Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management

S.A.I.B.S. Arachchi, Indika Perera
Department of Computer Science and Engineering
University of Moratuwa
Moratuwa, Sri Lanka
saibandara@gmail.com, indika@cse.mrt.ac.lk

***Abstract*— Agile practices with Continuous Integration and Continuous Delivery (CICD) pipeline approach has increased the efficiency of projects. In agile, new features are introduced to the system in each sprint delivery, and although it may be well developed, the delivery failures are possible due to performance issues. By considering delivery timeline, moving for system scaling is common solution in such situations. But, how much system should be scaled? System scale requires current system benchmark status and expected system status. Benchmarking the production is a critical task, as it interrupts the live system. The new version should go through a load test to measure expected system status. The traditional load test methods are unable to identify production performance behavior due to simulated traffic patterns are highly deviated from production. To overcome those issues, this approach has extended CICD pipeline to have three automation phases named benchmark, load test and scaling. It minimizes the system interruption by using test bench approach when system benchmarking and it uses the production traffic for load testing which gives more accurate results. Once benchmark and load test phases are completed, system scaling can be evaluated. Initially, the pipeline was developed using Jenkins CI server, Git repository and Nexus repository with Ansible automation. Then GoReplay is used for traffic duplication from production to test bench environment. Nagios monitoring is used to analyze the system behavior in each phase and the result of test bench has proven that scaling is capable to handle the same load while changing the application software, but it doesn't optimize response time of application at significant level and it helps to reduce the risk of application deployment by integrating this three phase approach as CICD automation extended feature. Thereby the research provides effective way to manage Agile based CICD projects.**

Keywords*— *continuous integration; continuous delivery; agile project management; version management; configuration management

I. INTRODUCTION

Traditional software development methodologies are not enough to fulfill nowadays business requirements. Adaptation of Agile practices enables flexibility, efficiency and speed of Software Development Life Cycle (SDLC), which is attracted by software development companies [1]. As per Agile manifesto [2], the twelve principles have defined the integrity of process and practices and Agile Project Management, which applied on methodologies such as Extreme Programming (XP), Scrum, Kanban, Crystal, Lean Software Development (LSD), and Feature-Driven Development (FDD). Implementation of CICD pipeline on agile has enabled fast delivery of software [3] and increase in productivity. In year 2000, Martin Fowler [4] presented the idea of Continuous Integration (CI), and later J.Humble and D.Farley [5] extended these ideas into the approach of Continuous Delivery (CD) as a concept of deployment pipeline. Main benefits of CI practices are reducing the risk and make software bug free and reliable, which removes the barrier of frequent delivery. Accelerated time to market, improve product quality, improved customer satisfaction, reliable release, improved productivity and efficiency are key benefits [6] which motivates companies to invest on CD. Since most of software and mobile developments are deployed on infrastructure-as-a-service (IaaS), CICD also has become essential part of cloud computing.

A system can be well implemented and may pass holistic test cases, but it may revert to previous version, due to a performance issue which is identified after a deployment. The decision of system scale is an immediate solution for the issue, since agile process has more concern on delivery of product in committed deadlines. The current system benchmark level and new software benchmark level are two factors which define the target system scale size. System benchmarking can cause system to be unstable, since it has to evaluate on production. Also, new version should go through a load test to aware its target level. But traditional load test simulations are highly deviated from production live traffic patterns, which produces unreliable results. Therefore, how to benchmark the system with minimum impact, how to perform load test for more reliable result, and how to perform them in less effort, are the key problems address through this work.

Understanding the CICD pipeline by analyzing the existing methods, finding suitable method to benchmark the existing system, suggest mechanism to perform incremental load test by using production traffic with minimum system impact, defining a scaling factor to maintain the consistence of system performance for latest production release, automating the benchmark process and suggested load test method by integrating into CICD pipeline, and evaluating system

978-1-5386-4417-1/18/$31.00 ©2018 IEEE

performance using monitoring tools, are the main objectives of this research.

Rest of the paper is organized as follows. Next section explains the CICD concepts with related work and Section III explains the methodology of proposed solution with selection of tools. Section IV elaborates the research approach through designs and implementations. The results of these approaches are being evaluated in Section V. Finally, Section VI presents the research limitations, conclusion and future work.

