マイクロサービスにおけるコンテナID付与によるレプリカレベルでのログ特定

飯島 貴政[1,a)]　串田 高幸[1]

概要：IKontainer は Sidecar パターンを用いて，マイクロサービスにおけるサービスのログ監視の一環としてコンテナにユニークな ID を付与する．マイクロサービス上での同一 IP のコンテナの判別が可能になるサービスコンテナがスケールする環境においても cuid を用いてコンテナを特定できる．これによりスケール/複製されたポッド/コンテナを含むコンテナレベルのトレーサビリティを取得可能となった．クラウドの障害分析の負担を軽減する．また，大規模化するマイクロサービスにおいてリクエストレベルでのサービスのデバッグの負担を軽減する．

1. はじめに

1.1 背景

クラウドサービスではサービス指向アーキテクチャ (SOA) が重視されたマイクロサービスアーキテクチャ (MSA) は Netflix や Amazon 等, 様々な企業で採用されている [1].

マイクロサービスアーキテクチャとは個々の機能を単機能なサービス (以下, マイクロサービスと呼ぶ) としてそれらの集合を用いてユーザーに提供するサービス構築するアーキテクチャである [2]．マイクロサービスアーキテクチャの利点は大きく以下の 4 点が挙げられる．

(1) マイクロサービスごとに別の言語で実装できる
(2) 継続的に行われるテスト (CI)
(3) 自動デプロイ (CD)
(4) 負荷がかかった際に容易にスケールできる

1 つ目は，MSA では各機能が単体で疎結合となっている．そのため各機能ごとに開発をするための言語を使い分けることが出来る. これにより各機能に特化したマイクロサービスを構築することが出来る [3]．2, 3 は開発者が開発したコードをパイプラインと呼ばれる，事前に定義した処理に従って，コンパイルエラーなどのチェックや開発環境への自動デプロイなどが可能になる [4], [5]．4 はマイクロサービスに対して過剰なリクエストなどで負荷がかかった際に同機能のサービスを複製したのち処理を振り分けることで，過負荷にサービスのダウンを防ぐことが可能である [6]．対して，以下のような欠点がある．

(1) 開発にかかるコスト
(2) ネットワークによる障害
(3) MSA におけるログ調査

開発にかかるコスト

1 つ目は MSA では個々に単機能なサーバーが API を用いてネットワーク経由で通信を行う．モノリシックなアーキテクチャと比較すると，機能間での処理されるデータの受け渡しを事前に定義する必要がある [3]. そのため，設計に関わり知識が必要となるため，開発にかかるコストが増大する [7].

ネットワークによる障害

2 つ目は機能と機能がネットワークを経由して通信しているため，ネットワークの帯域幅の圧迫により通信ができない状態になった際には該当ネットワークを用いている機能が利用不可能になる．その結果全体の処理の流れが利用不可能な機能によって止まってしまうことになる．通常このような単一障害点とならないよう，同じ機能を複数にレプリケーションさせたりする対処法が取られている.

MSA における障害発生時のログ調査

上に述べたようにマイクロサービスはそれぞれが単機能かつネットワークを経由して実行されている．そのため，ネットワークのダウン，マイクロサービスのリソース不足が発生した際にサービスの停止や大幅な遅延が発生する (以後，障害と呼ぶ). MSA においては，障害が発生した際には 2 つのタスクが順に実行されるべきである．1 つ

[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)] C0116023

目はサービスを利用可能にするためへの復旧を行うことである．2 つ目はサービスが復旧し，今後同じ原因で障害が発生しないようにするための原因調査を行うことである．しかし，MSA ではそれぞれのマイクロサービス にログの形式が異なるため，障害に関して調査する際には Fluentd や Logstash, Kafka といったログ収集モジュールを用いて，それぞれのマイクロサービスのログを調査することになる [8], [9], [10]．Envoy などといったユーザーのリクエストに対してユニークな ID をつけることで，特定のリクエストについて複数のマイクロサービスをトレース出来る仕組みがある．しかし現在のソフトウェアでは MSA の長点にあったスケールをした際にそれぞれのマイクロサービスの IP アドレスが重複した際に，実際にはどのコンテナが処理を実行したかが調査できないため，対応するオペレーターは意図せず必要のないコンテナのログまで調査を行っていた．本研究では最後に述べた MSA における障害発生時のログ調査の問題点を解決する．

