとのメトリクスの取得値を I に挿入する．

3.2 相関係数および回帰直線の算出によるメトリクスの順位付け

アルゴリズム 2 はアルゴリズム 1 で取得した I を用いて，各マイクロサービスの複数メトリクスから相関係数及び回帰直線を用いてマイクロサービスにおけるメトリクスの重要度を算出し，リスト PoM に保存する．その後算出した重要度を基に昇順ソートし，最終的な各マイクロサービスのメトリクスの優先度となるリスト PoM_Final を作成する．

アルゴリズム 2 各マイクロサービス内のメトリクス順位を算出する

Input:
I: アルゴリズム 1 で作成したリクエストレートごとのメトリクスの値が格納されたリスト
Output: PoM_Final: メトリクスの重要度順位リスト

```
1: function SELECT_METRICS(I)
2:     ▷ 各メトリクスごとにリクエストレートとの相関係数を算出する
3:     for all metrics ← I do
4:         for all req_rate, metrics_value ← metrics do
5:             x(array) ← req_rate
6:             y(array) ← metrics_value
7:             ▷ 相関係数を算出する ρ = cov(X,Y) / (σ_x σ_y)
8:         ▷ 相関係数の 2 乗が 0.64 以上のものを有効な相関とみなす
9:         if ρ^2 > 0.64 then
10:            ▷ 回帰直線を算出する
11:            r = Σ_{i=1}^{n}(x_i − x̄)(y_i − ȳ) / (√Σ_{i=1}^{n}(x_i − x̄)^2 √Σ_{i=1}^{n}(y_i − ȳ)^2)
12:            PoM.insert([metrics, r])
13:        end if
14:    end for
15: end for
16:
17: function GET_METRICS_RANK(PoM)
18:     PoM_Final = {}
19:     max = 0
20:     for all selected_metrics_r ← PoM do
21:         for all metrics, r_value, index ← PoM_Final do
22:             if PoM_Final.Length()is0 then
23:                 PoM_Final.insert([metrics, r_value])
24:             elseselected_metrics_r > r_value
    PoM_Final[index − 1].insert([metrics, r_value])
25:             end if
26:         end for
27:         return PoM_Final
```

2 入力としてアルゴリズム 1 で作成したリクエストレートごとのメトリクスの値が格納されたリスト I がある，出力はマイクロサービスのメトリクスの優先度が昇順に保存されているリスト PoM_Final である．1 行目の function $SELECT\_METRICS$ を以下に説明する．3,4 行目では I よりメトリクスごとにリクエストレートをキー，各メトリクスの値をバリューを再帰的に取得する．また，各メトリクスの処理中におけるリクエストレートの集合を x, メトリクスの値の集合を y と示す．各メトリクスとリクエストレートにおいて x および y を用いて相関係数 ρ を計算する．相関係数を 5 段階に人為的に分類した先行研究と同様の分類を行い ρ^2 が 0.64 以上のものを相関があるものとみなし，相関メトリクスリスト PoM に挿入する．ここで作成された PoM を用いて $GET\_METRICS\_RANK$ 関数

を実行する．*GET_METRICS_RANK* 関数では，*PoM* に保存されたメトリクスを回帰直線の傾きの大きさ順にソートする．このソートを行うことにより，メトリクスの値がリクエストレートに影響を与えている度合いによって順位づけすることができる．この順位は後のマイクロサービス全体でのメトリクスの優先度に用いられる．