II. Related Work

A. Agile Software Development to CICD

Individuals and interactions over the processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, responding to change over following a plan, are the key agile values which are motivated to follow the twelve principles defined in Agile manifesto [2]. As in the agile principles, "Satisfy the customer through early and continuous delivery of valuable software". With this CI came and extended its features with CD and combined to have CICD pipeline, which provided more efficiency and productivity on agile processes.

Delivering the software manually is one of the hardest things in SDLC, it takes more time, it needs experts in field who can handle operational tasks, mistakes are inevitable, and teams have more responsibility. CICD encourages and motivates team to deliver software frequently due to automated builds and deployments. With CICD practices leading organizations deploying 10s, 100s, or even 1000s of software updates per day [7].

B. CICD Pipeline

When an organization tries to adopt CICD pipeline, they may not be able to adopt it once. First they have to practice CI in order to adopt CD. When moving from CI to CD and then Continuous Delivery to Continuous Deployment, this pipeline has reduced the manual process execution and finally full process becomes automate. When adopt CD, other than production deployment rest of all the phases done through automation. The main difference between Continuous Delivery and the Continuous deployment is automation at production deployment.

C. Continuous Integration

CI is a software development practice where team members integrate their work regularly and automate build, test and validate. It helps to find and fix bugs quickly and improve the quality of software. Krusche et al [8] in his research work used Rugby which is an agile process model carried out using students, which CI helped to improve the code quality about 50% and it helped to find and fix broken commits faster more than 65%. About 70% of students have claimed that CI helped them to improve overall development workflow. The source repository, version control system and CI server are the main components of the CI. As per Amazon Web Services (AWS) in [9], more frequent commits to common codebase, maintaining single source repository, automating builds and automating testing are mentioned as challenges when following CI practices. As in [10], Build automation, Code stability, Analytics, CD enablement, faster releases & cost saving, improved productivity & code quality are mentioned as realized benefits when adopting CI practices.

D. Continuous Delivery

According to Humble [5], Continuous Delivery is the ability to get change of all types including new features, configuration changes, bug fixes and experiments into production or into the hands of users, safely and quickly in a sustainable way. In [8], Krusche has introduced CD into multi customer project courses and evaluated its usage, experience and benefits. As per Chen [6], there is a rapid trend in investment on CD due to its benefits such as improves productivity and efficiency, reliable releases, improves customer satisfaction, accelerated time to market and making the right product.

E. Continuous Deployment

As per AWS [9], the CD means every change committed is ensuring production ready and Continuous Deployment applies them in production automatically. As in [7], [11], [12] and [13], nowadays many researches used continuous deployment automation approach to efficient their work. Rahman et al [14] points that continuous deployment has speed up the processes in agile methods noting Facebook, GitHub, Netflix and Rally Soft as organizations that use continuous deployment efficiently on their production deployments.

F. CICD Tools

CICD tools can be classified as Repository and Version Control tools, Build tools, Automation tools, Test Automation tools, and Monitoring tools.

1) *Repository and Version Control Tools*

Version Management is defined as managing the changes. Due to the support of distributed architecture and being open-source software, Git has become more popular and widely use version controlling system with CICD approaches [8], [9], [14], [15], [16], [17].

2) *Build Tools*

Ant, Maven and Gradle are famous builds tools available under open source. CI servers provide centralized control over build tools and as per Shahin [15], Bamboo, Jenkins, Cruise Control, Hudson, Sysphus, Hydra, TeamCity are few available CI servers. But in CICD approaches like [9], [10], [14] and [16] has used Jenkins as their main CI server, due to its features such as customizable, scalable, cross platform support and security.

3) *Automation Tools*

Configuration Management (CM) is process of ensuring consistency in infrastructure, which is the key objective of automation with benefits such as cost reduction, reduce the complexity of release process, proper host management, and easy configuration, reliability, and efficiency and optimize resource utilization. CFEngine was the first modern open source CM tool released in 1993 [18]. Google has given a

solution for "Managed Compute Engine" in their cloud platform in [19] using CM tools Puppet, Chef, Salt and Ansible. In [9], AWS has mentioned that Puppet, Chef, Salt and Ansible as CM tools which should be skilled by development teams in automation. In [20], Ansible is selected over puppet and chef as it is the fast and simplest solution for the configuration management.

