2017 Fifth International Conference on Advanced Cloud and Big Data

BigVM: A Multi-Layer-Microservice-Based Platform for Deploying SaaS

Tianlei Zheng
School of Computing and Information Systems
The University of Melbourne
Melbourne, Victoria, Australia
Email: tianleiz1@student.unimelb.edu.au

Yuqun Zhang
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, Guangdong, China
Email: zhangyq@sustc.edu.cn

Xi Zheng
School of Information Technology
Deakin University
Melbourne, Victoria, Australia
Email: xi.zheng@deakin.edu.au

Min Fu
Department of Computing
Faculty of Science and Engineering
Macquarie University
Data61, CSIRO
Sydney, Australia
Email: min.fu@data61.csiro.au

Xiao Liu
School of Information Technology
Deakin University
Melbourne, Victoria, Australia
Email: xiao.liu@deakin.edu.au

***Abstract*—With the advent of Software-as-a-Service (SaaS), SaaS developers are facing many more challenges associated with multi-tenancy and dramatically increased number of users, e.g., scalability, availability, increased cost of development/testing/deployment, high cost of customization. As most of them are highly common, it is becoming very desirable if a generic and powerful deployment platform can be designed. For such a purpose, in this paper, a new platform namely BigVM is proposed to isolate SaaS developers from deployments and bridge the gap between the best practices and the real-world adoptions.**

BigVM provides microservice-oriented deployment kits to enable SaaS developer to create, customize, and deploy SaaS solutions in a multi-layer-microservice-based manner, which can utilize fault tolerance, optimize the resources, and scale in/out the underlying resources not only based on resource utilization but also on the non-functional requirements from the system, e.g., timing constraint. A set of experiments are implemented in sysbench to test one of BigVM's core components—Docker containers. The results show that Docker containers can achieve desirable performance in terms of CPU workload and file I/O, thus laying a solid foundation for our future work.

I. Introduction

One of the major reasons that SaaS (Software as a Service) is becoming increasingly popular is that software applications do not have to be developed through a long lifecycle as on-premise development such that new reports, data entries, etc., could be delivered or added on the fly. Many challenges for SaaS developments arise when applications are featured being real-time, automated, or batched where a minor change might risk breaking the critical business processes.

Some best practices have been concluded to deal with those challenges. However, they are not always employed for configuration managements in SaaS environments for the reasons including *1)* the applications might be supported by business instead of IT departments; *2)* SaaS administrators may not be familiar with the practices of configuration and release management; *3)* deploying an application often requires both manual and automated steps [1]. 困难

In this paper, a new framework namely BigVM is proposed to bridge the gap between the best practices and the real-world adoptions. BigVM aims at resolving the hassles in multiple-tenant SaaS developments and deployments for the developers. In particular, BigVM focuses on identifying the microservices that are highly reused and standardized in multiple applications. A set of standardized component interfaces and service communication protocols are implemented such that these microservices can be fully maintained in BigVM. Furthermore, BigVM enables a multi-layer microservice hierarchy, where the lower-layer microservices are featured being "black box" such that SaaS developers only need to deal with higher-layer microservices by automatically customizing the lower-layer microservices. As a result, SaaS developers can skip deploying a number of microservices, e.g., OS-resource-intensive microservices.

BigVM can enhance system performances by applying certain components. Particularly, a microservice orchestration engine is used to orchestrate microservices. A workflow engine is applied such that microservices can be choreographed and dynamically customized during runtime.

To evaluate and demonstrate how BigVM benefits SaaS development, a set of experiments that test one of its components—Docker container are implemented in sysbench. The experimental results indicate that Docker containers can achieve desirable performance in terms of CPU workload and file I/O, which validates the advantages of the BigVM architecture.

The remainder of this paper is organized as follows. Section II introduces the related work. Section III demonstrates

978-1-5386-1072-5/17 $31.00 © 2017 IEEE
DOI 10.1109/CBD.2017.16

BigVM with its associated features and potential advantages. Sections IV demonstrates the experimental results which show the advantages of BigVM. Section V concludes this paper and points out the future work.

II. Related Work

Deploying SaaS has been a long-term research topic. A framework that provisions applications and their associated infrastructure using workflows are presented in [2]. In [3], explicit variability models are proposed to systematically derive customization and deployment information for individual SaaS tenants. Specifically, these models can be derived by the already deployed SaaS application for handling new tenants. Another set of approaches that automatically configure tenant-specific applications are proposed in [4] based on feature modeling and XML filtering techniques. They are evaluated to excel in execution time. However, none of these approaches aim at handling microservice-based architectures, partly because microservice-based architectures have become popular very recently. Some microservice-oriented approaches, such as [5], can function only primitive so far.

