Overcoming Challenges with Continuous Integration and Deployment Pipelines When Moving from Monolithic Apps to Microservices

An experience report from a small company

Vidroha Debroy, *Varidesk Inc.*
Senecca Miller, *Varidesk Inc.*

We recently moved from a monolithic to a microservice-based architecture for the next generation of our applications at Varidesk. This created challenges for our existing continuous integration and deployment pipelines, which were unable to scale in their current form. Having robust pipelines in place is critical to having fast build and release cycles and thus, this was an important concern for us. In this article we present an experience report detailing how we tackled the challenges that arose because of the architectural switch to microservices, and present a novel idea of how the very infrastructure that supports the continuous integration and deployment pipelines for the various applications, can also have its own pipeline. This has allowed us to scale very well as the number of microservices has grown and has shown to be cost-effective.

IEEE Keywords: Microservices, Automation, Cloud computing
Author Keywords: DevOps, Continuous Integration, Continuous Deployment, Containerization, Orchestration

Introduction

Turning code that is in version control into software that is in the hands of users is a non-trivial activity. In fact, every change to software goes through a complex process, possibly involving many phases, on its way to being released, and automation is key to ensuring that things can be done repeatably, quickly and without error [1]. That is why practices such as *continuous integration* (CI) – automating software build tasks such as compilation and testing; and *continuous deployment* (CD) – automating the deployment of software, are rapidly gaining adoption. These practices help organizations reliably release new features [2], and greatly reduce the overall time between when code is completed and made available to end users/customers. For example, reports show that CI/CD can help

companies such as Flickr to deploy software more than 10 times a day [3], and Facebook to increase the frequency of its deployments [4].

Irrespective of the software development lifecycle which an organization employs (be it the Waterfall model, the Spiral model, etc.) the various CI/CD activities are not performed in an ad hoc manner, but rather occur in stages. For example, software must be successfully compiled before it can be tested, and it should be properly tested before it is deployed. A CI/CD *pipeline* defines the flow of these events, in terms of steps that may be performed sequentially or in parallel. In other words, the pipeline models the order of the CI/CD processes and allows us to see and control the progress of each software change as it moves from version control, through various tests and deployments, ultimately to release to users [1]. Having well-defined CI/CD pipelines is considered to be a best practice and in fact, an essential practice from the perspective of *DevOps* [5], which is a portmanteau of 'Development' and 'Operations', and represents another rapidly growing movement in industry. DevOps emphasizes increased automation and collaboration among teams, and the impetus behind it is that software engineers will benefit from better connecting the previously isolated silos of development and operations [6].

The architectural approach of microservices, which emerged out of the idea of service-oriented architecture, has also received considerable attention and adoption in industry [7,8]. Microservices emphasize self-management and lightweightedness as a means to improve software agility, scalability, reliability and autonomy. Each microservice is meant to be modular and having a singular focus, such that large complex applications, can be built by composition of (individually simple) microservices. Microservices are also advantageous because they can be developed, tested and maintained independently of one another [9]. Indeed, each microservice may be written in a different programming language, even though they may be part of the same larger application or software system. There has therefore been a drive to replace large monolithic applications, which can be harder to maintain, with smaller and more focused microservices, which are relatively easier to maintain.

Many studies have been conducted into the areas of CI/CD and microservices, and challenges and impediments that an organization might face when adopting these practices on a standalone basis have also been identified [2,8,9,10,11,12]. However, while CI/CD and microservices are related and both considered good practices, there is a subtle interplay between the two and new challenges can appear when they are adopted concurrently. The move to microservices does not just affect development/testing teams, but also has a very real effect on CI/CD pipelines. To better understand this, consider that a single monolithic application can be built and deployed via a single pipeline; for example, at its simplest: code changes are committed by a developer, the application is built and tested, and then packaged and released as a unit. But the move to microservices means that each microservice should be buildable and deliverable independent of the other which means a pipeline per microservice. At the same time, microservices work together to collectively offer the functionality previously offered by the monolithic app,

and thus, they may all need to be built and delivered together (to offer full functionality). This ultimately translates to individual integration and delivery pipelines (one for each microservice), while also retaining the ability to build and release all of them *en masse*. Consequently, the number of builds and release definitions increase significantly (as opposed to the single definition that was needed with the monolithic application), and this in turn can also necessitate augmentation of build/release infrastructure.

