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) はマニュフェストの記述をベストプラクティスにするためには, 考慮する部分が多く難しいという点と pod のコンテナがどれだけの cpu, メモリを使うのかは, 実際に起動してみないとわからないという課題がある. その為, ロードテストでどの程度のリソースがかかるのかリクエストを処理できるかを知らないと予期せぬ動作の不具合を招くため, 事前に把握する必要がある. 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. これによって動作させたいコンテナを本番環境で動かしたいコンテナを実際の環境で動かす前に cpu, メモリ使用量から処理できるリクエスト数を提示することが可能となる.

1. はじめに

1.1 背景

コンテナ型仮想化は, コンテナエンジンを使うことでコンテナごとにアプリケーションを動かすことができる, 非常に軽量な仮想マシンのようなものである [1]. ハイパーバイザー型仮想化と比べると, OS の起動を不要とするため起動・処理速度が VM 型よりも優れていることがメリットとして挙げられる. コンテナ仮想化は, docker イメージ利用の環境構築による同環境で同じアプリケーションを実行することができる点, それによる別環境ごとの運用コストの削減, 異なる OS 環境の管理の簡単化, 異なる環境への迅速な対応が利点として挙げられる. 上記の利点による新たな仮想化の手段としてシステム構成におけるコンテナ利用が急速に進んでいる.

コンテナ仮想化は, システム構成における IT インフラの基盤としての活用がなされている. アプリケーションごとにコンテナを利用する機能単位の小さなアプリケーションを組み合わせてシステムを実現できるため, サービスを構成する各要素をコンテナで分ける事ができるため, マイクロサービスとの相性も良い. 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登場したのがコンテナオーケストレーションシステムである. オーケストレーションシステムには, サービスにおける設定・管理・配置の自動化を行う意味があり, 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というと, 代表的なものとして Kubernetes(以下 K8s), Docker Swarm, Mesosphere が挙げられる. その中で最も使われているのが K8s である. Docker Swarm は, コンテナ仮想技術の Docker に対応するクラスタリング用のツールであり K8s が普及する前は, こちらが多く使用されていた. Mesosphere では, コンテナのクラスタを指定することで仮想マシンのプロビジョニング及びコンテナのクラスタ運用まで自動的に行うプラットフォームとして登場していたが後述する K8s の需要増加による K8s へのサポートへ舵を切った. その為, オーケストレーションシステムとしては, K8s のサポートの普及で K8s の活用状況は増加傾向といえる状態である. K8s はコンテナオーケストレーションの業界標準となっており, Google 社内で利用されていたコンテナクラスタマネージャ Borg が基になって作成された OSS である [2]. K8s は, 大手クラウドプロバイダーやパブリッククラウドベンダが参加する Cloud Native Computing Foundation に参加し, 開発が進められた. K8s が大きく使われている背景として, このように大手のベンダーとの開発により GCP, AWS, Azure の大手クラウドサービスでもサービスとして提供されたことが現在利用される大きな理由の一つである.

1.2 課題

K8s では, YAML 形式や JSON 形式で記述したマニュ

[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

フェストを用いてデプロイする pod やそのリソースの指定・管理を行う．しかし, このマニフェストの記述方式はルールが存在しないため, ユーザー個人に記述は委ねることとなる．そのため, 記述された YAML や JSON のファイルは製作者によって異なってしまう. そうするとユーザごとに記述の仕方が変わり作成者以外のユーザが確認した際に正規化していないとわかりにくいという点がある．さらに, 公式の設定のベストプラクティスには, いくつか考慮するべき点が存在する [3].

- 構成を定義する際に, 最新の安定した API バージョンを指定
- マニュフェストは, クラスターに反映される前にバージョン管理システムに保存
- JSON ではなく YAML を使って記述
- 意味がある場合は, 関連オブジェクトを単一ファイルにグループ化
- 不必要にデフォルト値を指定しない

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り得る可能性がある．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ュ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．