Docker containers [6] are widely used in microservice-based cloud infrastructure in industry. A reusable architectural pattern address the problem of migrating a legacy Web application to a cloud service is developed in [7] where Docker containers are used to to deliver a multi-tenant cloud service by re-using a legacy codebase. Docker containers are used in edge computing [8] to offload the processing to the edge from centralized to decentralized paradigm that indirectly reduces application response time and improves overall user experience. The security design and architecture quality using multilateral security framework for Docker container are investigated in [9] by using OSI/TCP/IP stack model with reference to Cloud service stack model and deployment stack model.

Microservices are used to specify implementation approaches for service-oriented architectures (SOA) to build flexible, undependable, and deployable software systems [10]. Instead of building a monolithic application in which features are coded and deployed as a whole, microservice-based architecture implements each feature as independently running services that communicate via some protocols like HTTP or RPC. Microservices are widely considered being desirable to structure SaaS systems [5], because microservice-based architecture not only allows for greater flexibility in development (implementing each component using different programming language and technology) and better scalability (scaling each component independently), but also implies fundamental changes in organizational structures. In traditional organizational structures, IT professionals are usually grouped into different departments by their skill sets, while microservice teams are usually cross-skilled where developers and operationals in one team take sole responsibilities for the entire lifecycle of a product (from development to deployment to maintenance). Such organizational structure allows for shorter lifecycle development since developments, testings and re-developments take place in one team and hence result in less communication inefficiency and clearer responsibilities among different departments. The recent hype of containerization and all-in-one solutions, e.g., Docker container has inspired the utilization of microservices due to its features such as networking optimization and cluster management.

III. BigVM Architecture

BigVM is proposed to resolve the increasing challenges of developing and deploying multiple-tenant solutions for SaaS providers (In this paper, we use the terms "SaaS providers" and "SaaS developers" to refer to organizations and individuals who build and sell SaaS services, rather than people who build business solutions based on customizing SaaS applications.). In particular, BigVM can deliver SaaS applications with the following advantages: *1)* SaaS applications can be highly-customizable while still providing strong abstraction ("black box"). *2)* Accordingly, it is possible for non-developers (e.g., SaaS providers) to conduct developments by themselves, e.g., assembling development kits as needed.

A. Domain Problems

Though microservices are considered to improve multiple-tenant SaaS applications, some challenges regarding microservices arise and many challenges regarding multiple-tenant SaaS applications still remain unsolved. Some major ones are listed as follows.

1) Inefficient deployment: The state-of-the-practice for configuring microservice-based SaaS applications can be generally categorized to two groups—macro- and micro- deployments. Macro-configuration refers to the uniform deployments for the overall system, where each microservice obtain almost identical resource. Micro-configuration takes one more step forward and focuses on configuring individual microservices. It is difficult for original deployment schemes to stay optimal under dynamic circumstances. For instance, by investigating a giant Chinese food supply online service store that employs more than 10,000 microservices in its SaaS applications, it is observed that the inappropriate macro-deployment leads to inaccurate resource allocation for individual microservices while the inappropriate micro-deployment results in triviality. A straightforward improvement can be realized by combing these two methods. However, the efficacy is limited because the combination policies are hard to be made accurate with incidental holidays and/or joining and canceling branches and customers when loads become dynamic and are hard to be provisioned.

2) infrastructure limitation: No matter whether IaaS or bare-metal servers are used, there is still an inevitable human resource and equipment cost that the SaaS providers want to avoid. Specifically, when on-premise solutions (physically owned servers) are employed, it is quite often that engineers need to take on-call duties in a 24/7 manner whenever servers are overloaded. The on-call duties are about plugging-in new servers, manually deploying software systems, etc. When Cloud IaaS is used, SaaS providers do not need to plug-in servers physically yet they still tend to encounter high costs

