Podの処理能力の計測を連携させたKubernetes構成の自動生成

伊藤 佳城[1,a)]　串田 高幸[1]

概要：Kubernetes(以下 K8s) はマニフェストの記述をベストプラクティスにするためには, Pod の CPU, Memory 使用量の limits(上限)，requests(下限) の設定が必要となる．しかし，Pod のコンテナが使用する CPU, Memory 使用量は実際に起動して動かしてみないとわからないという課題がある．そのため, 想定されたリクエストを処理した時に CPU, Memory 使用量がどのくらい必要となるのかを知ることが必要となる．この問題を解決するために, ロードテストの自動化およびマニフェストの自動化を行うことで解決を図る．これによって運用環境で動かしたいコンテナを運用環境で動かす前にリクエスト数の処理に必要な CPU, Memory 使用量を提示することが可能となる．

1. はじめに

背景

コンテナ仮想化の利用者増加には, マイクロサービスの普及が関連している．マイクロサービスは, サービスごとに異なる OS やバージョンの組み合わせで構成されている可能性があるため．それぞれのサービスをコンテナ化することでサービスごとにアプリケーションと必要なライブラリをパッケージ化することができ, 環境に依存せず展開し実行することができる [1]．しかし, コンテナは手軽に開発・実行環境を構築できるが, 管理対象のコンテナの数が増えると, その運用・管理が複雑で手間がかかるという課題がある．そこでこれらの課題を解決するために開発されたのがコンテナオーケストレーションシステムである．コンテナオーケストレーションを活用することで複数のコンテナの統合的な管理が容易になる．コンテナオーケストレーションシステムでは, 代表的なものとして Kubernetes(以下 K8s), Docker Swarm, Mesosphere が挙げられる．　この中で最も広く使用されているコンテナオーケストレーションシステムは，K8s である [2]．

K8s には, 利用するためのベストプラクティスが存在する公式の設定のベストプラクティスが存在し, いくつか考慮するべき点がある [3]．その中でも大きく関係するのが Pod や node に関する K8s 全体の CPU, Memory に関連するものが複数挙げられる．コンテナのイメージのサイズを小さくするべきというものがあるがこれは K8s における不要な CPU, Memory 使用量を避けるために存在する要項である．また同じように K8s の namespace を活用するというものがあるがこれも CPU, Memory を効率的に使用するための方法である．そして直接的なのがコンテナリソース（CPU, Memory）に対する requests と limits を設定する点である．CPU, Memory の制限がないと Pod(コンテナ) が不安定になり、終了する可能性がある点や不適切なサイズのリクエストは、スケジューリングの問題を引き起こす可能性がある危険性がある．

K8s には，Horizontal Pod Autoscaler(HPA) と Vertical Pod Autoscaler(VPA) という CPU, Memory 管理におけるオートスケーリングの機能が備わっている [4]．HPA は，現在の Pod の CPU 使用率の合計からユーザが設定した CPU 使用率になるように Pod の数をスケールさせて使用率の分散を行う．しかしメモリの観点からみると Pod の数がスケールされるとメモリの消費量は増加する．なのでこの方法では Memory 使用率は考慮されない．最終的な数値の設定やスケール数は結局，どのような image の Pod を動かすかやデプロイ後の挙動を知らなくてはオートスケールできても CPU, Memory を効率的に使用できていない．VPA は，CPU，Memory の使用状況からリクエスト値を自動で算出して Pod 作成時に付与する方法が取られている．この場合は Pod を再起動させて設定を変更とするため反映まで数分のブランクが存在すること現状ではリージョンクラスタでしか利用できないといった制約が存在する．