Our current web application (our website) at Varidesk[1] is built on top of a Commercial Off-The-Shelf Content Management System, which also handles responsibilities related to eCommerce, and communication with our Enterprise Resource Planning (ERP) system. It is therefore, somewhat monolithic in its design, and hard to extend and customize. Trying to realize the aforementioned benefits of microservices ourselves, we decided to move towards a microservices-based architecture; decomposing our web app into many smaller, and more focused services[2]. However, in doing so we ran into problems when trying to reconcile the move to microservices with our existing CI/CD pipeline (and infrastructure). This article provides an experience report detailing the challenges we faced when trying to build CI/CD pipelines around the newly adopted microservice-based architecture, and how we overcame them. Studies such as [13] have shown that while there is a general agreement on concepts, there are differences in how CI/CD processes can be interpreted and implemented from one organization to another. Other companies in similar situations are likely to run into the same problems as we did, and similar to studies such as [14], we wish to broaden the body of knowledge in this emergent field, and hope our work can benefit industry and academia alike.

Pipeline and Infrastructure that had Supported our Monolithic App

For our monolithic app – we already had a CI/CD pipeline in place, and we relied heavily on proprietary tooling. Visual Studio Team Services (VSTS), an offering by Microsoft, had been our one-stop-shop for source/version control, task planning and bug tracking, as well as the repository for our build and release definitions. Each build/release definition consists of one or more tasks (each task in turn corresponding to a step in the CI or CD process) which are made available in VSTS when creating/editing the definitions. These definitions together comprise a pipeline as discussed earlier in this article.

Regarding the actual infrastructure – *where the tasks are executed* – Microsoft abstracts this away to the concept of an agent – "installable software that runs one build or deployment job at a time". Microsoft then offers the ability to use their hosted agents –

[1] We are a relatively small company (< 200 full-time employees) based out of Coppell Texas, USA, that offers active workspace solutions for home and office spaces. Our core software development team is comprised of 22 full-time personnel with some off-shore part-time resources, and includes developers, testers, cloud/database administration, and DevOps teams.

[2] The overall development time for this effort spanned a year and a half (between March 2017 and August 2018) and was incremental in nature, resulting in about 24 new microservices to collectively replace the functionality of the original web application.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2019.2947004, IEEE Software

which is where they provide the machine-power and take care of maintenance and upgrades for a certain price point; or the ability to self-host agents where organizations can install the agent software on their own machines. For reasons associated with pricing, we made use of Virtual Machines (VMs) that were hosted in the cloud (in the interests of transparency we state that this was Microsoft Azure in our case, but also clearly acknowledge that the choice of cloud provider has no technical bearing on this discussion, i.e., the same setup is possible using a different cloud provider). We used 2 VMs with one agent each, and each VM was sized at "Standard A2" with 2 vcpus and 3.5 GB of memory, running Windows Server (2012), and this had always been sufficient for us with respect to our monolithic web application. The overall setup is depicted in Figure 1.