1.2 課題

マイクロサービスにおいて現状のクラウドサービスではサービスのリクエストの情報をそれぞれのサービスで保持している. 障害が発生した際のリクエストごとのステータスを抽出するのに時間がかかっている [11]．既存の MSA におけるログ監視システムではサービスごとのそれぞれの入力や出力，処理内容が含まれるリクエスト情報を保存していた．大規模なクラウドアプリケーションの場合，多くは大量のリクエストを処理するためにマイクロサービスをスケール可能な形でデプロイすることになる [12]．その際に既存の環境では同一マイクロサービス内では IP アドレスの割り当て範囲に限りがあるため，IP アドレスが意図せず重複することがあった [13]．マイクロサービス内のコンテナ (以下サービスコンテナ) では，スケールアップとスケールダウンを繰り返すため同一の IP アドレスでも同一のマイクロサービスが発行したとは限らない．図 1 は Kubernetes システムを用いた時のスケールアップとスケールダウン時のサービスコンテナへの IP 割り当てを示した図である．図中では 10.10.10.2 の IP アドレスが別なコンテナにもかかわらず 10.10.10.2 が割り振られているため，container 2 と new container 2 が判別できない状態である．

そのため，障害が発生し，ログを調査して障害の原因を探す際には過去に存在した同一 IP のサービスコンテナを調査する．直接原因とは関係ないサービスコンテナを調査することで障害原因調査の時間を増加させる．また，該当サービスコンテナを通過したリクエストを周辺マイクロサービスに及ぼした影響を調査することにかかる時間は増加する．クラウドサービスにおける原因調査 (Root Cause Analysis) は次の障害を発生させないためには欠かせない

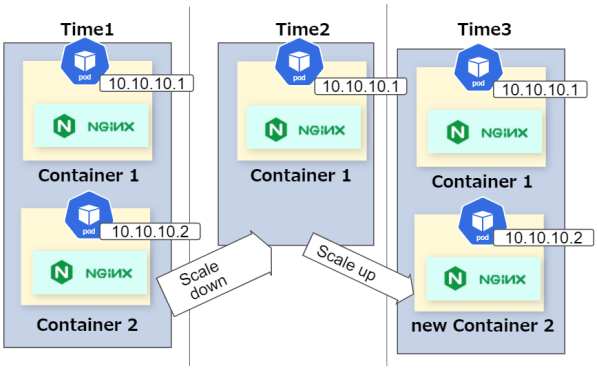

図 1 サービスコンテナスケール時の IP 割り当て

ことである．ログの例としてクラウド上でプロキシ型のロガーとして機能する Envoy で作成されたログを以下のソースコード 1 に示す．

```
1 [2019-03-06T09:31:27.354Z] "GET /status/418
    HTTP/1.1" 418 - "-" 0 135 11 10 "-" "
  curl/7.60.0" "d209e46f-9ed5-9b61-bbdd
  -43e22662702a" "httpbin:8000"
  "172.30.146.73:80" outbound|8000||
  httpbin.default.svc.cluster.local -
  172.21.13.94:8000 172.30.146.82:60290-
```

ソースコード 1 では 172.30.146.73 にて実行された httpbin のログである．リクエストの送信元は 172.21.13.94 であり，172.30.146.82 がリクエストの送信先であることが取得できる．前述のとおり，172.30.146.73 がスケールされたサービスコンテナだと仮定した場合，これらの情報は 172.21.13.94 に紐図いており，今後別の同じ IP が割り振られた場合はログメッセージだけではどのタイミングでの 172.21.13.94 であるかを明確にすることができない．