ユースケースとして K8s を使用してマイクロサービスである EC サイトを提供するユースケースを考える．その図

[1] 東京工科大学コンピュータサイエンス学部
〒 192–0982 東京都八王子市片倉町 1404-1
[a)] C0117039

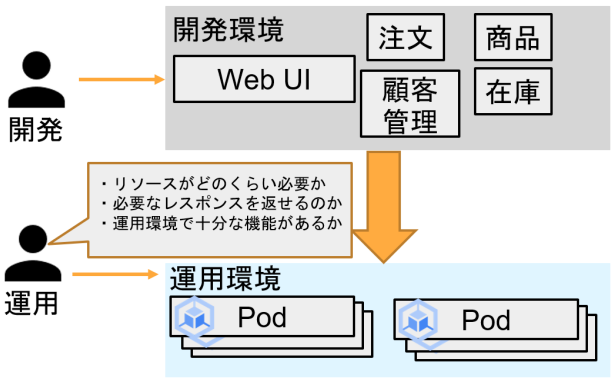

図 1　ユースケース図

を 1 に示す．想定されるターゲットのユーザとして，K8s を使用して EC サイトの構築を行っていることを想定する．その時の問題点として EC サイトの機能が運用環境では全て動かないケースを例に挙げる．EC サイトが 5 つのコンポーネントで構成されていることを考える．この際に想定される問題として運用する人物・環境とサイトを開発する人物・環境は異なることがある．その場合，開発されたものが運用環境で動くのか，また自社開発された EC サイトが一体どの程度のアクセス数，リクエスト数で運用環境での CPU, Memory 使用量が必要となるのか各種機能がどの程度のレスポンスを返すのかを算出しなくてはならない．その場合，運用環境で想定されるアクセス数，リクエスト数でテストを行うことで各種機能の必要 CPU, Memory 使用量を算出する．機能ごとに使われるソフトウェアは異なるが自動化による負荷テストから自動デプロイを行うことで実際の構築及び運用での運用手順を減らすことができる．

課題

CPU, Memory 制限を実際に行う際には, YAML 形式や JSON 形式で記述したマニフェストを用いてデプロイする Pod やその CPU, Memory の指定・管理を行う．しかし，運用する Pod の量が増加するとその分のマニフェストの総数が増えその分の管理の工程も増加する．その大量の YAML はよく wall of YAML（YAML の壁）と揶揄される．それらの問題を解決するツールはある (Helm, Kustomize)．Helm は，K8s のパッケージマネージャである．設定ファイルを元にアプリケーションのデプロイを自動実行するツールであり複雑なマニフェスト管理がアプリケーションごとにまとまっている為, 管理が容易となる．しかし, 上記の Helm を使用したとしても実際の運用環境と実用環境は異なるため上記のような CPU, Memory 管理までは考慮されていない．また，ベストプラクティスとして CPU, Memory 制限を行うべきとあるが実際にどのように CPU, Memory 制限を行うべきかの指標がなく, 結果的に設定していないケースが課題として挙げられる．

上記の要素を初心者やユーザが常に考慮しながら運用するのは, 使用するユーザに大きな負荷となる．その為, 今回の実験及び実装では, 上記の部分の構成に関するベストプラクティスを解決するマニフェストの自動生成を行えるプログラムの作成を成果の 1 つとして提案する．

実際に Helm を使用して Pod をデプロイした際にそのステータスが Pending になってしまうことがある．そのようなことが起こる理由は，製作者の設定した環境と自身の環境は異なっていることが考えられる．ノードの空き容量が足りない, port の空きがない, CPU, Memory 設定が適切でない requests の要求量が適切でない理由が考えられる．そのため，運用時の環境でテストを行い適切な設定を行わなければならない．

しかし，そのためにテストを行うとしても K8s における負荷テストは，負荷分散テストに焦点が当てられていることがほとんどであり，現状の負荷テストでは，リクエストをさばききることができるか，Pod がオートスケールできるか否か, ノードがスケールされるかがテストされる．Pod 単体の CPU, Memory 使用量を図るようには使用されていないことが現状である．しかし背景でも述べた通りベストプラクティスには CPU, Memory 制限を付けるべきとある．

ノード内で Pod が使用可能なリソース量が以下のような場合を想定して CPU, Memory 使用量を変更した場合を考える．requests，limits を同じにした Pod を 10 個デプロイ CPU: 300m Memory: 75Mi したところ 10 個中 5 個が起動したが 5 個は Pending となってしまった．これは，Pod の requests の設定ミスによる Pod のスケジューリングミスである．

各章の概要

この論文は, 次のように構成される．1 章では, コンテナ仮想化及びコンテナオーケストレーションシステムの背景・課題．2 章は, 関連研究の説明．3 章では, 本研究の提案に就て述べ.4 章で実装とその実験環境,5 章で評価と分析を行い, 6 章で今後の課題や議論・考察を行う．最後に 7 章で, 最終的なまとめで締めくくる．