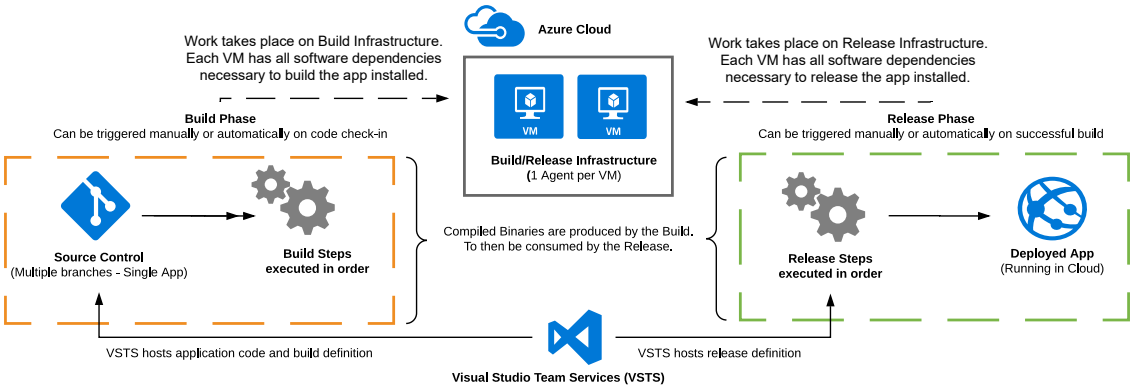

Figure 1. Existing CI/CD pipeline and infrastructure to support the monolithic web app

Challenges Faced When Applying Existing Pipelines to Microservices

It seems like we might have been able to use the earlier model to support our switch to using microservices. But this approach was problematic (in that it led to a number of challenges) and an overlay of the prior model as applied to a microservice architecture is shown in Figure 2 to help understand this.

Challenge C1: Reconciling dependencies among microservices

An advantage of the microservice-based architecture is the reduction of coupling among microservices, and microservices can be written in completely different programming languages. This advantage from the perspective of development, can quickly turn into a disadvantage from the perspective of CI/CD. Each microservice could have different software dependencies and yet the build/release machines would have to be able to support all of them (as without the dependencies present, the code will not even compile successfully). For example, *Service A* may require a certain version of the .Net framework while *Service B* may require a different version. *Service C* may not even be .Net based, but might rely on Java or NodeJs. To support this, using our prior infrastructure and approach, each VM would need to be updated (one by one) with the necessary dependencies (highlighted in red text in Figure 2) to build all of the services successfully.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2019.2947004, IEEE Software

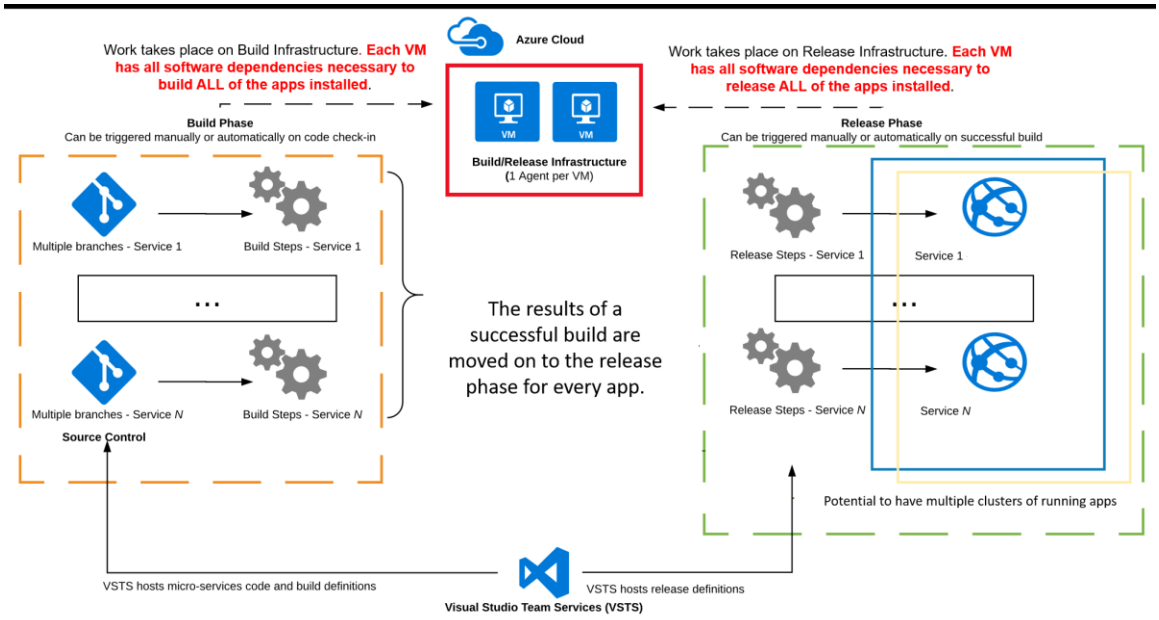

Figure 2 Applying the current CI/CD model to the microservice-based architecture.