公式のベストプラクティスでは, 不用意なデフォルト値の設定は好ましくないが適切なパフォーマンスを行うには, リソースの制限は行うべきである．

K8s では, リソース制限の指定を行うことができ, 必要なリソースの下限は Requests で指定することができる．この際にリソース制限を考慮せずにスケールアウトだけに頼ってしまうと, Requests で指定したリソース容量がノードに無かった場合, pod はスケジューリングされない．上限は, Limits で指定することができ, この場合, 実際の Limits のリソース量が越えないならば全体のリソース総量を越えてしまっても動くことはできる．このように基本的に pod はリソースの Requests を見てスケジューリングされるため, 適切なリソース割り当てができていないと, 実際の使用量と確保される使用量との差が開いてしまい．実際には使用できるリソースがあるが Requests が足りずにスケールされない状況が起きてしまう．また Requests と Limits との差が開きすぎてしまうと負荷が高いのにノードが追加されないといった挙動が起きてしまう．例えば, 割り当て可能 cpu5000millicores(m), Requests:50m Limits:5000m の場合で個のリソース割り当ての pod を 10 個生成した場合を例とすると実際の使用量が 5000m に達したとしても Requests 合計は 500m しか使っていないためまだ, スケジューリングが可能であると判断されてしまいスケールアウトしない状況になってしまう．

リソース制限を全く行わない場合, デフォルトの Requests の空き状況が足りればスケジューリングできてしまう．こういった制限を行わないと永遠にスケジューリングされてしまいプロセスの停止や, 全体の動作に影響を及ぼす可能性がある．しかし, 多くの場合使われる用途や実際に起動してみないとどの程度のリソースでパフォーマンスを発揮するのかは, 実際に動かしてみないと分からないという課題がある．その為, 一概にあるイメージを動かそうとしてもそのイメージが使用する適切なリソース Limits や Requests はわからないという課題がある．その課題を解決するためには実際にそのイメージがどのくらいのリソース量で処理を捌けるのかを知ることができれば良い．その為, 負荷テストを自動化することでこの問題解決をするためにプログラムを用いた自動化プログラムの生成を行う．

ソースコード 1　YAML による CPU リソース指定例

```
resources:
  requests:
    cpu: 50m
  limits:
    cpu: 5000m
```

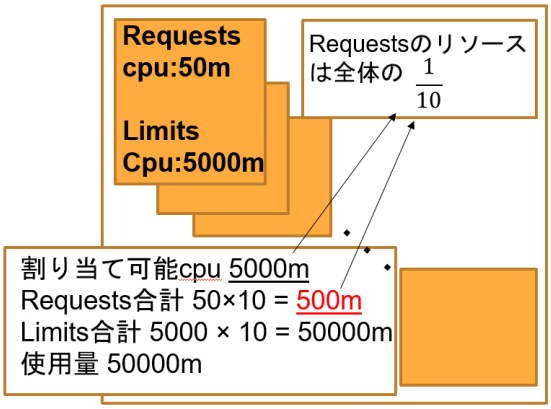

図 1　例によるリソース不足

負荷テストにおける手順は, シナリオ作成, リクエスト数, ユーザ数を考えて, 負荷テストにおけるシナリオを作成する, その後負荷テストツールの準備を行った後にシミュレーションを行う．最終的にその結果を分析するが K8s におけるテストは, 正規化されていないため負荷テストにおける手順を 1 から行わなければならず, 運用時にさらなる手順の追加をしなくてはならず, ユーザへの負荷となりえる．

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, 本稿の課題にについての提案をアーキテクチャと図を使って説明を行う.4 章で実装と評価について述べ,5 章で今後の課題や議論・考

察を行う. 最後に 6 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 [4]. 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 [5]. 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 [6]. 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 [7]. 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 [8]. 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 [9]. 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 [10]. 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].

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少ない.

3. 提案

上記の課題を解決するようなマニュフェストの自動生成と負荷テストの自動化を提案する. またその流れを下記に示す.

(1) ユーザがイメージを指定
(2) イメージを元にした pod を生成
(3) pod ごとに limits(リソース上限) を変更したモノをデプロイ
(4) 3 で生成した pod のリソースで閾値を設定 (例:cpu 使用率が 90%になるまで)
(5) 閾値に達するまで負荷テストを行い, pod ごとの処理結果を提供
(6) 生成した yaml の確認
(7) 3 で生成した pod を元に実際にデプロイして終了
(8) ユーザは yaml マニュフェストと負荷テスト結果を受け取る

今回の提案では, 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 (1) で指定するだけで以下の (2) からの手順は全てプログラムによる自動化を目指す. 自動化することでリソース量から捌けるリクエスト数の算出を行い, リソース制御の手助けが行えるようにする.

4. 実装と評価

4.1 実装

提案で述べた手順を実行するアーキテクチャ図を以下の図 1 に示す.