2.　関連研究

この章では本研究と関連した関連研究について取り上げていく．

まず自動化という面では, ビルド, テスト, デプロイを自動化・継続的に行う手法である CI/CD を DevOps で用いる際の流れでどのように自動化をするべきかが述べている．品質におけるパイプラインが自動化することができれば, 迅速で一貫性のあるリリースができると述べられている [5]．もう一つは, ソフトウェア開発者がハードウェアコンテナを構成するセキュリティマニフェストを作成するた

めに, 自動化ツールがどのように役立つかを議論し自動化ツールの提案を行っている [6]. IaaS クラウドのキャパシティプランニングに対応するため, 総所有コストを最小化・経済的な物理マシンという 2 つのコスト最小化問題を検討しており, 確率的探索アルゴリズムであるシミュレーテッドアニーリングを使用を利用することで最適解を求めている. ここでは, アルゴリズムを使用することによりそのようなコストが最適化を述べている [7]. もう一つの最適化の手法を用いた論文では, ベイズ最適化を用いて複数のカテゴリのクラウドワークロードを対象に新しいフレームワークを提供している. その結果ワークロードの性能チューニングの向上が見込まれた [8]. コンテナのシステムリソースに関する論文では, 挙動を監視するためコンテナの一連の評価を行っており. 手動にデプロイされたクラスタとの比較を行っている. 比較対象を手動と自動化に絞ることで評価を行っている [9]. 自動化する部分を K8s をベースとした動的リソースプロビジョニングに焦点を当てて, それらを容易にするための汎用的なプラットフォームを開発することを目的とした論文である. 他の手動設定なしで, ユーザーが定義した時間間隔に従って, 実行中のすべてのアプリケーションに適用されている [10]. この論文では VM 構成プロセスを自動化の提供を行っているがその手法に強化学習 (RL) ベースを用いている. 結果として, システム管理に RL を適用する際のスケーラビリティと適応性の問題に対処した ?. コンテナオーケストレーションシステムにおける配置に焦点を当てた論文では, Docker Swarm を使った動的な異種クラスタにおける Docker コンテナの配置を行いシステム性能を向上させるためのリソースを考慮した配置スキームである DRAPS を開発し, 物理ノードと仮想化コンテナの両方の異質性を考慮できるような開発を行っている [11]. 同じく配置に焦点を当てたこの論文は, コンテナ化されたアプリケーションの実行時間と初期化時間の両方のパフォーマンスを向上させるためのアプローチを行っている [12].

K8s のマニフェストとして用いられる YAML は，ワークロードの仕様には適しておらず間違ったインデントが発生する．そのために開発された DSL についての研究であるが記述形式が YAML から変更になり，新たな DSL に対する学習コストが発生している [13]．本研究では既存の手順を自動化するため新たなコストは発生しない．

XML, YAML, 属性指向プログラミングだけでは，ユーザの求める構成システムを表現できない部分に焦点を当てられたこの研究では，複数の設定言語を使用するがそれに伴うユーザの使用リスクの増加やシステムの実装・保守コストが高くなる問題がある?．本研究では，YAML の設定を自動化で行うためこのような問題は発生しない．

K8s に基づく動的なリソースプロビジョニングを容易にする汎用プラットフォームの開発を目的としたこの研究では，既存のリソースプロビジョニングに加え動的な管理と監視対象をリソース使用率と QoS メトリックとしている [10]．しかし，この研究では Pod に対する負荷があった場合のリソース量の判別は，デプロイ時の情報を元に行われるため，想定して設定を構築するわけではない．

ペトリネットベースのパフォーマンスモデルによって達成された K8s のパフォーマンスを分析するこの研究では，コンテナのライフサイクルの流れを分析することで弾力性のあるサポートの実現を行っている?.

上記のように数値の最適化には, アルゴリズムが用いられる傾向にあり, 自動化では, 開発者やユーザ視点での問題解決に勤しんでいる. コンテナによるリソースやパフォーマンスでは, 起動時にどこに設置するかという部分に焦点が置かれるため実際のリソースの必要量が考慮されることはない.