2. 関連研究

クラウドサービス上のマイクロサービスの監視・制御にかかる技術者の時間を短縮し, 新たなクラウドサービスを設計する際に, 実装を抽象化してパッケージ化することで各サービスの状態を共有し, 適切なリソースをクラウドに格納できるシンプルな仕組みを提供することにある．実際のシナリオでは, デプロイは運用コストに制約される [14]. クラウドリソースをリアルタイムで管理することで, その制約の中でサービスが展開され, リソースに負荷がかかってしまった場合のダウンタイムやプリクエストの失敗を軽減することができる. ハイブリッドクラウドやマルチクラウド環境では, コスト削減策は各ベンダーに依存する [15]. クラウド資源の節約は, 信頼性が最も重視されるサービス以外のサービス（バックアップや災害対応サイトなど）では, 経常的なコストを削減できるので有用です. そのため, ベンダーに依存しない形でリソースを削減することが望ましい. また, マイクロサービスに障害が発生した場合, サービスの平等性を確保するために, リクエストを再試行し, リク

エストレベルでの各マイクロサービスの状態をネットワークと共有しなければならない. 現在のマイクロサービス間のデータ共有方法では, 特定のノード間でのみ通信を行っているが, より透明性を高めるためには, すべてのノードがサービスの状態にアクセスできるようにする必要がある.

Meina Song らは, Envoy を用いて, アプリケーション層, Envoy が配置されたプロキシ層, Zipkin と Jeager を用いたデータアナライザ層を作成した. そのデータを用いて障害検出を行うシステムを提案している. この方式では, Istio とは別に Data Analyzer 層を利用するため, Istio の強みである導入の難易度が高くなっている. 本研究では, Istio の Control plane と Envoy を目標として, 各サービスの状態を監視し, 関連するサービスを見に行くレポーターをネイティブに組み込める形での実装とした.

サービスメッシュ上の監視・制御トラフィックに関する研究では, マイクロサービスアーキテクチャにおけるサービス間の相互監視において, アプリケーションのテナントとは別のデータに監視・制御トラフィックを展開することが困難であることが示されており, セキュリティ上の問題が発生する可能性があることが, [16] によると示されている. これに対して, Envoy 側で監視制御データを同期させながら, アプリケーションテナントがパスに入らない設計を提案している. 本研究では, リクエストデータ取得の性質上, アプリケーションテナントと監視・制御テナントを完全に分離することはできなかった. 本論文では, プロキシ側のリクエストの生データから抽出するのではなく, アプリケーション側でリクエストの状態をあらかじめ定義しておくことで, 監視・制御用のデータを暗号化することができ, 従来のパフォーマンスを損なわないモデルを採用している. クラウドサービスにおけるリソースプロビジョニングの研究では, クラウドブローカやスケジューラを利用したプロビジョニングの手法がある. しかし, 監視・展開するモジュールはサービスネットワークの外部に実装されるため, 外部のプロビジョニング計算サーバへのアクセスを検討する必要がある. しかし, 構成要素がサービスから独立している各サービスの Envoy から Istio のコントロールプレーンにこれらを報告するモデルをサービスメッシュに統合することで, 外部との通信を追加実装する必要がない. プロビジョニングの意思決定をネットワーク内で行うことで, より迅速な意思決定が可能となり, 外部サービスに依存しないプロビジョニングが可能となる.

Envoy のプロキシによるロギング手法は既存の Kubernetes 環境の追加のコンフィグが最小限でパフォーマンスを損なわないため, 本研究においても同じサイドカーモデルを用いてクラウド上にデプロイすることとする.

3. 提案

同一 IP のサービスコンテナを調査する際にサービスコンテナごとでのログを調査が可能になるようにサービスコンテナを立ち上げる際に以下の 3 つを行うコンテナ (以下 IKontainer) をサイドカーとしてデプロイする.

- UUID を作成する
- cuid をコンテナに保存する
- cuid を各ログメッセージに追加する

すべてのサービスコンテナへの通信は IKontainer を経由するため, サービスコンテナ内のログは同一 Pod の共通ストレージに存在する. また, リクエストの入出力に係るログは IKontainer を経由した際に自動で取得できる. 例として nginx をサービスコンテナとして動かしているときの IKontainer のデザインパターンを以下の図 2 に示す.