3.3 リクエストレートの採用理由

リクエストによる影響とメトリクスの値との関係を取得するためにサービスにかかるリクエストの数値を統一化する必要がある．本研究では毎秒におけるリクエストレート (単位 [req/s]) とした．以下にマイクロサービスへのリクエストにおける他の指標及びパラメータが指標として不適切である理由を述べる．(メトリクス=¿リソースに影響があると仮定して取得するデータ) マイクロサービスが受信するリクエストの中身として，例えば GET,POST といったヘッダーの種類，データサイズ, ステータスコード, ペイロード (実データ部分) が挙げられる [6]．ここではマイクロサービスアーキテクチャの設計原則から，マイクロサービスとマイクロサービスでのやり取りは RESTful API で行われているものとする．RESTful API におけるデータの送受信では JSON 形式に基づいたテキストデータを用いることが一般的である [7]．これらのテキストデータのサイズが肥大化すると通信における遅延およびネットワーク内におけるパケットロスが発生した際の損失したパケットの修復といったオーバーヘッドを発生させる (要参考文献 3)．(例えばどのくらいのデータサイズでどのくらいのオーバーヘッドがあるか例を書く) そのため，RESTful API を用いた際のリクエストにおけるデータサイズはマイクロサービスでの処理にかかるリソースへの影響が少ないといえる．またペイロード部分は実際にマイクロサービスが処理を実行することに必要なデータ, ここではマイクロサービスの処理が用いるテキストデータやテキストデータの配列が含まれていることが一般的である [8]．ペイロードは各マイクロサービスによって大きく内容が異なるため，メトリクスとしては不適切である．例えば検索ポータルにおける検索バーに文字を入れて検索ボタンを押したときと，SNS(Social Networking Service) に投稿をアップロードするボタンを押したときに全く処理が異なることがわかる．両方の処理はテキストデータをペイロードとして持っていることである．しかしながら前者は検索窓に入れた検索したいテキストデータを，後者は自分が投稿したい内容が書かれたテキストデータである．このように同じテキストデータだとしてもペイロードに含まれた情報がシステム内の異なるマイクロサービスで共通な要素を持つとは限らない．そのためマイクロサービスのシステム全体の指標としては不適切である．

そのため，本研究においては，相関係数をユーザで意図的に５段階に分けた研究を参照する．当該研究においては相関係数を 0.2 ごと５段階に分けたものを更に 2 乗した指標を用いている．ここでは相関係数が 0.8 1.0 の 2 乗の値である 0.64 1.00 の範囲にあるメトリクスを有効な相関とみなす．11 行目で有効な相関が取得できたメトリクスの値とリクエストレートについて回帰直線を算出する．

全てのマイクロサービスでその処理が終わり次第，ラベル付けが行われたものの数を加算し，マイクロサービスアーキテクチャを採用しているシステムの上位 3 ラベルを決定する．1 位には 3 ポイント, 2 位には 2 ポイント,3 位には 1 ポイントと重み付けを用いる．

また，同時に各サービス開始当初からのアクセス数の平均を取得するその後, マイクロサービスの優先度を作成する．ここで作成した優先度の上位から順に別にシステム内で類似の Pod がないか検索する．Pod は Docker コンテナ技術を基に構成されている．Docker コンテナのランタイム及びライブラリを比較する手法を用いて，マイクロサービス上で実行されているプログラムをプログラムファイルのコピーのみ，もしくはライブラリの追加を短い時間で行える Pod を検索する．ここでの短いとはライブラリの追加インストール時間が，同一の Pod をシステム上に新規作成する時間よりも短い場合を指す．検索した結果を基に優先度の高いマイクロサービスから低いマイクロサービスへのプログラムコピーを行う．その後，プログラムコピーを送った先の Pod に負荷分散するためのロードバランサーを設置する..

その後優先度が高いものから低いものに対してリクエストをリダイレクトする．

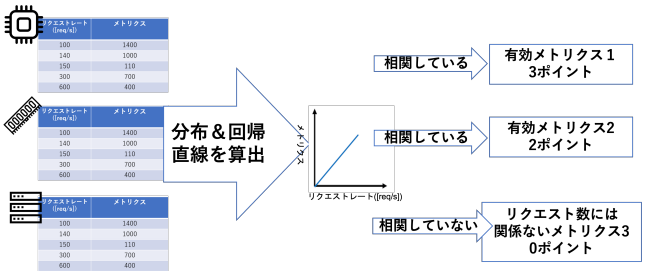

図 3 リクエストレートごとにメトリクスを取得

取得したメトリクスとリクエストレートの相関係数が 0.64 以上のものに関して，メトリクスとリクエストレートの相関があるものと認め，以下の手順に進みリソースに影響を与えるメトリクスを順位づけする