Challenge C2: Keeping build and release times low

As mentioned before, we would need to be able to build/deploy each microservice individually, and also do so for all of them collectively. The number of build/release definitions therefore increases greatly and so does the number of actual builds/releases (where each build/release is an instantiation of a build/release definition). Thus, the build/release infrastructure (the 2 VMs highlighted in the red box in Figure 2 now becomes a bottle-neck. If both agents were busy with a current build or release, then any more requests would just get queued, and the queue itself would start growing.

Challenge C3: Keeping infrastructure costs low

Given challenge C2, a naive way to deal with this might be to just to provision more VMs (i.e., scaling horizontally such that there is more infrastructure that is available), but this is unfortunately accompanied with an undesirable growth in costs as each additional unit of infrastructure (in this case a VM) costs more regardless of which cloud provider we went with. The same would be true even if we used our own physical infrastructure as opposed to cloud-based resources.

Challenge C4: Parallelizing builds

An alternative to just provisioning more VMs is to install multiple VSTS agents on a machine (thus, one VM can run multiple builds/releases concurrently). Unfortunately, Microsoft states that users may run into problems if concurrent build processes are using common dependencies such as NPM packages – for example, one build might update a dependency while another build is in the middle of using it, which could cause unreliable results and errors. This applies to us as our new microservices use NPM packages and

thus, we were restricted to one agent per VM. This led to significant waiting time when just waiting for infrastructure to be available to perform a build or release, i.e., when one machine was being used, it could not be used for anything else.

Collectively, this made our prior approach to CI/CD unfit for a microservice-based architecture.

> *After only 4 microservices, we saw a serious deterioration in the performance of the CI/CD process as a whole, which meant the current model would not scale. It was not uncommon to see builds and releases queued for more than a couple of hours. Indeed long build/release wait times has been recognized as one of the barriers to the adoption of CI/CD [11], and we were suffered from this too.*

Addressing the Challenges: DevOps Principles Applied to Support DevOps!

To tackle these problems the DevOps Team decided to apply DevOps principles towards its own infrastructure and pipelines. We applied concepts such as *containerization*, *orchestration* and *infrastructure-as-code* (selected because they are well known in the field of DevOps as being beneficial) to take specific actions as part of our solution.

Action A1: Containerizing build/release agents

Containerization involves OS-level virtualization to deploy and run apps without the need for a dedicated VM on an app-by-app basis. Apps can run on the same host and underlying OS and not even know it. Therefore, if we have each build agent running in a container, it isolates each agent from another even though they run on the same operating system.

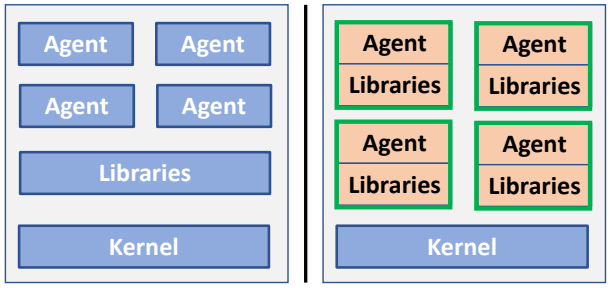

Figure 3. Non-Containerized vs Containerized Agents

The traditional approach (part a) is contrasted with the new container-based approach (part b) as shown in Figure 3. By insulating each agent from another we can safely provide parallelization, thereby addressing Challenge C4. To achieve containerization, we used *Docker* (a popular technology for containerization) and each *running container* is created from a *source image* (a container can be thought of as a run-time instance of an image).