4) *Test Automation*

The idea of "test automation pyramid" in agile process was introduced by Mike Cohn [21]. This pyramid consists of three different levels as Unit, Service and UI. AWS [9] has introduced another layer called Performance/ Compliance on top of service test layer. Test automation orchestration approach in [22] described deployment pipeline with Unit test, functional test, Sniff test and Performance test phases. The continuous testing approach described in [12] has benefit of stable code base, faster response and easy decision making.

5) *Monitoring*

Khan et al [23] mentioned the monitoring benefits on developers and organizations and Aceto et al [24] mentioned that cloud systems are mostly benefitted by monitoring. There are various commercial available monitoring tools such as Monitis, Reveal Cloud, Cloud Harmony, Nagios, Nimsoft, Cloud watch, Open Nebula, Logic Monitor and many more as per [23].

III. Methodology

In this approach CICD is used as main deployment process pipeline methodology. When deploying a system where performance is concerned, the deployment should always consider a rolling of two levels. First level is that deploying to test-bench environment and evaluates the new development, and then second level is for completing production deployment. The main benefits of deploying in two levels are early identification of performance issues in latest development and then fix those issues. When existing system already in optimized level as per team capabilities, team may try to evaluate the option of scaling. Fig. 1, shows the deployment scaling solution with CICD, which is proposed by this research work. According to it, benchmark, load-test, scale and provisioning are four iterative phases which should follow with deployment scaling. In the benchmark phase, it checks the production capability and limitation, and then identifies what is the current benchmark level of production. When defining the scaling requirement, it uses current benchmark level. The new software will proceed through load test phase, to check whether it can perform same as production or to see any performance issue. The scaling factor is used to check whether new software requires additional nodes or not, by using the benchmark levels and the load test result took at early phases (it does not scale, but it identifies the requirement of scale); if it requires new servers, and then it will provision them in next phase. Therefore, the provisioning is not a mandatory phase. The outer boundary circle is for deployment and it is denoted in dotted lines as there can be several deployment methods, which can be applied on this process.

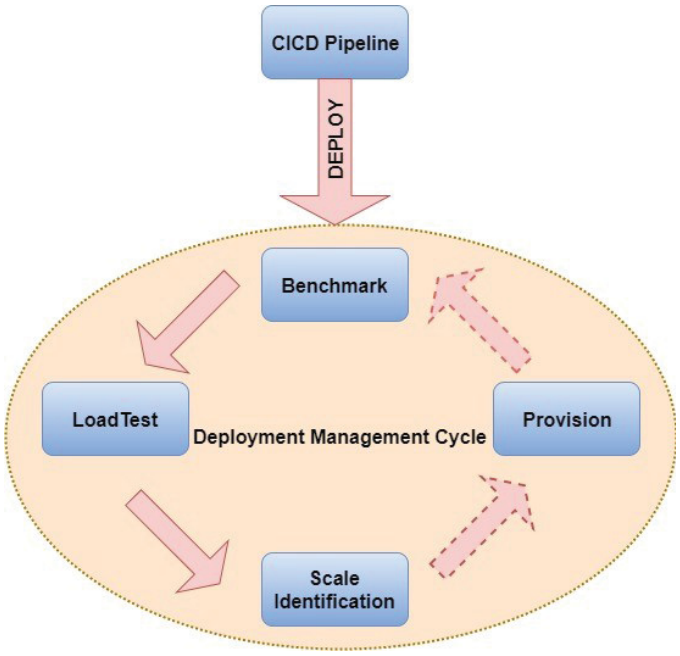

Fig. 1.The deployment management cycle with CICD

When the system has a trend of having peak seasons, this iterative approach is added advantage to perform an elastic scaling based on system load demand, which is more cost effective in cloud systems. Introducing of new level before production release may consider as additional overhead but having proper automations in CM tools with CICD pipeline can save time and risk of deployment failure and make sure having stable production release.