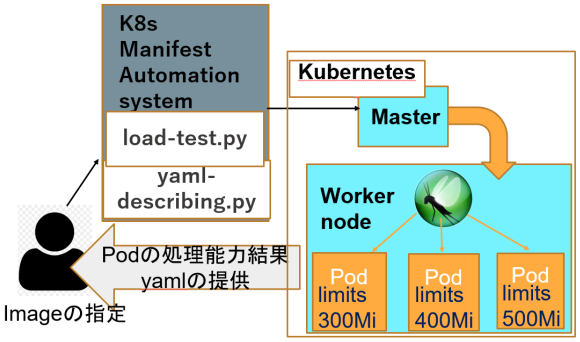

図 2 アーキテクチャ図

ユーザはイメージの指定を行い, それを基に K8s Manifest Automation system でテストの自動化と yaml ファイルの生成を行う. load-test.py では, Python の subprocess モジュールを使用し, python プログラムでコマンド実行に

よるテストの自動化を含めた, データの収集を行う. また後述する yaml-describing.py の起動も自動化の一連の流れとして実行する.

yaml-describing.py では, PyYAML という Python 用の YAML パーサーを利用する. ここでは, ユーザの指定したコンテナイメージを基に K8s のマニュフェストを YAML 形式で生成する.

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. k3s 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 Ubuntu VM1 台を構築し, そこから作業を行う. K8s Manifest Automation system は, アクセスするクライアントに設置を行う予定.

4.3 評価

評価項目としては, マニフェストの記述通りに動作するのか, 自動化された負荷テストのデータが正しいのかという点で評価を行う. 今回は評価はできていないがある 2 点の評価部分を実行できるかを評価として考察していく.

まず 1 つ目は, 自動生成された yaml が本当に正しいのかという点である. yaml-describing.py で生成された YAML マニュフェストは, ユーザの要望通りのイメージで動いているのか, 実際に運用できる状態で動いているかを kubectl コマンドを用いて各種の詳細のデータと共に議論していく.

2 つ目は, 自動化された負荷テストデータが本当に正しいのかという点である. load-test.py で実行された一連のテストは実際に実行できたのかを確認するには, 各手順の実行結果を subprocess を用いてファイルに出力することで各種データと実行結果の有無を確認する. その際に出力各種データ項目は, 以下の 3 つとする.

- kubectl get pods による起動 (STATUS が Raunning になっているか) の確認
- kubectl describe による pod の詳細情報
- kubectl top node・pod で CPU, メモリのリソース使用率の確認

その後, 実行結果のデータ及び実行結果について議論を行う.

5. 議論

前期の実験では足りない部分について記述していく. まず初めにどのようなテストを行うかがいまだ不十分である現段階では, 負荷テストツールとして locust の使用を見込んでいるが locust はコンテナの内部に対しての負荷ではなくコンテナに対しての負荷テストとなるため中で動いているイメージに左右されるのではなく, コンテナのリソースに左右される. その為, コンテナイメージに対する負荷テストを考えなくてはならない. 次に考えられるのはどのようなリソース割り当てで負荷テストを行うかという点である. 負荷テストをメトリクスの使用率から行うがどのようにリクエスト数を増加させるのか初期値のリクエスト数までの実装. また, 上限下限をどう設定という点についても実装には至っていないため今後の実装の課題となる.

6. おわりに

本研究では, K8s のマニュフェストの自動化と負荷テストの自動化を行う提案を行った. K8s では, 実際の運用に関する部分ではいまだ学習コストが高い K8s に対して, 自動化ツールの作成でマニュフェストの作成と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, 利便度の向上に貢献できる. 今後の課題としてユーザに対してどのように pod を提供するかまた, その実装を深く突き詰めていく.

謝辞

参考文献

[1] Anderson, C.: Docker [Software engineering], *IEEE Software*, Vol. 32, No. 3, pp. 102–c3 (online), DOI: 10.1109/MS.2015.62 (2015).

[2] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E. and Wilkes, J.: Large-scale cluster management at Google with Borg, *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France (2015).

[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] 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).

[5] 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).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] Rao, J., Bu, X., Xu, C.-Z., Wang, L. and Yin, G.: VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-Configuration, *Proceedings of the 6th International Conference on Autonomic Computing*, New York, NY, USA, Association for Computing Machinery, p. 137–146 (online), available from ⟨https://doi.org/10.1145/1555228.1555263⟩ (2009).

[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).