．上記の条件にそったメトリクスとリクエストレートについて以下のようなグラフを作成する．

縦軸は取得したメトリクスの値で，横軸はリクエストレートである．その際，回帰直線の傾きが大きい順にメトリクスを各マイクロサービスで順位付けする．以降ここで算出されたメトリクス郡を影響メトリクスと呼ぶ．各マ

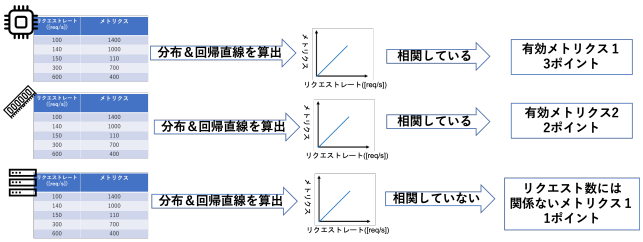

図 4　リクエストレートごとにメトリクスを取得

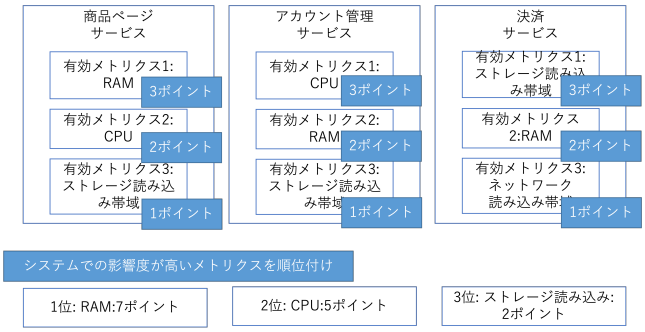

図 5　リクエストレートとメトリクスとの相関関係例:CPU の場合

マイクロサービス	優先度	メトリクス
商品	110	RAM
在庫	80	Storage
アカウント	20	CPU
履歴	30	Storage
ポイント	15	RAM
トレンド	10	RAM

メトリクス別, 優先度順に共助

マイクロサービス	優先度	メトリクス
アカウント	20	CPU

マイクロサービス	優先度	メトリクス
商品	110	RAM
ポイント	15	RAM
トレンド	10	RAM

マイクロサービス	優先度	メトリクス
在庫	80	Storage
履歴	30	Storage

図 6　リクエストレートとメトリクスとの相関関係例:CPU の場合

イクロサービスの影響メトリクスを中央の PoS Calc サーバーに保存する．PoS Calc サーバーでは，上記の手順に従って保存された各マイクロサービスのメトリクス順位を基にシステム全体でのメトリクス順位を策定する．

3.4　異なる 2 つのマイクロサービスの類似度計測

マイクロサービスにおける優先順位の算出では，先述した優先順位のうち共助できるコンテナを探すために異なる 2 つのマイクロサービスの類似点をコンテナ内にコピーされたプログラムのコードをコンテナのレイヤを用いて比較する．ここでの目的は 2 つのマイクロサービスを構成するコンテナを比較することでプログラムが互換している部分に関しての処理を代行が可能であるかどうかを判断できるようにすることである．

コンテナレイヤの比較

異なるサービスにおいてコンテナイメージが同一の場合，コードやプログラムの共有なしで処理が代行できる．それぞれのコンテナには ID が振られている．コンテナ ID が全ての文字列において同一の場合は処理を代行できる．しかし実運用環境において，コンテナは開発者によって独自のソースコードやプログラムを実行するため，ID が全てにおいて同じであることは少ないと言える．ここでコンテナを構成しているレイヤ (以降コンテナレイヤと呼ぶ) に着目する．コンテナレイヤは一つのコンテナイメージの読み込み専用のイメージ部分と，読み書きが可能なイメージ部分を分けることで OS，カーネルの部分を共通化し，開発ユーザー独自のソースコードがイメージに上書きできるようになっている．開発ユーザー独自のソースコードを実行したコマンドを比較することでより類似度の精度を向上することができる．以下に例を示す．ここでは私が開発した論文を検索するためのクラウドアプリケーションでの例を示す．今回着目するのは以下に示す 3 つの異なるサービスである．

