OPEN ACCESS Journal of Software Engineering

ISSN 1819-4311
DOI: 10.3923/jse.2018.12.19

Survey Article
Toward a Reference Model for Adopting Software Continuous Delivery: A Practical Approach

Salem S. Bahamdain and Basem Y. Alkazemi

College of Computer and Information Systems, Umm Al-Qura University, Makkah, Saudi Arabia

Abstract

Releasing a new version of a software system is a relatively complex process that requires lengthy and complicated preparation and coordination between different role-players of the development and operation teams. The difficulty originates during the early stages of gathering requirements and continues to the stages of software deployment and maintenance. Enhancing the software development process and overcoming some of the obstacles encountered by developers can be achieved through the utilization of continuous delivery (CD) practices. This analytical study investigated the currently available CD implementations in industry to identify the key characteristics of CD from a practical perspective and to evaluate them based on the commonly defined characteristics in the literature. A number of solutions were examined and a taxonomy that highlighted the similarity and variations in every solution was generated. Subsequently, a discussion of the requirements needed for a CD reference model was presented, which represents the key contribution of this work. The proposed analytical framework was evaluated in terms of a taxonomy that was generated in this work. The key characteristics of a CD model were highlighted at both the conceptual and implementation levels in light of the currently available technologies and the overall results obtained favored the assertion made in this work regarding the essence of generating a standard reference model for CD. The experiment also identified the common underlying components of a CD model, which were believed to be essentially the building blocks for the successful implementation of a CD model. The model proposed in this work may act as a baseline for defining a reference model for CD in practice. Software development organizations can refer to this model for the implementation of their underlying delivery pipelines. This model was investigated thoroughly in the context of web-portal development environments.

Key words: CD model, continuous integration, web-portal architecture, continuous delivery, software delivery pipeline, CD implementations

Citation: Salem S. Bahamdain and Basem Y. Alkazemi, 2018. Toward a reference model for adopting software continuous delivery: A practical approach. J. Software Eng., 12: 12-19.

Corresponding Author: Basem Alkazemi, Umm Al-Qura University, Al Awali, Makkah 24381, Saudi Arabia Tel: +966 21 5270000

Competing Interest: The authors have declared that no competing interest exists.

Data Availability: All relevant data are within the paper and its supporting information files.

INTRODUCTION

In competitive industrial environments, the rapid delivery of a workable release shows the essence of progress to stakeholders and fosters trust among bidders. Chief information officers (CIOs) are consistently required by their company's chief executive officers (CEOs) or stakeholders to demonstrate certain workable features of the system under development, even during the very early stages of a project's lifetime. This sudden request to deliver a workable release of a system is beyond the capability of today's software development life cycles (SLDCs)[1], based on current practices. Thus, the concept of software continuous delivery (CD) was coined to handle such emerging situations. The CD is the process of delivering workable releases of the system under development at any instant of time during the project lifetime, as described by Fowler[1]. A number of software development organizations have tried to adopt the notion of CD in their development practices while complying with the guidelines prescribed by Humble and Farley[2]. For instance, Zend Technologies Inc.[3] defined the Zend Blueprint for CD as a design pattern that identifies the best practices for each phase of the development cycle. This company highlighted some components for implementing their CD pipeline, including code version control, code management policies and CI. XebiaLabs Company[4-6], on the other hand, identified some common practices that should be followed by CD adopters without defining the low-level components for implementing CD. Their main focus was on developing automation tools to help software deployment and delivery. Duvall[7] and DZone Inc.[8] highlighted numerous challenges in the adoption of CD that can hinder organizations in their attempts to move in that direction, including organizational and cultural change challenges. Moreover, presented a guide to CD and DevOps implementation and management with a focus on CD best practices, automation and CI.

Despite the availability of somewhat few attempts to adopt CD among the software industry, there seems to be some disagreement upon defining the underlying components of CD practices by software development organizations[9], which resulted in different implementations for CD. It is believed that the variation among vendors is attributed to the lack of a precise documented implementation in terms of defining all the underlying components needed by a CD model. Hence, it is not possible to determine the optimal approach for those who might need to utilize the capability of CD and investigate its applicability for their development environments in a vendor-neutral manner.

Therefore, this work aimed to discuss and highlight the low-level implementation details of CD practices that adopters should consider in their development environment. This work was built on the definition of CD by Humble and Farley[2] and attempted to complement their work by identifying the underlying components and software modules in light of their definition for CD. The identification of CD components was reported in this work after both reviewing some of the prominent adopters of CD in the web-portal development industry and generating a corresponding taxonomy of the characteristics and modules their solutions exhibited. In this study, the web-portal application domain was specifically considered because it is the main area of interest for the authors of this work who are working as the key investigators of a migration project to enhance the web-portal development environment at Umm Al-Qura University (UQU)[10] by adopting CD practices. The outcome of this work will help UQU establish a baseline for implementing CD practices. Eventually, it is believed that the highlighted pipeline components identified in this work might also help software developers and decision makers in a similar application domain to understand how CD practices, as a model, can be adopted in their environments to gain the real value and fulfill the benefits that Fowler[1] described regarding CD. This work is a step toward developing a standardized reference model for adopting and implementing CD by software development organizations.