A. Deployment Methods

Feature flags, Dark Launches, all at once, rolling, in place – Doubling, Canary, Immutable Deployment, and Blue-Green Deployment are deployment strategies mentioned by Ville [25] and AWS [9]. When analyzing their features, it is found that all are applicable with CICD approach but rolling method is selected since it is the simplest and widely used method.

B. Benchmark

Benchmark shows the relative performance of one system which was accepted as better performs based on business and other technical requirements. It should be repeatable on each delivery, should be performed in identical environment, and it should be isolated while it performed. In [26], Li mentioned several benchmark methods, performance metrics and relative approaches such as transaction speed, availability, latency, reliability, throughput, scalability and variability. In this work, latency and throughput are considered as metrics on benchmarking, while increasing the system load.

1) *Duplicate Traffic*

To have same production traffic pattern and load, Gorepay has been used which is capable of duplicate traffic without impact on host server. Goreplay does capture the web traffic, it can replay traffic from one server to another, and also it can save traffic to a file and replay them in target server later.

C. Load Testing

Load testing identifies the new software limitations and it benchmarks the existing system. Load testing, stress testing, and spike testing are types of performance testing as per AWS [9] and they are used for benchmarking under predefined criteria. GoReplay is used for traffic duplication from production to have identical traffic pattern and Ansible automation is used to control process.

D. Scale Indentification

The over-scaling is accepted for some extent, but it suffers from low resource utilization and under-scaling is unacceptable since it causes system instability. Therefore, identification of scaling requirement is an important factor. By using the load at benchmark and the load testing the scaling requirement can be calculated as; Z is current benchmark load per server, Z1 is maximum load handled in load test. The total load of the system is (Z x N) where N is number of nodes. The load handled by new version in existing nodes is (Z1 x N). Therefore, additional server requirement (Zx) is:

$$Zx = (N\ (Z — Z1)) / Z1 \quad (1)$$

Since number of servers cannot be a decimal value, node requirement should always return next (round to ceil) integer value from the result.

E. Provisioning

When scaling identification phase states that system needs to be scaled, then required amount of servers provisioning will be initiated by Ansible automation. In this approach automation may use vagrant for virtual machine load which is the simplest way of creating lightweight pre-boot virtual servers for development environment.

IV. Implementation

The CICD pipeline is a collaboration of various tools and services which perform as one unit. In order to implement the pipeline at initial phase, Git is used as version control and source code management system, Nexus is used as repository for executable files, and Jenkins is used as CI server. To avoid network dependency in production and local, additional cloud nexus repository can be setup since production is in cloud. Ansible automation Jobs are added to CI server to initiate deployment to staging and production environments. At deployment, the Nagios monitoring is used to monitor abnormal behavior of the process. The automation process can be categorized as benchmark, load testing and scaling.

A. Benchmark Automation

The benchmark automation process is initiated by CI server and automation progress can be displayed through Jenkins job process output. It requires new benchmark server which is identical to production server which is provisioned in same network environment. As mentioned earlier, Go-replay has been used for traffic duplication purpose which is controlled by Ansible automation and Nagios is used for monitor behavior on benchmark server. Since go-replay is able to duplicate traffic at percentage level from host, it gives more clear idea of amount of load which breaks the system. The benchmark automation process can be listed as below:

1) Provision new server with homogeneous specs with same capacity as benchmark server.
2) Deploy current production version into benchmark server.
3) Initiate Go-replay from pre-selected app server list.
4) Provide initial traffic from App1 starting with 5% load and gradually increase it with addition of 5% in each step with a gap of 5 minutes.
5) Check the Nagios monitor status for latency of output result to see it reaches defined expected maximum latency while service is available.
6) If benchmark server can perform when App1 traffic emission reach to 100%, then automation starts traffic duplication as in step 4 by selecting next App server to increase traffic load.
7) Continue steps 4, 5 and 6 until benchmark server breaks it limitation of latency or its service get unavailable.
8) Stop the traffic from all App servers when benchmark has reached its limit, and the load breaks the server limit is the benchmark level of current production server.

B. Load Test Automation

Once the benchmark automation is completed, the load test automation can be initiated by using same benchmark server or creating a new server by cloning a production as Load test server. It has to repeat same except in step 2, instead of existing live version, it is used new development version, and mark final load.