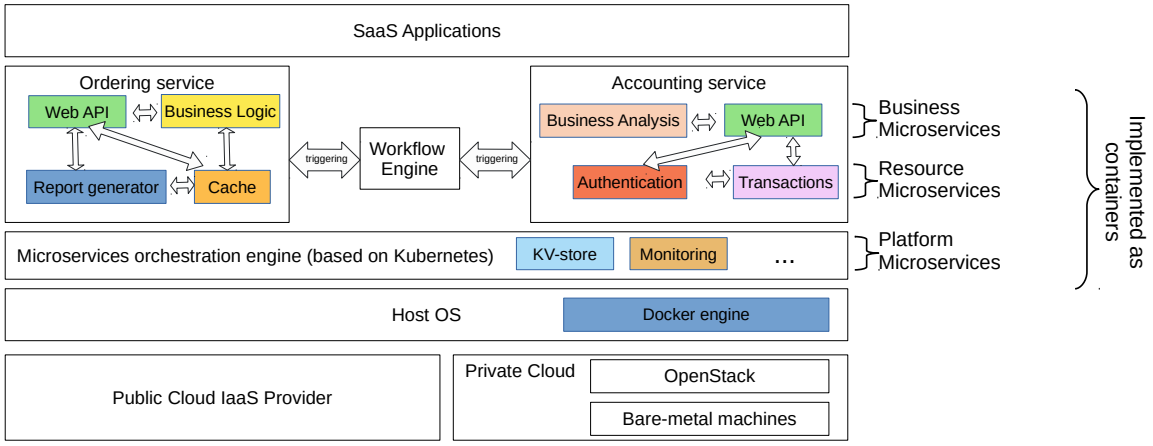

Fig. 1: BigVM Architecture and Deployment Environment

due to static provisioning. For either way, SaaS providers have to pay addtional costs.

B. BigVM Architecture

BigVM is a solution to provide more automation for developing and deploying microservices in SaaS applications where technical concerns can be separated from business concerns, so that even non-technicians can compose microservices together to achieve desirable features.

BigVM is deployed in a recent dominating manner as Figure 1 where the bottom layer can be either private cloud IaaS that can be provisioned by OpenStack [11] implemented on bare-metal servers or public cloud IaaS with the host OS layer built above. Docker containers are layered on top of the host OS layer, that are coordinated with the microservice orchestration engine, e.g., Kubernetes [12]. This architecture advances in fast launching time (within a few seconds) and dynamic configuration due to its light-weight process isolation features. Therefore, it could be helpful for provisioning the SaaS applications and composing the corresponding microservices, especially on the fly.

1) Multi-Layer Microservice Hierarchy: Nowadays, the APIs of the underlying cloud infrastructure are widely used by SaaS developers to build microservices. While each microservice can achieve better agility and independence to form a business process, a number of work regarding utilizing the APIs that abstract the core OS codes from the underlying hardware are highly reused and standardized, such as runtime monitoring (file usage, memory usage, database usage, etc), logger, and key-value (kv) storage. Usually they are common to be implemented in all SaaS systems. By designing BigVM to obtain or estimate the pre-knowledge of the resource utilizations of these specific microservices and the potential associated pitfalls, these microservices can be encapsulated in Docker containers and orchestrated by the microservice orchestration engine. Specifically, these microservices are defined as platform microservices, that allows no single point of failure for them.

Furthermore, BigVM enables a mechanism that can identify and peel off all reused and standardized APIs with respect to the underlying resource utilization and allocation for these APIs. Through service choreography techniques, e.g., WS-CDL [13], these APIs can be customized by SaaS developers for their composite service developments and deployments. In this way, microservices can be partitioned to be two layers, where the lower-layer microservices, defined as the resource microservices, refer to the APIs that are intensive to utilize and allocate the underlying resources (report generater, cache, authentication, transactions, etc.), and the higher-layer microservices are defined as business microservices to delineate the ones that are intensively developed towards business-level needs (web API, business logic, business analytic, etc.). Resource microservices enable customization of access for business microservices.

Note that customizing resource microservices implies a possible "black box" feature for BigVM. Ideally, by applying BigVM, not only can it provide SaaS developers with faster and easier developments and deployments, but also it is possible for business actors to get involved in customizing the microservices for business-level needs.

2) Components: To fully realize the "black box" feature for BigVM, some specific components, including a microservice orchestration and a workflow engineengine are applied.