- /api/upload アップロードされた論文ファイルをデータベース及びオブジェクトストレージに保存するマイクロサービス．Python により処理されている．API で受け取ったファイルを登録する処理内容で構成されている．単純な機能なため，必要となるリソースは少ない．
- /api/textize アップロードされた論文ファイルからタイトル，アブストラクト，本文をテキストファイルに抽出するマイクロサービス．ファイルを解析するため upload サービスと比較すると処理に必要なリソースは多い．
- /api/pdf upload サービスで外部のオブジェクトストレージに保存された論文ファイルを一覧表示する WEB アプリケーションで構成されているマイクロサービス．

ここではこの 3 つがどのようにコマンドを比較し，類似度の精度を向上するか説明する．以下の図 2 では textize サービスと upload サービスでそれぞれ実行されたコードを history コマンドで比較した時，これらは全て一致したと仮定する．すなわちプログラムに使用されるコマンドは全て一致した事になる．そのためこの 2 つのコンテナはソースコード及びプログラムを共有することで実行できる可能性，すなわち類似度が高いと言える．

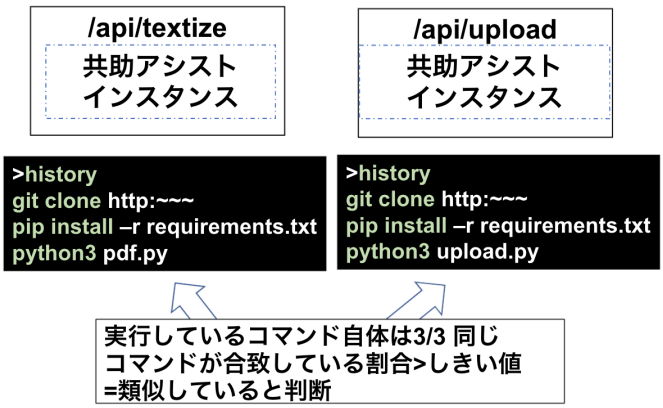

図 7　共助できる例

以下の図 3 では textize サービスと pdf サービスでそれぞれ実行されたコードを history コマンドで比較した時，1 つしかコマンドがなかった場合にはこの二つのコンテナで同じプログラムを実行することは難しいといえる．

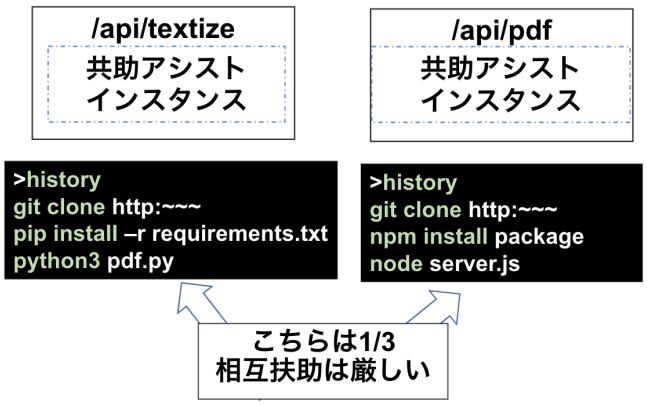

図 8 共助できない例

3.5 ソースコードもしくはプログラムの共有による他マイクロサービスプログラムの実行

ユーザー独自のソースコード部分以外のレイヤが同一であるならば，ソースコードやプログラムを共有すれば実行できる可能性が高いと言える．ソースコードやプログラムの共有手法については次章で述べる．

また，ユーザー独自のソースコード部分以外のレイヤが同一であった場合，より類似度の精度を高めるための検証をする．ユーザー独自のソースコード部分プログラムのソースファイルをテキストベースで比較する．