C. Scaling Automation

Once identified, the production benchmark load and new development load, scaling requirement can be calculated as in (1). The scaling automation will deploy existing version on new servers and monitor, then later new version release has to be done for all servers. Scaling automation steps are listed as below:

1) Calculate new server requirement as in (1).
2) Provision new servers with homogeneous specs with same capacity
3) Deploy current production version into new servers
4) Adding servers to load balancer with equal weight
5) Enable live traffic

V. Evaluation

A. Test Bench Setup

Test bench automation contains five modules such as server provisioning, Nagios setup, Benchmarking, load testing and scaling. Though cloud virtualization is one of the evaluation environment options, in this approach the test bench is setup in local high capacity computer with local virtual instances. Initially, CICD pipeline is implemented followed by test bench

setup. The test bench setup is categorized into three phases as benchmark, load test and scaling. As in Fig. 2, four application servers are provisioned including nagios nrpe client tools and provisioned separate nagios master server to monitor system, and APP4 is selected as benchmark/load-test server. The HA proxy was used as load balancer in testbench-lb server and Jmeter was used to generate load on load balancer. Through automation, Go-replay initiated traffic duplicate from APP3 to APP4 then increase the load as in section 4 A. In Fig. 2 shows the test bench setup for load test, and when benchmarking, APP4 contains the production version instead of new version.

The application used in test bench approach is simple XML based application. It does not use any database system to reduce the complexity of the process and used two different logics to represent production version and new version.

B. Results and Performance Analysis

Test bench initiated with benchmark phase and each phase is performed 20 minutes time frame. Nagios graphs, Jmeter response time diagram and proxy traffic statistics were used to analyze the performance of the system.

1) *Benchmark Phase*

At the benchmark the response time was around 55ms-72ms. Fig. 3 (area marked as 1) shows the CPU pattern of the benchmarking. When it had 100% load for initial time frame, it performed below critical level but when increased the load, it reached beyond critical level and system became unstable while increasing the response time. The load with marked at load balancer at 100% traffic is 15254 and 5085 per node.

2) *LoadTesting Phase*

In load testing, the new version started to respond between 70ms-80ms which is higher than initial benchmark level. In Fig. 3, (area marked as 2) shows the load testing CPU pattern, and it is slightly higher than benchmark initial time frame. In this phase, it could be able to process only 12351 (4117 per node) requests in given time frame.

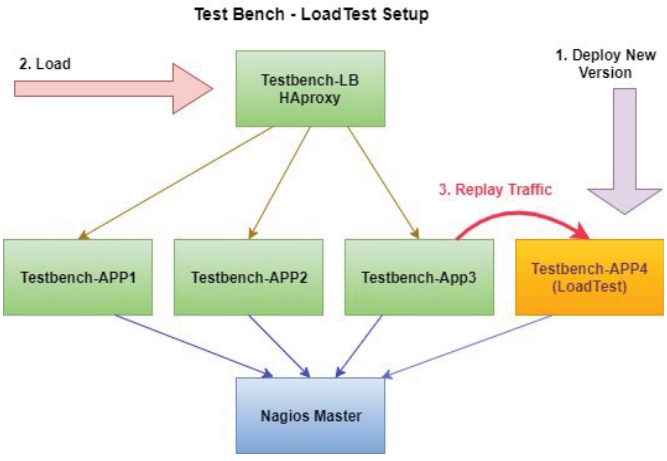

Fig. 2. Test bench – load test setup

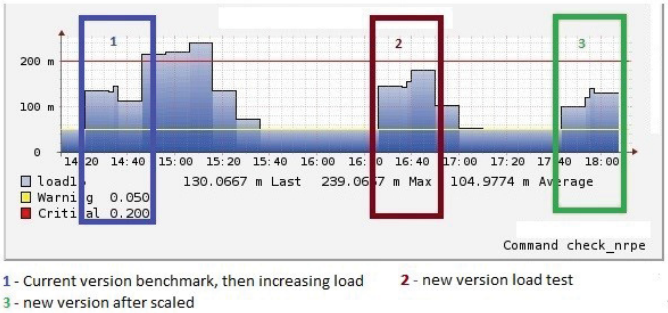

Fig. 3. Test bench – nagios CPU load pattern

3) *Scaling Phase*

Scaling Requirement calculation using (1):