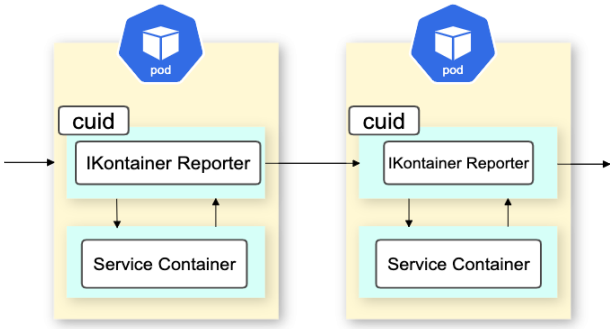

図 2 IKontainer デザインパターン

サービスコンテナごとにユニークな Container unique id (以下 cuid) を Pod が起動したときに割り当てる.

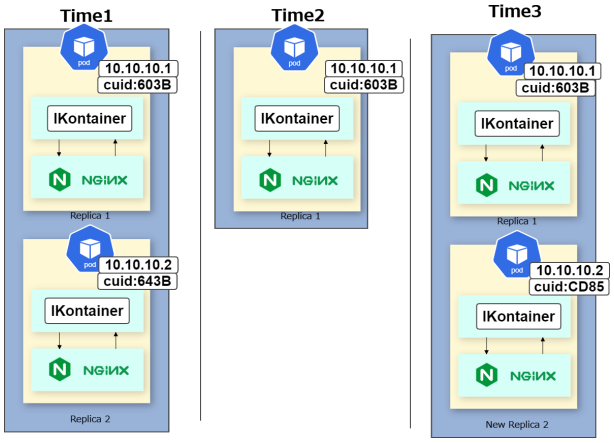

図 3 IKontainer を使った後のコンテナ差分の明確化

環境の例としてクラウド上でプロキシ型のロガーとして機能する Envoy で作成されたログを以下のソースコード 1 に示す.

```
1 [2020-10-23T01:35:31. 442Z] "GET /status
    /418 HTTP/1.1" 418 - "-" 0 135 11 10
    "-" "curl/7.60.0" "d209e46f-9ed5-9b61-
    bbdd-43e22662702a" "httpbin:8000"
    "172.30.146.73:80 " cuid : 76473b06
    -332d-4095-87d3-c9ea68f7a2ef " " "Image
     :Nginx" "Name:Nginx: " outbound|8000||
    httpbin.default.svc.cluster.local -
    172.21.13.94:8000 172.30.146.82:60290
```

ソースコード 2 ではログに cuid が付与されているため，同一のサービスコンテナでのログを調査する際には cuid でフィルタリングすることで不必要な調査をする必要がなくなる．

4. 実装

4.1 実装

IKontainer のレポーターは Python コンテナを実装する．コードの実行速度の高速化のため，コードは Cython を用いる記述方式にて作成する [17]．コードをデフォルトの Python から Cython に変更するだけで JSON の読み書き速度がデフォルト 0.933 秒に対し Cython の場合は 0.125 秒と 86.6%高速化できている．そのため既存の Kubernetes システムと比較してのオーバーヘッドを抑えることが可能になる．各 cuid の保存，新規 Pod が起動した際の既存の ID との衝突チェックには MariaDB を用いる．また，このノードの名前を Status store pod とする．Status store pod では主に Reporter から POST されてきたデータを整形し，ID とリクエスト内容を保存する．サービスコンテナと IKontainer の通信は同一 Pod 内のローカルネットワーク上で行う．別サービスコンテナからの通信は HTTP で受け付ける．リクエストを受け付けた際に cuid をリクエストのヘッダーに付与する．以下の図 4 に各サービスコンテナ間の通信について示す．