3. 提案

提案では, ユーザが指定するのはテストとデプロイしたいイメージを指定するだけで以降の手順は全てプログラムによる自動化を目指す. 自動化することで処理されるリクエスト数から Pod に必要な CPU, Memory の使用量を構築に含めることで, リソース制御の手助けが行えるようにする. 提案するアーキテクチャ図を 2 に示す.

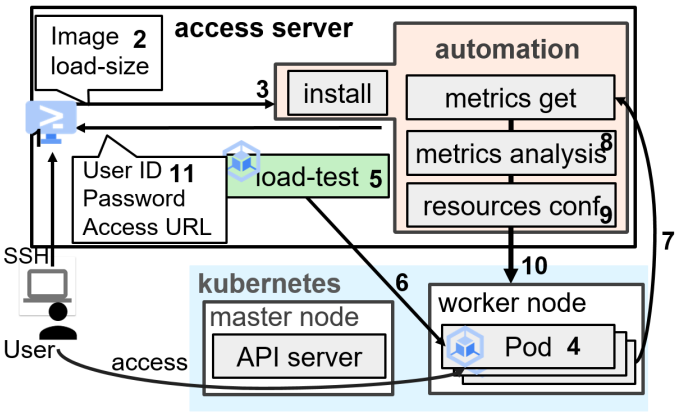

図 2　全体アーキテクチャ図

2 における提案の図の説明の流れを下記に示す.

(1) User が自動化プログラムを起動

(2) User に image を選択制で提供→選択

- User がコンソールを使用して access server にログインする．そこで自動化のプログラムである automation を起動することでテストしたい Imege と load-size の指定を行う.

(3) 環境構築に必要なソフトウェアをインストール

- 環境設定に必要なソフトウェアのインストール，構築を自動で行う.

(4) 選択された Image をデプロイ

- 1 で選択された Image の Pod をデプロイする.

(5) load-test を構築
- 負荷テストに必要な設定ファイルを読み込み，load-test を構築する．

(6) load-test で Pod に負荷テストを行う
- 入力された load-size を基にリクエストを送信する負荷テストの実行を行う．

(7) metrics get を使用して負荷テスト中の Pod のメトリクスを収集
- デプロイされた Pod に対して負荷テストを行ったメトリクスを収集する．この際収集したデータは CSV ファイルで保存する．

(8) 収集したデータを metrics analysis で分析
- 取得したデータを基に平均値，最大値，最小値，標準偏差を求め，最大値を limits に設定する．

(9) 分析データを基に resource conf で Pod リソースの limits, requests を設定
- リソースの requests, limits の設定を行う．そしてその設定を YAML ファイルのマニフェストに記入する．

(10)生成した Pod を実際にデプロイして終了
- 設定を変更した Pod を更新してデプロイする．

(11)Pod のアクセスに必要な情報を提供を受け取る
- デプロイを行った Pod にアクセスするために必要な情報である User ID, Password, Access URL を User に出力し，User は与えられた情報を基に Pod にアクセスする．

今回の提案では，実際にデプロイをして本番環境で実用する前に実装環境で負荷テストを行いその結果に応じた設定を Pod に入れ込むことで実際のデプロイ時にリソース不足やデプロイすらできない状況を無くし，手動による設定コストも削減する．

4. 実装と実験環境

4.1 実装

実装におけるソフトウェアの構成図を図 3 に示す．

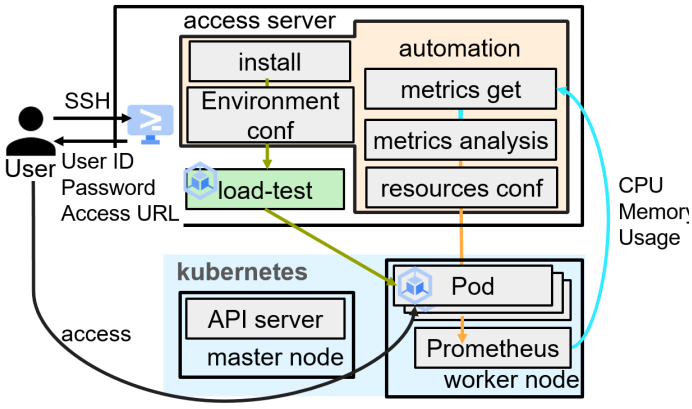

図 3 ソフトウェア構成図

今回の提案における実装のプログラムを下記に示す．

- 自動化プログラム (automation)
  - 自動インストールプログラム (install)
  - Pod メトリクス取得プログラム (metrics get)
  - Pod メトリクス分析プログラム (metrics analysis)
  - CPU, Memory 設定プログラム (resource conf)