Action A2: Using custom images for our build/release agents

Initially, we had used an out-of-the-box image provided by Microsoft to run a VSTS agent; but then soon we realized the benefits to customizing this and building an image for ourselves. Docker can build images automatically by reading instructions specified in a *Dockerfile* – a text document which contains all the commands a user could call on the command line to assemble an image. We could therefore, use the image provided by Microsoft as a basis, and then add commands to pull in the dependencies that were required to build any of our microservices, all in the same Dockerfile; such that an image is generated that can then be used by our build/release containers. This addresses Challenge C1. Now the dependencies needed to build any and all microservices only had to be specified only in one place (the *Dockerfile*) and containers spawned using the resulting image, would automatically be able to do builds/releases regardless of whether they were running on the same underlying operating system or not.

Action A3: Introducing Orchestration to manage resources

We leveraged *Kubernetes* which is an open-source platform to provide container-centric orchestration. Kubernetes manages each of the containers and introduces the abstraction of a node (which could be a VM or a physical device) which is what containers run on. Thus, Kubernetes would allow us to have multiple agents automatically managed on a single node. At the same time multiple nodes can be used to form a cluster and the maintenance and monitoring of the nodes is handled seamlessly by Kubernetes. Thus, our existing 2 build/release VMs could be re-purposed to form a 2-node cluster for Kubernetes to run the build/release agent containers on. We would have no additional infrastructure cost, addressing Challenge C3.

Action A4: Describing our infrastructure as code to handle scaling

Employing an orchestrator such as Kubernetes allows for declaratively (i.e., via a configuration file) specifying *how to manage the resources* in terms of allocation; number of instances of the container to spawn (which equates to the number of agents running); related infrastructure such as volume mounting; environment variables that needed to be set up at container launch and so on. By simply updating this configuration file (*code*) we could easily spawn more agents at will, thereby handling large build/release queue volumes, addressing Challenge C2.

Table 1 maps the challenges we faced to the solutions adopted by us that resolved these problems in our context. Once again, while interpretations and implementations may differ from organization to organization, we share what worked for us in the interests of furthering research, as well as helping other companies that may be in similar situations.

Table 1 Mapping of challenges faced to solutions adopted (actions taken)

Challenge		**Addressed by Action**
C1.	Handling microservice dependencies	A2. Using custom images for our build/release agents
C2.	Keeping build and release times low	A4. Describing our infrastructure as code to handle scaling
C3.	Keeping infrastructure costs low	A3. Introducing orchestration to manage resources
C4.	Parallelizing builds	A1. Containerizing build/release agents

Results and Observations

Herein we provide information on the actual results achieved as a result of our approach. The total number of man-hours spent is estimated to be around 720 person-hours (6 weeks X 40 hours/week X 3 full-time engineers).

Reduction of build/release queue time: We looked at data from 3000+ builds and 350+ releases of historical data (decomposition into microservices had translated into 42 different build and release definitions). The difference between the number of builds and releases is because not every build completes successfully which prevents it from making to the release phase; and to fix the issue results in at least one other build. Furthermore, check-ins into branches always results in builds, but typically only incorporation into the master-branch/trunk results in a release.

Table 2 Comparing Queue Times for Builds/Releases

	Agents		
	Hosted	Basic	Containerized
Avg. Build/Release Queue Time	1hr, 18 min	23.1 min	4.4 seconds

Table 2 presents the average time spent by a build or release in the queue, i.e., waiting to be processed – where the 'Hosted' column represents using the VSTS Hosted agent; the 'Basic' column represents using non-containerized agents; and the 'Containerized' column represents our approach with 4 containerized agents in all, managed by Kubernetes (on just 2 VMs). We see that our containerized approach significantly outperforms the hosted and basic approaches in terms of reducing the queue times for builds/releases. Additionally, to evaluate using sound statistics, we make use of the Wilcoxon signed-rank test[3] which uses the sign and the magnitude of the rank of the differences between pairs of measurements. In our context a pair of measurements means comparing the queue time using either approach for the same microservice. Corresponding to the data in Table 2 we evaluate the one-tailed alternate hypothesis that the queue time for builds/releases using our containerized approach is less than that of the hosted and basic approaches for each of the microservices. As far as interpreting the results go, a low *p*-value would indicate a high confidence with respect to the claim that