4. 実装と実験方法

実装

以下の 3 つのソフトウェアを実装する．

4.1 各サービスの実装詳細

4.2 Kubernetes クラスターの構築

実験環境として実際のクラウドサービスとして論文のポータルサイトを作成した．実験ではこのサービスに共助アシストインスタンスシステムを導入する．

本提案を利用する環境の前提としてサービスメッシュを用いているものとする．マイクロサービスでのトラフィックは全て Envoy を経由することで監視し, datadog を用いてメトリクスを取得できるものとする．

これらのサービスの構成表を以下の表 2 に示す．これらは以下のマイクロサービス (図内では μS と表記) で構成される．

以上の構成を元に作成した実験構成図を以下の図 3 に示す．

サービス名	役割
Web サーバー	ユーザーは WEB から登録, 検索をする
API サーバー	直接バックエンドサービスにリクエストを転送する
アップロードサービス	論文の登録を管理する μS
検索サービス	PDF をタイトルによって検索する$ μS
論文ファイル textize	PDF のファイルから文字を抽出する$ μS
DB 共通サービス	DB へのアクセスを管理する μS
タイトル/著者用 DB	タイトルと著者を保存するデータベース
タイトル/PDF DB	タイトルと PDF を保存するデータベース

表 1 実装構成表

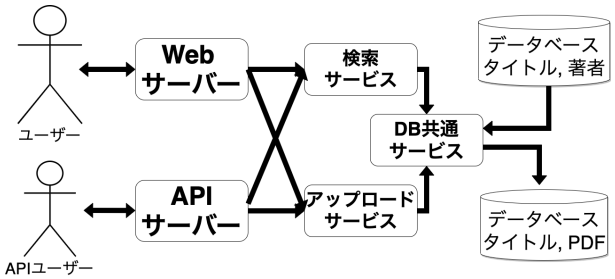

図 9 実験概要図

4.3 Kubernetes クラスターの構築

はじめに, ノード VM を作成するための仮想環境として表 3 のマシンに ESXi をインストールする．次に Kubernetes クラスターを構築するため, 以下の表 3 に示すスペックの VM を 2 ノード構築した OS はデフォルトで Kubernetes,Docker モジュールがプリインストールされている microk8s を使用した．

CPU	AMD Ryzen 3950X
RAM	128GB
SSD	NVMe 2TB
NIC	Intel 1Gbps

表 2 ESXi サーバーのスペック

CPU	4vCPU
RAM	16GB
Storage	200GB
OS	Ubuntu 20.04

表 3 各ノードのスペック

4.4 各サービスの実装

実験は Kubernetes システム上で以下のアーキテクチャで行った.

- Python/Locust: ユーザーと API ユーザーの代用として自動でリクエストを送信するツール
- Flask(Scalable max2):Web サーバー
- Flask(Scalable max3): アップロードサービス
- Flask(Scalable max3): 検索サービス

各サービスごとのレプリカを設定, ロードバランスをする Kubernetes Service をデプロイし, 各 Kubernetes Service にアクセスする.

5. 評価手法と分析手法

評価として, マイクロサービスが 15 個稼働している環境に API を用意しアクセスし, ユーザーが優先順位付けしたサービスと. また, 上記の順位付けを用いてノード内で CPU が効率的に割り当てられたかを確認する. ここではクラスターを構成している各ノードの計算リソースの使用量を以下のように求める.

今回の実験では恒常リクエストレートを 100 [req/s] とし，ピークを想定したリクエストレートを 500 [req/s] とした．また対象リクエストは Web サーバーに対する GET リクエストに制限した．実験時間はノードの使用率を算出する実験は 180 秒，サービスに対し初めてリクエストを送った際の不要なスパイクを実験から除くため，100 [req/s] を 60 秒間送信し，安定した状態から実験を行った．60 秒でサービスのレスポンスタイムが安定してから，すなわち計測開始から 60 秒間は継続して 100 [req/s] を送信した．60 秒後，リクエストの増加をシミュレーションするため，リクエストレートを 500 [req/s] に上昇させた．各サービスは各ノードに 1Pod のみ実行できるものとし，スケールアウトする際には別ノードの拡張が必要とした．また，オートスケールのしきい値は 80%とした．また，計測のばらつきを抑えるため，試行回数は 50 回とした.