個別のプログラムを一連の流れとして動作させるのが自動化プログラムである automation であり，python の subprosess を用いて基本的にコマンドで呼び出している．Helm を使用し K8s 上に一括でアプリケーションのデプロイを行う為，install にて Helm のダウンロードを行う．また，load-test を行うコンテナは，Docker,Docker-Compose でデプロイするためそれらのインストールも行う．install したものを使用してテストツールを含めたコンテナのデプロイを行い環境構築を行う．load-test を行った Pod のメトリクスを Prometheus を使用し，データの収集を行う．metrics get では，収集したデータを CSV ファイル形式で保存，metrics analysis でテストデータの値を平均値，最大値，最小値，標準偏差を求める．そして取得したデータを元に resouces conf で CPU, Memory 制限の値を Helm の value.yaml に記述する．

4.2 実験環境

今回の研究では，研究室内に構築した K3s 環境を使用する．3s 環境は，OS が Ubuntu18.04 の VM で構成されており master ノード 1 台，worker ノード 2 台の計 3 台で構築されている．下記にその K3s の基本構成を示す．

- vCPU x 2
- RAM 4GB
- HDD 50GB

さらに master にアクセスするクライアントを 1 台を構築し，そこから作業を行う．実装したプログラムは，アクセス用に構築した Ubuntu18.04 サーバに設置を行う．その次に Helm を使い，負荷を掛ける pod のデプロイを行う．負荷テストに用いる Pod には，WordPress を使用する．負荷テストに用いるテストツールとして Locust を使用する．

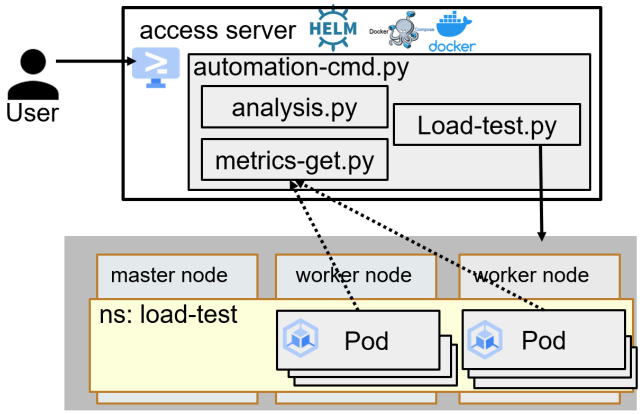

図 4 実験環境の図

5. 評価と分析

評価には，条件として，検証用の namespace である load-test を作成し，Pod のメトリクスを K8s の api を使用したプログラムである metrics-get を使用して取得する．負荷テストをリソース変更以外は同じ条件で行うことを前提とし，デプロイする Pod の Limits, Requests を指定した後の Pod を config Pod, 変更を加えていない Pod(デフォルト値) を default Pod として両者の比較を行うことで Pod に対する設定が有意義かどうかの評価を行う．

CPU, Memory 制限の有無で同一リクエスト数を処理した場合の Node の CPU, Memory 使用量使用率を図る．もう一つは Pod 自体の CPU, Memory 使用量使用率を図ることで 2 つの Pod の差分でどちらが少ない使用率で稼働でき，処理できているかを 1 つの評価とする．もう一つは，標準偏差を求めることで Pod の CPU, Memory 使用量 使用率のデータのばらつきを評価する．そうすることで Pod の安定性の観点から評価を行う．