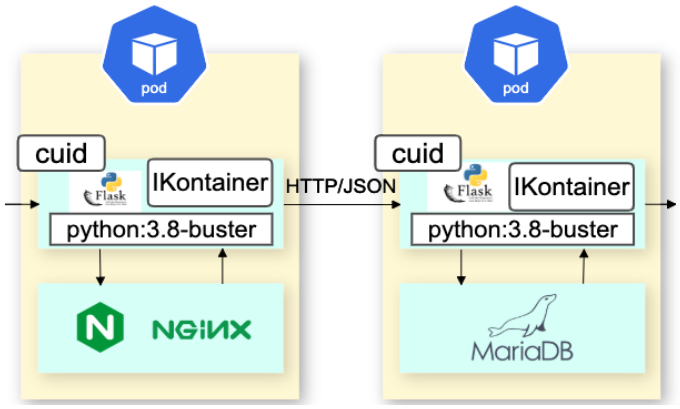

図 4 実装概要図

4.2 実験環境

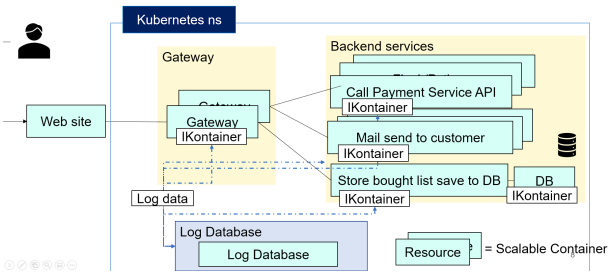

図 5 実装概要図

実験は Kubernetes システム上で以下のアーキテクチャで行った．

- Spring boot:WEB フロントエンド
- Flask(Scalable max2):Gateway
- haproxy(Replica max2):LoadBalancer
- Flask(Scalable max3):決済中継 API
- Ubuntu(Scalable max3): メール送信 API
- Flask(Scalable max3) : 決済情報登録 API
- MongoDB : 決済情報登録 DB

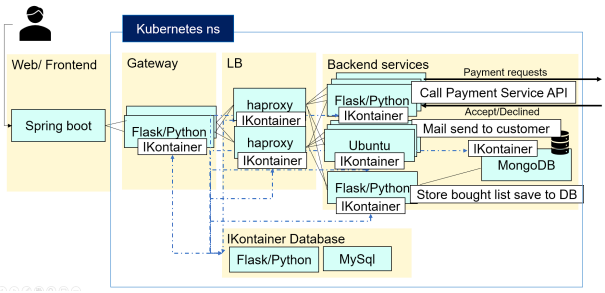

図 6 実装機能図

5. 評価と分析

以下の二点を評価する．

- 既存の手法とのサービスの入出力の遅延の比較
- 検索ステップ数の比較

5.1 既存の手法とのサービスの入出力の遅延の比較

前に述べた実験環境において 1 万回 HTTP リクエストを送信し，Kubernetes のみ，Istio デプロイ済み，IKontainer デプロイ済みの 3 つの環境で応答速度を比較した．

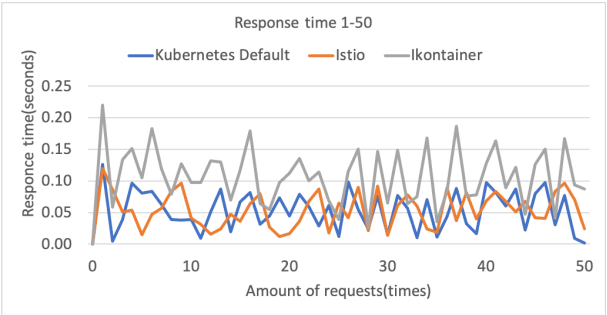

図 7 試行回数 50 回までのレスポンスタイム

ネイティブの Kubernetes に対して約+50%，Istio 環境に対して約+40%の応答速度の遅延が見られた．これは IKontainer の通信部分の実装が HTTP であり，IKontainer 自体の cuid 生成コンテナの維持とコンテナ間の通信時にかかる応答時間の遅延によるものと考える．Istio ではコンテナ間の通信は gRPC によって最適化されているため，IKontainer も gRPC を用いて通信を行うことで応答時間の改善が見込まれる．