$$\text{CPU 効率} = \frac{\text{ノードで使用されている CPU 量}}{\text{ノードに搭載されている CPU 量}} \quad (1)$$

また，API からのレスポンスタイムを本研究モデルの導入前後で比較する．その際に PoS が高いサービスはリクエストが急増しても従来のクラスターよりもレスポンスタイムが増加しないことが想定される．この実験の際に PoS が低いサービスでのレスポンスも確認し，考察する．また，クラスターの利用効率が上がることによるノードの追加頻度の減少を確認する.

6. 評価

以下に上記に示した実験の実行結果及び評価を示す.

6.1 PoS システムを導入したときのノードの効率的な利用

図 7 は実験を行った際の初めから用意しておいたノードにおける CPU 使用率を示したものである．青色の線が Kubernetes の純正のオートスケール，赤色が本研究の PoS システムを導入した結果のノードの CPU 使用率である.

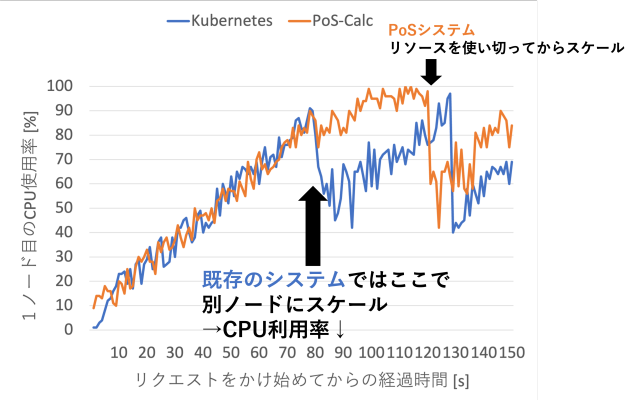

図 10 Relationship with CPU and PoS

評価開始から 60 秒間は 100 [req/s] でリクエストがサービスに到達しており，60 秒まではほとんど 2 つのシステムに差はないといえる．その後 Kubernetes のオートスケール機能が動作したことにより，70 秒時点ほどで大きく CPU 使用率が低下している．ここでの CPU 使用率の低下は，Kubernetes により別ノードに同一サービスが展開され，負荷分散したことによる．70 秒時点では CPU 利用率を 10% 程度残した段階でのスケールであるため，ノードの資源を有効に使い切れていないことがわかる．これと比較し，PoS システムを導入した結果，70 秒時点では元のサービスでは処理しきれなくなったため，類似コンテナに負荷分散したことにより，CPU の使用率を極限まで使用してからスケールを行っている．この点に関して本研究の提案する PoS システムは Kubernetes のオートスケーラにくらべ, 約 40 秒ノードの高効率な利用に貢献しているといえる.

しかし，130 秒のあとの CPU 使用率はスケールをしているのにも関わらず CPU 使用率が Kubernetes のオートスケーラほどに低下していない．これは PoS システムが極限までシステムのリソースを使い切った結果，別ノードにスケールしたときと Kubernetes が余裕を持ってスケールを行った際では，システムの受けているリクエストの数が前者の方が多いからである.

6.2 PoS システムを導入したときのレスポンスタイムの維持

図 8 はリクエスト開始からリクエストレートの増加を行った際の Kubernetes のオートスケーラ，または PoS システムの共助プロセスが開始された際のレスポンスタイム