a) microservice orchestration engine: In BigVM, a mi-

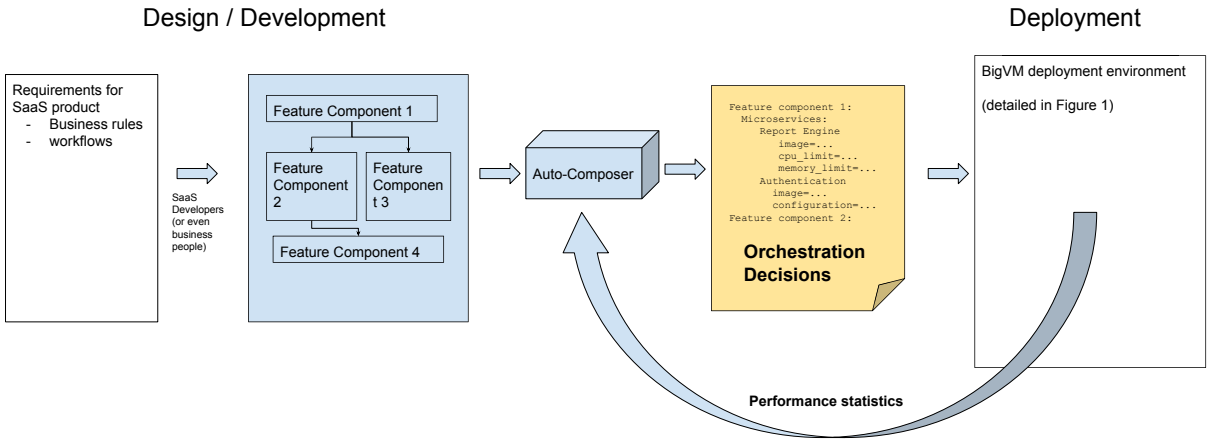

Fig. 2: BigVM Enabled Development Workflow: Seperation of Development and Deployment

croservice orchestration refers to the process that identifies platform microservices and coordinates, assigns, and exposes multiple business microservices and resource microservices as a single aggregated service (ordering service, accounting service, etc.). This process can be as simple as altering the parameter settings for the identical sets of resource microservices, or as complex as handling cross-domain microservice composition. A microservice orchestration engine should take QoS metrics, robustness, etc., into consideration such that resource microservices can consistently deliver the aggregated service for business services to acquire. It is usually used to automate business processes by loosely coupling services across different applications and enterprises and creating new composite applications [14].

A microservice orchestration engine should aim at auto-scaling, which refers to automatically adjusting resource utilization and allocation for the resource microservices. The state-of-the-practice demands an explicitly-specified baseline requirement. However, lacking or having inaccurate baseline requirements tends to cause inaccuracy of resource utilization and allocation as well as performance bottleneck. In BigVM, baseline requirements can be obtained since the associated underlying resource patterns can be designed to be fully understood by platform microservices. Therefore, an efficient auto-scaling can be resolved in BigVM.

Decisions can be made for selecting microservices and composing them to achieve the features requested by SaaS developers. This is made possible by standardized component interfaces and service behavior definition protocols, e.g., WS-CDL. In this way, microservices understand how to communicate with each other and developers do not have to implement their own communication mechanisms for every single project development (which often leads to inconsistency). Some sample microservices and their orchestrations can be pre-built such as as report generation, data aggregator, etc.

b) workflow engine: BigVM employs a workflow execution engine, that adopts a workflow-modeling-compatible approach to align business models with IT architectures, developments, and deployments. Specifically, a workflow engine can be used for resource planning of runtime workflow execution that results in microservice choreography. Therefore, resource microservices can be rendered consistent with the runtime business microservices as part of the "black box". As a result, BigVM enables a better flexibility such that SaaS developers can be further isolated from deployment environments. On the other hand, it is possible that SaaS providers, e.g., project managers, can be more involved in configuring SaaS solutions according to the underlying business microservices.

3) Abstraction and Optimisation: BigVM offers solid functions of abstraction and optimization to reduce the burden on the SaaS developers. Specifically, SaaS developers only need to specify the desired features by putting together the predefined feature components (or custom ones using our Rapid Development APIs). The orchestration decisions are then passed to the deployment environment where the microservice coordination engine, e.g., Kubernetes, implements the decision by provisioning Docker containers.

There are three layers of abstraction in BigVM: *1)* the physical resources that are hidden from the container scheduler by OpenStack of private IaaS cloud; *2)* the container coordination and scaling that are hidden by the scheduler (e.g. Kubernetes) so it appears to be a pool of containers that can meet the needs of microservices developers. These two layers are already realized by the off-the-shelf technologies.

In BigVM, we add a third layer of abstraction that hides the underlying microservices from the feature developers. Usually the state-of-the-practice demands the knowledge of developing microservices. With the multi-layer microservice hierarchy,

the lower-layer platform and resource microservices can be easily accessed by the higher-layer business microservices and can be composed by workflow or orchestration engines. Therefore, SaaS developers do not have to worry about what microservices comprise their applications. Eventually, business actors are able to create applications that are implemented as microservices without knowing code-level details about them.