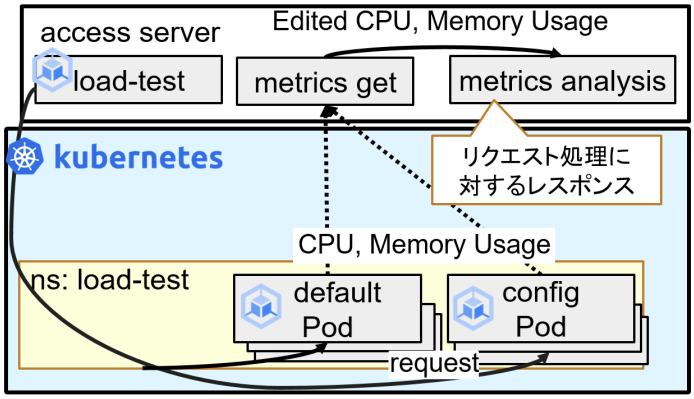

図 5　実験構成の図

評価時のグラフでは，X 軸: テスト経過時間 Y 軸: メトリクス使用率 (CPU, memory 使用率), として評価を行う．

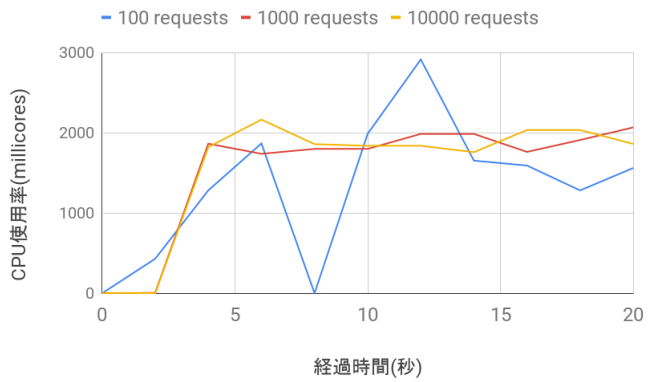

図 6　負荷テスト時の CPU 使用率

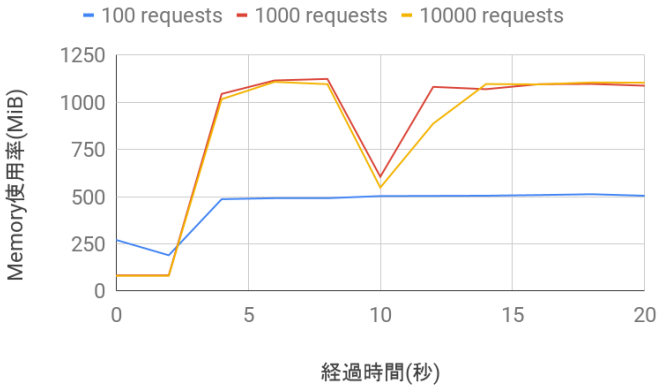

図 7　負荷テスト時の Memory 使用率

6. 議論

本論文では, Pod の CPU, Memory 制限を行う自動化について議論を行った．リクエスト数から Pod の処理能力の判別には, 負荷テストを行い，その時の CPU, Memory 使用量のデータからの平均値, 最大値，最小値からだと算出できたとは言えない．また，制限を加えた Pod が制限なしの Pod よりもスループットが同じになるとは言えないため更なる算出方法を適応する必要がある. その方法として limits の設定方法として様々なアルゴリズムを適応し，適切な値を算出する．例としてベイズ最適化の適応方法がある．負荷テストにおけるリクエストの処理応答時間に対しての必要な CPU, Memory の最大値を求める．負荷テストを一定時間行った際の CPU, Memory 使用率を取得し，サンプリング化を行う．それを基に Pod のライフサイクル全体の CPU, Memory 使用量を予測する．もう一つは，制約付き最適化として Pod の CPU, Memory 使用率を判別する方法である．この場合の制約は, Pod のデプロイ先であるノードの使用できる CPU, Memory 量を制約条件として負荷テストを行ったときの Pod の必要 CPU, Memory 量から最小設定数を算出する．

もう一つの議論としてデプロイする Pod に対して負荷テストを行ったがその負荷テストを行う request 値をどのように設定するかも今後の課題として残っている．例として 1 日に 10 アクセス数の場合と 1 秒に 10 アクセス数という 2 つの場合があった時，この 2 つのパターンには大きな差が出ている．1 日に数回のアクセス数ならば小さいサイズの CPU, Memory に設定しておくことが可能であるが 1 秒に複数アクセスある場合は，その状況に近い負荷テストを行いそれに伴うような設定が必要となる．どのように判別を行うのか，動いているもしくはすでに動かしている Pod の場合は，Pod のログを参照して 1 日のアクセス数を設定してそれを想定した設定を構築することが必要となる．その際のログの取得間隔は，1 日とすることで曜日ごとのアクセスデータが収集でき過去のデータを参照し，日にちにあった CPU, Memory の割当ができる．しかし，新しく