$$
\begin{aligned}
Zx &= (N\,(Z - Z1))/\,Z1 \\
Z &= 5085 \qquad Z1 = 4117 \\
N &= 3 \text{ (Three APP servers one node in each)} \\
Zx &= (N\,(Z - Z1))/\,Z1 \\
&= (3\,(5085 - 4117))/\,4117 \\
&= 0.71 \Rightarrow 1
\end{aligned}
$$

As per the scaling requirement calculation, it needs one more addition server to serve same load. After scaling, as in Fig. 4, the response time was more towards to 70ms which is higher to initial benchmark but slightly better than load testing. At this phase, it could process 15644 total requests and the CPU pattern is much better than other phases since per server load is reduced as in Fig. 3 (area marked as 3). The addition of new server has gained the benefit of consistent throughput, but it does not show much improvement in response time in system. Therefore, the test bench approach has proven that, when the software is changing, the system is enabled to process same load through scaling automation.

VI. Conclusion

A. Conclusion

CICD has made efficient agile delivery process as well as improved the productivity of the system. The proposed extended feature on CICD pipeline has achieved goals through test bench approach. It enables continuous benchmark of each software version and go-replay has given access to same live traffic on bench mark and load test phases.

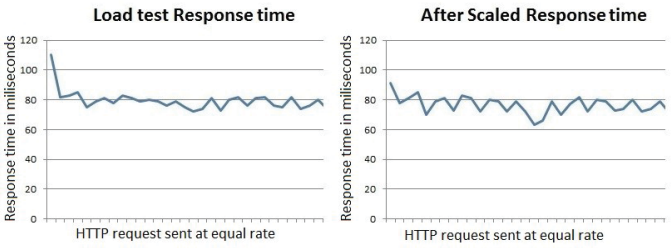

Fig. 4. Load test and after scaling response times

Ansible automation has enabled to perform test bench approach more efficient and addition of benchmark, load test and scaling phases on CICD has not introduced extra effort after initial setup. Test bench approach has proved that after scaling system could process same or better load with new development.

B. Study Limitations

In this approach, simulated test data was not enough to cover all aspects, and application used cached data from local file instead of data from database system. Infrastructure was limited, as working on local virtual instance which bounded by computer capacity. Max memory used 2.5GB and only single core instance is created. Due to that reason maximum APP servers limited to 4, could not see any cluster behavior, and used simple rolling deployment strategy due to limited servers.

C. Future Work

The test bench approach can be setup in cloud virtual servers and evaluate with enough infrastructure for realistic results. The impact of application level cache on performance measurements can be evaluated by extending this approach for accurate scaling. It has considered response time on given load as main key performance parameter, but parameters such as CPU, memory, threads and IO can be used for calculation of scaling requirement which can predict more accurate scaling value. This work could be extended to create self-healing elastic cloud approach.

REFERENCES

[1] N. D. Fogelström, T. Gorschek, M. Svahnberg, and P. Olsson, "The impact of agile principles on market-driven software product development", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 1, pp. 53-80, 2010.

[2] W. Cunningham, "Principles behind the Agile Manifesto", *Agilemanifesto.org*, 2017. [Online] Available: http://agilemanifesto.org/principles.html. [Accessed: Sept. 2017]

[3] H. H. Olsson, H. Alahyari, and J. Bosch, "Climbing the "stairway to heaven"-a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," In Proc. *EUROMICRO conference*, 2012, pp. 392-399.

[4] M. Fowler, "Continuous Integration", *martinfowler.com*, 2017. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html. [Accessed: Sept. 2017].

[5] J. Humble, and D. Farley, *"Continuous delivery : reliable software releases through build, test, and deployment automation"*, 1st ed. Addison-Wesley Professional, 2010.

[6] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too", *IEEE Software*, vol. 32, no. 2, pp. 50-54, 2015.

[7] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at facebook and oanda," In Proc. 38th International Conference on Software Engineering Companion, 2016. pp. 21–30.