C. BigVM Enabled Development Workflow

A typical BigVM enabled development workflow can be demonstrated in Figure 2. SaaS developers (or even SaaS providers) can parse the business requirements into feature components, that are delivered to the auto composer (reflected by workflow and orchestration engines) for customizing the business microservices. Then the corresponding resource microservices are identified and orchestrated. The SaaS applications are deployed accordingly and the runtime effects trigger feedback for the auto composer for online orchestration for performance improvements.

D. Benefits

BigVM presents the following benefits due to its desirable features.

1) Cost Efficiency and Optimization: BigVM advances in efficient and optimal cost because *i)* due to the multi-layer microservice hierarchy and the "black box" effects it is associated with, SaaS developers can be freed from highly repeated or standardized deployments, such as OS-level deployments. Therefore labor forces can be reduced. *ii)* all the standardized components, e.g., RESTful APIs, allow BigVM to gather structured information that is easy to access and analyze and can be used to optimize the templates such that all the applications developed above can be benefited. *iii)* BigVM understands the characteristics for each component (sentivity to latency ,CPU intensity etc.) such that it can coordinate resources accordingly.

2) Easily-achieved Fault tolerance: Monitoring and recovery mechanisms are built-in in BigVM that are transparent to user applications. Through the runtime workflow planning and verification provided by the orchestration engine and workflow execution engine, fault tolerance can be easily achieved and SaaS developers do not need to tangle the underlying OS-level fault tolerance that are handled by BigVM in our work.

3) Security: Since standardised deployment environment is offered and isolated from SaaS developers, security can be better maintained because less human intervention is expected to be involved—security policies can be enforced by BigVM on multiple layers, rather than depending on each individual developer.

IV. Evaluations

It would take a long-term development lifecycle to fully realize the functionality of BigVM. Note that BigVM can be built on many infrastructures, e.g., Docker containers. At this point, the efficacy of BigVM can be deduced by evaluating the performance of the Docker containers that are applied within. Particularly, a set of tests are conducted to evaluate the performance of the structure with Docker containers on top of virtual machines against the one without. All the tests are run on an AWS Lightsail VM with 512MB memory, 1 Core Processor, and 20GB SSD Disk (1GB as swap). The running system is Ubuntu 16.04.1 LTS and the Docker version is 1.13.1, build 092cba3. We use sysbench, which is a tool to provide benchmarking capabilities by supporting testing CPU, memory, File I/O, and so forth. Due to the lack of the data from other SaaS platforms, we do not enable comparisons of evaluation metrics in this paper.

A. CPU Benchmarking

When running with the CPU workload, sysbench verifies prime numbers by doing standard division of the number by all numbers between 2 and the square root of the number. When a remainder of 0 is given by a certain number, the next number is iterated with the identical process. Hence this iteration puts stress on the CPU with a limited set of the CPUs features.

Table I delivers the experimental results of the CPU load benchmarking. It can be observed that for both structures with or without Docker containers, the minimum, maximum, average completion time per request and the threads fairness execution time grow proportionally when the maximum run grows. Interestingly, it can be observed that overall, the response time of the structures with and without Docker containers are similar sometimes. The structure with Docker containers is even better, which can validate the advantages of the process isolation feature of Docker containers.

B. File I/O

Table II presents the experimental results of File I/O testing with file size 1G. To test file I/O, a set of test files with size larger than the available memory are utilized such that the file caching does not impact on the workload too much. It can be observed that when the number of threads grows, the minimum, maximum, average response time per request, the threads fairness events, and the threads fairness execution time grow accordingly. Typically, the structure with Docker containers takes averagely 4.8% to 36% longer than the one without Docker containers to complete the tasks, that can be totally acceptable given the reasonable amount of threads.

Our experimental results for Docker containers indicate that it can effectively use CPU without compromising the file I/O by much. It could be deduced that BigVM being built on Docker containers can achieve the similar performance while advancing in the Docker container features, such as process isolation.