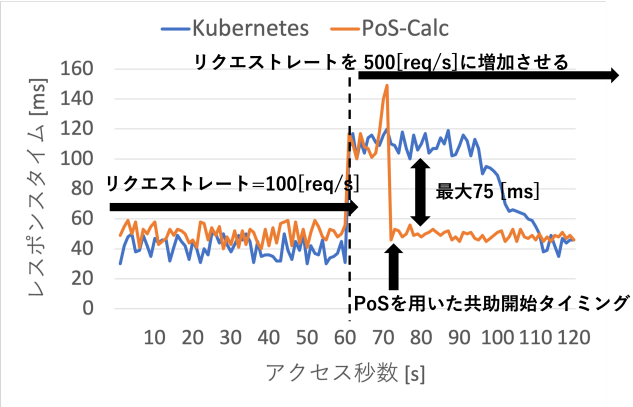

図 11 PoS システムを用いたときのレスポンスタイムの維持

を示している．60 秒を超えた段階でリクエストレートが 100 [req/s] から 500 [req/s] に増加したことにより，両システムともに一時的にレスポンスタイムが飛躍的に増加している．しかしその後 PoS システムを用いたときは 10 秒ほどでサービス間の共助により負荷分散され，レスポンスタイムがリクエストレートが 100 [req/s] の時に近づくように減少したのに対して，Kubernetes のオートスケーラでは新たなノードにサービスを展開するまでのタイムラグから PoS システムが共助開始したタイミングから 40 秒ほど最大 75 [ms] 差が見られる．

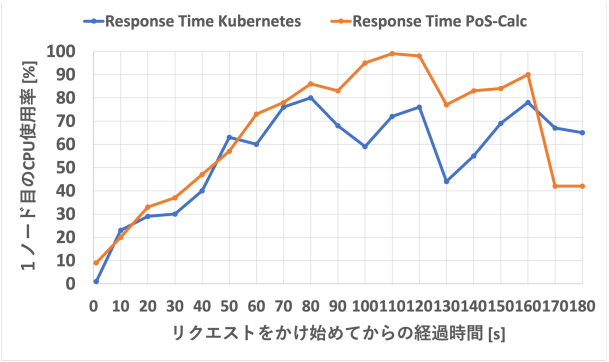

図 12 試行回数を 1000 回にした際の CPU 使用率の推移

7. 議論

PoS の算出方法ではトラフィック量と他サービス及び外部からの接続数を 1 つの指標として利用した．一方で単純なトラフィック量と接続数のみで判断すべきでないリソースも存在する．ここでは現状使用している指標以外で判断すべき 1 つの例としてログについて説明する．コンピュータシステムにおいて，ここではクラウド環境上において障害管理の観点からログを取得することは重要とされている [9]. そのためクラウド環境上では多数のログを取得するサービス (以降，ログサービス) が実行されている [10]．しかし，PoS にサービスにおける優先順位の内にログサービスを入れることは望ましくない．ログサービスはクラウドアプリケーションのアプリケーション開発時及びエラー時のデバッグや停電，過電流や CPU ストレージの破損やハードウェアの故障による復旧の際に極めて重要な情報群であるからだ．そのため，ログサービスに関わるリソースの効率化やレスポンスタイムの向上は，クラウドアプリケーションとは別に独立した形でするべきである．しかし，これを実現するためにはクラウドアプリケーションとログサービスの自動検出及び判別，そしてグルーピングが必要である．また，ログはアプリケーションと深く関わっているため，単純な方法ではプログラムを判別できない可能性がある．例えば，fluentd や Elasticsearch といったログエージェント，ログ検索エンジンはトラフィックの内容やコンテナに用いられるイメージ名からルールベースで検出することが可能である．しかし，クラウドアプリケーションを作成している企業が独自のログサービスを構築した場合，ルールベースでは検出が難しい．そのため，サービス間での通信のうち，日付や共通の単語が多い時にはログと判断し，アプリケーション側の PoS の対象外，もしくは別の基準の採用を利用する．これにより，クラウドアプリケーション側の PoS はログを排除したものになるため，より PoS の検出の精度が上昇する．

また，ユーザーの任意で PoS の計算より除外できる仕組みが必要である．構築ユーザーがセキュリティの観点といった意図的に PoS の計算より除外したいサービスが発生することが想定される．そのため，以下のような構築ユーザー側で明示的に PoS の動作を制御するコードを導入する．以下のソースコード 1 に示す．