[8] S. Krusche and L. Alperowitz. "Introduction of Continuous Delivery in Multi-Customer Project Courses," In Proc. of ICSE'14. IEEE, 2014.

[9] Amazon Web Services Inc*, "Practicing Continuous Integration and Continuous Delivery on AWS,"*2017 [Online]. Available: Amazon Web Services, https://aws.amazon.com/ [Accessed: Sept. 2017].

[10] A. Kumbhar, M. Shailaja, and R. Anupindi, "*Getting started with Continuous Integration in Software Development*", 2015 [Online]. Available:Infosys Limited, https://www.infosys.com [Accessed: Sept, 2017].

[11] T. Lehtonen, S. Suonsyrjä, T. Kilamo, and T. Mikkonen, "Defining metrics for continuous delivery and deployment pipeline." in *SPLST*, 2015, pp. 16–30.

[12] C. Dunlop, and W. Ariola, "DevOps: Are You Pushing Bugs to Clients Faster?", In *Proceedings of PNSQC* 2015.

[13] N. Dragoni, S. Dustdary, S. Larsenz and M. Mazzara, *Microservices: Migration of a Mission Critical System*. eprint arXiv:1704.04173, 2017. [Online] Available https://arxiv.org/abs/1704.04173 [Accessed: Sept, 2017].

[14] A.Rahman, E. Helms, L. Williams, and C. Parnin. "Synthesizing continuous deployment practices used in software development", In Proc. Agile Conference (AGILE), 2015, pp.1-10.

[15] M. Shahin, M. Ali Babar and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices", *IEEE Access*, vol. 5, pp. 3909-3943, 2017.

[16] "Adopting Continuous Delivery" in *Continuous Delivery Whitepaper*. [Online] Available: Levi9 IT Services, 2016 [Accessed: Sept, 2017].

[17] D. G. Feitelson, E. Frachtenburg, and K. L.Beck, "Development and Deployment at Facebook," in *IEEE Internet Computing*, vol. 17, pp. 8-17, July - August, 2013

[18] V. Hardion, D. Spruce, M. Lindberg, A.M. Otero, J.Simon, J. Jamroz, and A. Persson, "Configuration Management of the control system". *in* Proc. *ICALEPCS2013,* San Francisco, 2013

[19] "Compute Engine Management with Puppet, Chef, Salt, and Ansible | Solutions | Google Cloud Platform", *Google Cloud Platform*, 2017. [Online]. Available: https://cloud.google.com/solutions/google-compute-engine-management-puppet-chef-salt-ansible. [Accessed: Sep- 2017].

[20] F. C. Liu, F. Shen, D. H. Chau, N. Bright, and M. Belgin, "Building a research data science platform from industrial machines," In Proc. 2016 IEEE International Conference on Big Data (Big Data), Washington, DC, 2016, pp. 2270-2275.

[21] M. Cohn., *Succeeding with Agile*. Pearson India, 2015.

[22] S.G.Gaikwad, and M.A. Shah, "Pipeline Orchestration for Test Automation using Extended Buildbot Architecture", in Proc. Confernce on Emerging Applications of Electronics System, Signal Processing and Computing Technologies (NCESC 2015*)*, 2015.

[23] K. Alhamazani, R. Ranjan, K. Mitra, F. Rabhi, P. Jayaraman, S. Khan, A. Guabtni, and V. Bhatnagar, "An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art", *Computing*, vol. 97, no. 4, pp. 357-377, 2014.

[24] G. Aceto, A. Botta, W. de Donato and A. Pescapè, "Cloud monitoring: A survey", *Computer Networks*, vol. 57, no. 9, pp. 2093-2115, 2013.

[25] V. Pulkkinen, "Continuous Deployment of Software", *in Proc. Of the seminar no.58312107: Cloud-based Software Engineering*. University of Helsinki, 2013, pp 46-52.

[26] Z. Li, L. O'Brien, H. Zhang, and R. Cai, "On a catalogue of metrics for evaluating commercial cloud services", in Proc. of the ACM/IEEE 13th International Conference on Grid Computing (GRID), 2012, pp. 164 – 173.