V. Conclusions and Future Work

In this paper, we present a microservice-based architecture for SaaS applications namely BigVM, which can partition the traditional microservices into multiple layers. By targeting the lower-layer microservices and coordinating them, BigVM can enable a flexible customization for the business microservices

	WithDocker CPU Benchmarking maximum run = 10000	WithDocker CPU Benchmarking maximum run = 20000	WithDocker CPU Benchmarking maximum run = 30000	WithDocker CPU Benchmarking maximum run = 40000	WithoutDocker CPU Benchmarking maximum run = 10000	WithoutDocker CPU Benchmarking maximum run = 20000	WithoutDocker CPU Benchmarking maximum run = 30000	WithoutDocker CPU Benchmarking maximum run = 40000
per-request min	1.07ms	2.78ms	4.87ms	7.39ms	1.03ms	2.82ms	4.91ms	7.40ms
per-request max	5.61ms	3.55ms	13.23ms	12.77ms	5.08ms	3.48ms	5.80ms	14.74ms
per-request avg	1.17ms	3.00ms	5.24ms	7.77ms	1.16ms	2.99ms	5.23ms	7.77ms
Threads fairness execution time (avg/stddev)	11.6617/0.00	29.9566/0.00	52.4276/0.00	77.6734/0.00	11.6075/0.00	29.9141/0.00	52.3018/0.00	77.7133/0.00

TABLE I: CPU benchmarking with dockers vs. without dockers

	WithDocker number of threads = 32	WithDocker number of threads = 64	WithDocker number of threads = 128	WithDocker number of threads = 256	WithoutDocker number of threads = 32	WithoutDocker number of threads = 64	WithoutDocker number of threads = 128	WithoutDocker number of threads = 256
per-request min	0.00ms	0.00ms	0.00ms	0.00ms	0.00ms	0.00ms	0.00ms	0.00ms
per-request max	37.48ms	157.18ms	281.11ms	431.68ms	101.13ms	71.86ms	197.83ms	576.57ms
per-request avg	2.96ms	6.57ms	13.04ms	21.06ms	2.64ms	5.06ms	9.54ms	20.09ms
Threads fairness event (avg/stddev)	312.5000/29.72	156.2500/20.43	78.1406/18.82	39.0625/15.25	312.5000/17.50	156.2500/16.27	78.1250/15.99	39.0625/13.02
Threads fairness execution time (avg/stddev)	0.9249/0.06	1.0261/0.14	1.0193/0.24	0.8225/0.30	0.8256/0.06	0.7900/0.09	0.7449/0.22	0.7847/0.19

TABLE II: File I/O with dockers vs. without dockers

and therefore reduce the burdens of SaaS developers. Moreover, some components such as workflow and orchestration engines can help BigVM deliver properties such as cost optimization, fault tolerance, and security. A set of experiments are implemented that focus on the performance of Docker containers to estimate the efficacy of BigVM.

The future work of BigVM is to build it into a production-level framework. The development is actually undergoing. By applying the architecture of BigVM, research topics such as load balancing, load prediction, workflow optimization, and so forth can be investigated. Specifically, machine learning techniques can be adopted with the large-sized data flows regarding the underlying resource allocation and utilization can be fully obtained by BigVM. Moreover, a large amount of case studies can be implemented for encouraging business actors to get involved in the utilization of BigVM.

REFERENCES

[1] The challeges of saas. https://www.cio.com/article/2423666/enterprise-software/the-challenges-of-managing-saas-projects.html.

[2] Alexander Keller and Remi Badonnel. *Automating the Provisioning of Application Services with the BPEL4WS Workflow Language*, pages 15–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[3] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25, May 2009.

[4] Y. Cao, C. H. Lung, and S. A. Ajila. Constraint-based multi-tenant saas deployment using feature modeling and xml filtering techniques. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 454–459, July 2015.

[5] C. Xu, H. Zhu, I. Bayley, D. Lightfoot, M. Green, and P. Marshall. Caople: A programming language for microservices saas. In *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 34–43, March 2016.

[6] Docker datacenter. https://www.docker.com/products/docker-datacenter.

[7] A. Slominski, V. Muthusamy, and R. Khalaf. Building a multi-tenant cloud service from legacy code with docker containers. In *2015 IEEE International Conference on Cloud Engineering*, pages 394–396, March 2015.

[8] B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, and O. Hong Hoe. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 130–135, Aug 2015.

[9] A. R. Manu, J. K. Patel, S. Akhtar, V. K. Agrawal, and K. N. B. S. Murthy. Docker container security via heuristics-based multilateral security-conceptual and pragmatic study. In *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, pages 1–14, March 2016.

[10] Microservice. https://en.wikipedia.org/wiki/Microservices.

[11] Openstack. https://www.openstack.org/.

[12] Kubernetes. https://kubernetes.io/.

[13] Web services choreography description language version 1.0 - w3c. https://www.w3.org/TR/ws-cdl-10/.

[14] Service orchestration. https://www.mulesoft.com/resources/esb/service-orchestration-and-soa.

[15] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.