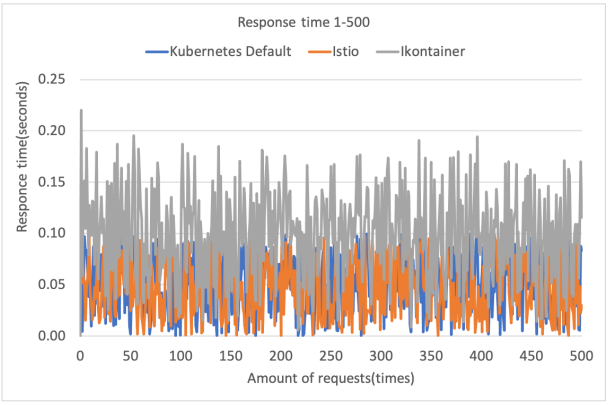

図 8 試行回数 500 回までのレスポンスタイム

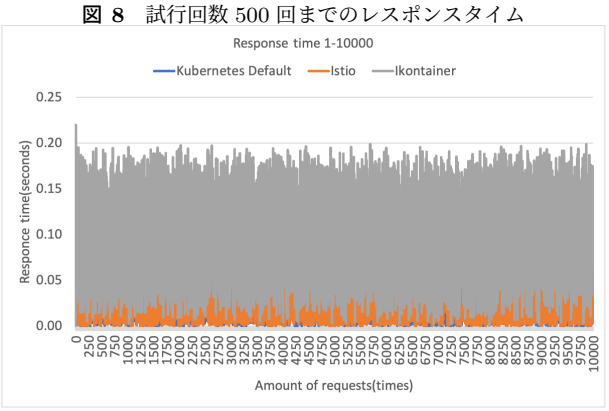

図 9 試行回数 1 万回までのレスポンスタイム

5.2 検索ステップ数の比較

システムダウンや障害が発生した際には, どのコンテナが障害の原因となっているのか, 他のサービスに影響を与えているのかを調査する必要がある．このケースでは, 図 7 の最下ログの Payment サービスからアラートが発生したとする．

現在の解決策は, IP からチェックして, サービスが通過したリクエストをチェックしている．

System alert in ip: “10.10.11.2 “

name	ip	request id
Gateway	10.10.1.2	46BB
Gateway	10.10.1.2	966D
Payment	10.10.11.2	46BB
Mail	10.10.12.1	46BB
Payment	10.10.11.1	966D
Store	10.10.13.1	46BB
DB	10.10.100.1	46BB
Gateway	10.10.1.2	7AD0
Gateway	10.10.1.2	F02F
Payment	10.10.11.2	7AD0

図 10 IP ベースでのログ調査ステップ

この場合, 10.10.11.2 のシステムでは, その IP から 2 つのログを見つけることができる．2 つのログのリクエスト ID を確認すると, 46BB と 7AD0 がある．そして, それぞれのリクエスト ID を確認することで, どのサービスがこのリクエストに影響を与えたのかを明確にすることができる．7 つの log のメッセージは存在するが, リクエスト 46BB はすべて正常に処理されている．また, そのコンテナはスケール前のものとは別のコンテナを使用しているため, これらの 5 つのログは必要ない．そのため, これら 5 つのログは不要な調査でである．

一方で, cuid を含めたアラートが出ている場合は, そのサービスがどのサービスに影響を与えたのかを明確にする必要がある．どのコンテナが使用されたかを追跡することができるが, この場合, そのコンテナのログは 1 つのみである．

System alert in cuid: ”7AD0”

Name	ip	cuid	request id
Gateway	10.10.1.2	643B	46BB
Gateway	10.10.1.2	643B	966D
Payment	10.10.11.2	2F52	46BB
Mail	10.10.12.1	82B8	46BB
Payment	10.10.11.1	7D53	966D
Store	10.10.13.1	CC62	46BB
DB	10.10.100.1	17E1	46BB
Gateway	10.10.1.2	CD85	7AD0
Gateway	10.10.1.2	CD85	F02F
Payment	10.10.11.2	4A23	7AD0

図 11 cuid ベースでのログ調査ステップ

また, 各リクエストの ID を確認すると, 2 つのログがあり, 両方ともそのコンテナに関連したリクエストである．これは調査が効率的であることを意味する．

6. 議論

IKontainer ではサイドカーデザインによってサービスコンテナと同一の Pod に起動する仕組みによって cuid を定義，利用していた．しかし，大量にスケール環境においては IKontainer がサービスコンテナを立ち上げる時間を既存のサービスメッシュの手法やロガーと比較して大幅に遅延させているため，IKontainer 自体のコンテナイメージサイズを縮小，起動を高速化する必要がある．