[3] The Wilcoxon signed-rank test is a good alternative to the paired Student's t-test when the population cannot be assumed to be normally distributed.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2019.2947004, IEEE Software

the containerized approach is faster (i.e., less time is spent in build/release) than the hosted and basic approaches. Per this test, the *p*-value comparing the containerized approach to the basic approach is 1.85×10^{-6} which corresponds to a confidence level of 99.999815%; and the *p*-value comparing the containerized approach to the hosted approach is 2.46×10^{-23} which is practically speaking ~100% confidence.

We emphasize that these build/release queue time metrics are not collected by us; they are instead only recorded and analyzed by us and are completely based on the data reported by VSTS (which handled the execution of builds and releases). This is an external threat to validity which may affect our ability to generalize results; but any such threats to validity also equally apply to any other studies that leverage data provided by their respective cloud providers.

> *The time spent in the queue for the Basic approach is about 330 times that of the Containerized approach, and using the Hosted agent is a whopping 1,110 times that of the Containerized approach, which translates to significant time saved.*

CI/CD is now first-class citizen in source control with its own pipeline: Per our approach we had a *Dockerfile* (consumed by Docker to build an image) and a *configuration file* (consumed by Kubernetes to manage containers at run-time). Since these are fundamentally text files we treated these as *source-code* and placed them under source control. This would allow the DevOps Team to version these files as appropriate and have a very clear history of changes - when they were made, by whom, and so on.

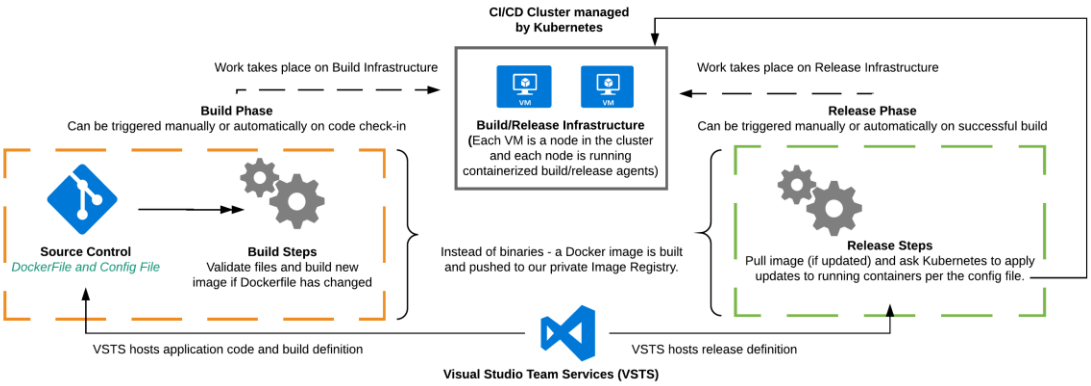

Figure 4. CI/CD Pipeline for CI/CD Infrastructure

To create a complete pipeline for this process, we then created a build and a release process associated with source control such that any changes to these files would lead to an automatic update of the build/release infrastructure. The build and release definitions exist in VSTS and a simplified form of the process is depicted in Figure 4. As seen in the Figure, with files in source control and the build/release pipelines in place, all the CI/CD infrastructure is automatically updated and managed end-to-end.