Pod を動かす場合，今までに動かしたデータやログが無いため設定を加えずに 1 分間デプロイ，その時の使用率の推移に当てはまるパターンを当てはめる方法があるがどちらの場合も Pod のアクセスログから動的に CPU, Memory 使用量の判別を行うことで適切な request 数を実現できると考える．

7. おわりに

本論文では, K8s のマニフェストの自動化と負荷テストの自動化を行う提案と Pod のリソース制限を行う自動化について議論を行った．リクエスト数から pod の処理能力の判別には, 単純なデータからの平均値, 最大値，最小値からだと判別することは難しいため，一度に取得するデータ値を複数回にして試行回数を増やすことで負荷テストデータの整合性を取れるようにする必要がある．もう一つは，現在は Pod イメージを決め打ちにしてテストを行ってしまうため，ランダムなイメージでもテストを行えるような機能をベイズ最適化のブラックボックスアプローチを取り入れることで実現しようと考えている．自動化ツールで課題を解決することができれば, ユーザにとって負荷の減少が考えられ, 利便度の向上とベストプラクティスを内包でき, K8s におけるリソース管理の課題を解決することができる．今後の課題として取得データから yaml への書き起こしの算出方法の実装が卒業論文までの課題となる．

参考文献

[1] Jawarneh, I. M. A., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R. and Palopoli, A.: Container Orchestration Engines: A Thorough Functional and Performance Comparison, *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pp. 1–6 (online), DOI: 10.1109/ICC.2019.8762053 (2019).

[2] Pan, Y., Chen, I., Brasileiro, F., Jayaputera, G. and Sinnott, R.: A Performance Comparison of Cloud-Based Container Orchestration Tools, *2019 IEEE International Conference on Big Knowledge (ICBK)*, pp. 191–198 (online), DOI: 10.1109/ICBK.2019.00033 (2019).

[3] : Configuration Best Practices, `https://kubernetes.io/ja/docs/concepts/configuration/overview/`. plus.33emminus.07emK8s Documentation(2020 年 7 月 20 日閲覧).

[4] Balla, D., Simon, C. and Maliosz, M.: Adaptive scaling of Kubernetes pods, *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–5 (online), DOI: 10.1109/NOMS47738.2020.9110428 (2020).

[5] Virmani, M.: Understanding DevOps bridging the gap from continuous integration to continuous delivery, *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, pp. 78–82 (2015).

[6] Leontie, E., Bloom, G. and Simha, R.: Automation for creating and configuring security manifests for hardware containers, *2011 4th Symposium on Configuration Analytics and Automation (SAFECONFIG)*, pp. 1–2 (2011).

[7] Ghosh, R., Longo, F., Xia, R., Naik, V. K. and Trivedi, K. S.: Stochastic Model Driven Capacity Planning for an Infrastructure-as-a-Service Cloud, *IEEE Transactions on Services Computing*, Vol. 7, No. 4, pp. 667–680 (2014).

[8] Shi, Y., Peng, Z., Wang, R. and Bian, Z.: Adaptive Cloud Application Tuning with Enhanced Structural Bayesian Optimization, *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 1–8 (2019).

[9] Pereira Ferreira, A. and Sinnott, R.: A Performance Evaluation of Containers Running on Managed Kubernetes Services, *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 199–208 (2019).

[10] Chang, C., Yang, S., Yeh, E., Lin, P. and Jeng, J.: A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning, *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–6 (2017).

[11] Mao, Y., Oak, J., Pompili, A., Beer, D., Han, T. and Hu, P.: DRAPS: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster, *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8 (2017).

[12] Boza, E. F., Abad, C. L., Narayanan, S. P., Balasubramanian, B. and Jang, M.: A Case for Performance-Aware Deployment of Containers, *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, New York, NY, USA, Association for Computing Machinery, p. 25–30 (online), available from ⟨https://doi.org/10.1145/3366615.3368355⟩ (2019).

[13] Xu, C. and Ilyevskiy, D.: Isopod: An Expressive DSL for Kubernetes Configuration, *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, New York, NY, USA, Association for Computing Machinery, p. 483 (online), DOI: 10.1145/3357223.3365759 (2019).