7. 終わりに

IKontainer では同一 IP が割り振られていたサービスコンテナのにサイドカーとしてデプロイすることでコンテナイメージに cuid を付与した．リクエストヘッダーに cuid を付与することでサービスコンテナがスケールする環境においても cuid を用いてコンテナを特定できる．これによりスケール/複製されたポッド/コンテナを含むコンテナレベルのトレーサビリティを取得した．これはクラウド上における障害の調査時間の短縮に貢献した．

参考文献

[1] Thönes, J.: Microservices, *IEEE Software*, Vol. 32, No. 1, pp. 116–116 (online), DOI: 10.1109/MS.2015.11 (2015).

[2] Balalaie, A., Heydarnoori, A. and Jamshidi, P.: Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture, *IEEE Software*, Vol. 33, No. 3, pp. 42–52 (online), DOI: 10.1109/MS.2016.64 (2016).

[3] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. and Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud, *2015 10th Computing Colombian Conference (10CCC)*, IEEE, pp. 583–590 (2015).

[4] Chen, L.: Microservices: Architecting for continuous delivery and devops, *2018 IEEE International Conference on Software Architecture (ICSA)*, IEEE, pp. 39–397 (2018).

[5] Zimmermann, O.: Microservices tenets, *Computer Science-Research and Development*, Vol. 32, No. 3-4, pp. 301–310 (2017).

[6] Hasselbring, W. and Steinacker, G.: Microservice architectures for scalability, agility and reliability in e-commerce, *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, pp. 243–246 (2017).

[7] Kalske, M., Mäkitalo, N. and Mikkonen, T.: Challenges when moving from monolith to microservice architecture, *International Conference on Web Engineering*, Springer, pp. 32–47 (2017).

[8] Dunning, T. and Friedman, E.: *Streaming architecture: new designs using Apache Kafka and MapR streams*, " O'Reilly Media, Inc." (2016).

[9] Indrasiri, K. and Siriwardena, P.: Microservices for the enterprise, *Apress, Berkeley* (2018).

[10] Balalaie, A., Heydarnoori, A. and Jamshidi, P.: Microservices architecture enables devops: Migration to a cloud-native architecture, *Ieee Software*, Vol. 33, No. 3, pp. 42–52 (2016).

[11] Manouvrier, M., Pautasso, C. and Rukoz, M.: Microservice Disaster Crash Recovery: A Weak Global Referential Integrity Management, *Computational Science – ICCS 2020* (Krzhizhanovskaya, V. V., Závodszky, G., Lees, M. H., Dongarra, J. J., Sloot, P. M. A., Brissos, S. and Teixeira, J., eds.), Cham, Springer International Publishing, pp. 482–495 (2020).

[12] Hasselbring, W. and Steinacker, G.: Microservice Architectures for Scalability, Agility and Reliability in E-Commerce, *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 243–246 (online), DOI: 10.1109/ICSAW.2017.11 (2017).

[13] Burns, B., Grant, B., Oppenheimer, D., Brewer, E. and Wilkes, J.: Borg, omega, and kubernetes, *Queue*, Vol. 14, No. 1, pp. 70–93 (2016).

[14] Villamizar, M., Garces, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A. et al.: Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures, *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE, pp. 179–182 (2016).

[15] Van den Bossche, R., Vanmechelen, K. and Broeckhove, J.: Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads, *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 228–235 (online), DOI: 10.1109/CLOUD.2010.58 (2010).

[16] Kang, M., Shin, J. and Kim, J.: Protected Coordination of Service Mesh for Container-Based 3-Tier Service Traffic, *2019 International Conference on Information Networking (ICOIN)*, pp. 427–429 (online), DOI: 10.1109/ICOIN.2019.8718120 (2019).

[17] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S. and Smith, K.: Cython: The best of both worlds, *Computing in Science & Engineering*, Vol. 13, No. 2, pp. 31–39 (2011).