Conclusions and Future Work

DevOps is currently receiving a lot of attention and many companies are adopting best practices such as Continuous Integration and Continuous Deployment (CI/CD), as well as making the move to microservices. Despite the general adoption of these practices, many unanswered questions still exist along with barriers that software engineers face when using CI and associated tools. Automated builds/releases can sound simple at first, but require a responsible team to implement, and constant care [15]. We share our experiences with adjusting CI/CD pipelines to accommodate a microservice-based architecture and discuss the challenges faced and how we addressed them. Our aim is to increase the general knowledge-base and issue a call for a closer collaboration between industry and academia on this important topic. We believe our experience report will be beneficial to other companies in similar situations as they will suffer from the same challenges. Future work involves making the pipeline smarter and automatically scaling build agents up and down based on estimated load.

Acknowledgements

We thank Lance Brimble (former Chief Information Officer at Varidesk) for his insights regarding Microsoft Azure billing; and Bryan Dodd (former DevOps Engineer at Varidesk) for his help in building our CI/CD pipelines. Special thanks also go to the Reviewers and Editor(s) who provided excellent feedback and worked with us to iteratively improve the quality of our submitted manuscript and shape it into the paper that it is today.

References

1. D. Farley and J. Humble, “Continuous delivery: reliable software releases through build, test, and deployment automation,” Addison-Wesley Professional, First Edition, 2010.
2. M. Shahin, M.A. Babar and L. Zhu, “Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices,” IEEE Access, vol. 5, 2017, pp. 3909-3943.
3. M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs and benefits of continuous integration in open-source projects”, in Proceedings of the 31st IEEE/ACM Intl. Conference on Automated Software Engineering (ASE), pp. 426-437, Singapore, September 2016.
4. C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor and M. Stumm, “Continuous deployment of mobile software at Facebook”, in Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp 12-23, Seattle, WA, USA, November 2016.
5. A. Dyck, R. Penners, and H. Lichter, “Towards definitions for release engineering and DevOps”, in Proceedings of the 3rd IEEE/ACM Workshop on Release Engineering, pp. 3-3, Florence, Italy, July 2015.
6. C. Ebert, G. Gallardo, J. Hernantes and N. Serrano, “DevOps,” IEEE Software, vol. 33, no. 3, 2016, pp. 94-100.
7. A. Bucchiarone, N. Dragoni, S. Dustdar, S. Larsen and M. Mazzara, “From monolithic to microservices: an experience report from the banking domain,” IEEE Software, vol. 35, no. 3, 2018, pp. 50-55.
8. P. Jamshidi, C. Pahl, N. Mendonca, J. Lewis, and S. Tilkov, “Microservices: the journey so far and challenges ahead,” IEEE Software, vol. 35, no. 3, 2018, pp. 24-35.

9. W. Luz, E. Agilar, M.C. de Oliveira, and C. de Melo, “An experience report on the adoption of microservices in three Brazilian government institutions,” in Proceedings of the XXXII Brazilian Symposium on Software Engineering, pp. 32-41, Sao Carlos, Brazil, September 2018.
10. T. Mårtensson, D. Ståhl, and J. Bosch, “Continuous Integration Impediments in large-scale industry projects”, In Proceedings of the International Conference on Software Architecture (ICSA), pp. 169-178, Gothenburg, Sweden, April 2017.
11. M. Hilton, N. Nelson, T. Tunnell, D. Marinov and D. Dig, “Trade-offs in continuous integration: assurance, security, and flexibility”, in Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 197-207, Paderborn, Germany, September 2017.
12. L. Chen, “Continuous delivery: huge benefits, but challenges too,”, IEEE Software, vol. 32, no. 2, 2015, pp. 50-54.
13. D. Ståhl, and J. Bosch, “Modeling continuous integration practice differences in industry software development,”, Journal of Systems and Software, vol. 87, January 2014, pp. 48-59.
14. A. Balalaie, A. Heydarnoori and Pooyan Jamshidi, “Microservices architecture enables DevOps: migration to a cloud-native architecture,” IEEE Software, vol. 33, no. 3, September 2016, pp. 42-52.
15. M. Meyer, “Continuous integration and its tools,” IEEE Software, vol. 31, no. 3, 2014, pp. 14-16.