ソースコード 1 明示的に PoS システムへの参加を拒否するコード

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: pdf-textize
5    pos: False #POS による自動優先順位の決定を許可
6    beHelper: False # True にすることで他サービスが
7    #このサービスのリソースを使うことを許可
8  spec:
9    type: NodePort
10   ports:
11   - port: 80
12     targetPort: 5000
13     name: http
14   selector:
15     app: flask
```

実際のトラフィックでの測定として，評価方法としてクラウドアプリケーションを提供している開発者と PoS 計測システムが出力するサービスの順位付けを比較した．また，レスポンスタイムを用いてクラウドアプリケーションのレスポンス高速化を評価した．しかし，実際にマイクロサービスを用いたクラウドアプリケーションでは 500 以上のマ

イクロサービスを使用することもある [11]. 今回の実験の 50 倍以上のマイクロサービスが使用されている環境での評価, 及び問題点の抽出が必要である. しかし実際に顧客が存在していて, SLA が定義されていた場合にはこのような研究のための実験は困難である. そのため trainticket のようなマイクロサービス用のベンチマークソフトウェアを拡張し, 研究分野でも 100 以上のマイクロサービスでテストできる環境を開発する必要がある.

8. おわりに

従来の研究及びクラウドアプリケーションにおけるマイクロサービスでの導入時にはサービスの優先度は人間が事前定義する必要があった. そのため, ノードのリソースが枯渇した際, どのマイクロサービスを優先的に処理するかを動的に策定する手法が存在していなかった. この研究では自動的に稼働中のマイクロサービスより優先度を算出するメカニズムを提案した. これによりコンテナの類似度計算及びプログラム共有を行うことでクラウドを構築するノードのリソース (今回の実験では CPU) を効率的に利用することができた.

参考文献

[1] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L.: Microservices: yesterday, today, and tomorrow, *Present and ulterior software engineering*, pp. 195–216 (2017).

[2] Lehrig, S., Eikerling, H. and Becker, S.: Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics, *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, pp. 83–92 (2015).

[3] Keshanchi, B., Souri, A. and Navimipour, N. J.: An improved genetic algorithm for task scheduling in the cloud environments using the priority queues: formal verification, simulation, and statistical testing, *Journal of Systems and Software*, Vol. 124, pp. 1–21 (2017).

[4] Chen, H., Wang, F., Helian, N. and Akanmu, G.: User-priority guided Min-Min scheduling algorithm for load balancing in cloud computing, *2013 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, pp. 1–8 (online), DOI: 10.1109/ParCompTech.2013.6621389 (2013).

[5] Alipour, H. and Liu, Y.: Online machine learning for cloud resource provisioning of microservice backend systems, *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2433–2441 (online), DOI: 10.1109/BigData.2017.8258201 (2017).

[6] Calzarossa, M. C. and Massari, L.: Analysis of Header Usage Patterns of HTTP Request Messages, *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, pp. 847–853 (online), DOI: 10.1109/HPCC.2014.146 (2014).

[7] Xin, J., Afrasiabi, C., Lelong, S., Adesara, J., Tsueng, G., Su, A. I. and Wu, C.: Cross-linking BioThings APIs through JSON-LD to facilitate knowledge exploration, *BMC bioinformatics*, Vol. 19, No. 1, pp. 1–7 (2018).

[8] Risso, F., Baldi, M., Morandi, O., Baldini, A. and Monclus, P.: Lightweight, Payload-Based Traffic Classification: An Experimental Evaluation, *2008 IEEE International Conference on Communications*, pp. 5869–5875 (online), DOI: 10.1109/ICC.2008.1097 (2008).

[9] Chuvakin, A., Schmidt, K. and Phillips, C.: *Logging and log management: the authoritative guide to understanding the concepts surrounding logging and log management*, Newnes (2012).

[10] Khan, S., Gani, A., Wahab, A. W. A., Bagiwa, M. A., Shiraz, M., Khan, S. U., Buyya, R. and Zomaya, A. Y.: Cloud log forensics: foundations, state of the art, and future directions, *ACM Computing Surveys (CSUR)*, Vol. 49, No. 1, pp. 1–42 (2016).

[11] 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).