REQUIREMENTS AND CHARACTERIZATION OF CD

Requirements to adopt CD: A number of requirements were highlighted in the literature and by several vendors about the essence of CD. These requirements include the following:

- **Avoiding the ad hoc delivery of releases:** In some cases, a project sponsor or customer might request to see a workable release of the system under development. In other cases, a project leader might be interested in assessing the productivity of their team by requesting the delivery of a prototype within a certain period of time. This assessment requires the system to be built in such a way that it can be delivered instantly without affecting the flow of the project
- **Lowering development costs:** It is commonly known that software development is a costly process because it involves considerable management and organizational work to arrange tasks and pipeline production, in addition to the technicality of system development. Thus, it is necessary to establish the development environment

in a flexible manner to accommodate changes and effectively facilitate management and follow-ups

- **Facilitate bug tracking:** Any software development project involves a stage of testing and troubleshooting. In some cases, fixing a bug at a later stage of development might impact other core components in the system. Thus, it is necessary to ensure that bugs are tracked and to monitor the testing activities in a regular manner to identify and fix bugs as soon as possible. It would be beneficial to implement an automated method that can raise warnings in the case of a defect or bug, which allows detection at any stage of the process
- **Better control of the development cycle:** The software development cycle contains different stages, starting from early analysis to the later stages before system delivery. Each stage might involve a considerable number of members or even multiple teams. At some stages, it might be difficult to monitor their progress, particularly in case of requirement changes, as the team will be required to accommodate these changes and reflect them in the system that they have already developed. Therefore, a method of visually monitoring the overall process at various stages will help to manage any complexity at any point in time

Based on the aforementioned requirements, the characteristics that a CD model needs to be fulfilled.

Characterization of CD: The defined characteristics are selected based on a review of the available implementations of CD in the literature and some available CD resources. Accordingly, the characteristics of CD have been categorized as follows:

- Environment structure and organization
- Automation
- Continuous integration (CI)
- Visibility and Quality assurance (QA)

Environment structure and organization: One of the most important requirements for adopting the CD approach is that some of the development practices within an organization must be changed to reflect all the CD principles in the environment. The practices include the following:

- **Support partial planning:** This change can start during the planning phase of the software; partial planning is conducted, rather than a complete plan being drafted to release the full features of the software system. The planning phase can be expanded continuously by releasing a simple workable version of the software to obtain customer feedback. The feedback may change the current plan to reflect real user needs and to make it possible to release frequent small incremental updates
- **Support agility:** The software development life cycle must be transformed from a traditional software development life cycle to Agile Scrum or Kanban to achieve flexibility, productivity and faster releases
- **Adopt source control:** There are numerous aspects to consider in the development process, such as source control management (GIT, SVN). Every change in the source code created by the developer should be committed to source control
- **Branching:** Branching is recommended in source control to manage a new feature or fix a bug. Developers should commit their work frequently and those commitments should not break the software
- **Like-production environments:** Production-like environments should be built, such as testing environments to run the automated tests and staging environments to perform manual user acceptance tests (UAT)
- **Flexible software system architecture:** Another important aspect of development is the software architecture, which must be sufficiently flexible and testable to facilitate small incremental releases. There are numerous different patterns that can be implemented, such as object ordinated programming (OOP) and the model-view-controller (MVC) pattern. Feature toggling can also be considered and previous aspects can vary from one software type to another
- **Adopt the DevOps mechanism:** To ensure that software is released rapidly, frequently and more reliably, the development team, QA team and IT operation team should communicate and collaborate as one team to eliminate difficulties during the process from development to production by automating all the processes of software delivery and infrastructure changes
- **Self-organized team:** The team should be self-organized, enabling it to solve and deliver any tasks to production. Additionally, the team should monitor the software and make all metrics visible to everyone, such as the delivery pipeline status, application performance monitoring (APM) and infrastructure monitors

Automation: The CD is mainly focused on automating the processes involved in system development, including building, packaging, testing, deploying the software system and infrastructure automation (infrastructure provisioning). These processes should be represented in a specific scripting language to enable the automation of the entire process in the delivery pipeline. The key principle behind the adoption of CD practices is to facilitate automated testing on the various stages of system development. Automated testing improves the code integration status visibility on every commitment by developers by providing rapid feedback about the status of tests (whether they pass or fail) after each new code is pushed to source control.

There are numerous types of automated and manual tests and there are also different tools used for their implementation. One of the most important tests is the unit test, which can be implemented in the hypertext preprocessor (PHP) including the programming language framework PHPUnit and the Java language by JUnit. There are different types of automated and manual tests, some of which are listed below, along with examples of the tools that can facilitate their implementation:

- Unit test (PHPUnit, JUnit)
- Acceptance test (Codeception, Selenium)
- Integration test (PHPUnit, JUnit, Jenkins)
- Functional tests (PHPUnit, JUnit, Selenium)
- Visual User Interface (UI) tests (Selenium, Manual)
- User Acceptance Testing (UAT) (Selenium, Manual)
- Usability tests (Selenium, Manual)
- Network tests (Ping, Wireshark)
- Performance tests (LoadRunner, Gatling)
- Security tests (Nessus, OWASP ZAP, BDD-Security, Burp)

The necessity to implement a test type depends on the software type and organizational policies, but each one is important to partially fulfill the requirements of CD where all the testing must be automated.

Continuous integration (CI): The core of CD is CI. The CI establishes the backbone of the delivery pipeline, which runs and manages all the automated tasks for building, testing, packaging, deploying, rolling back in case of failure and any additional tasks needed by the software system. These activities can be defined and managed by a number of CI servers, such as Jenkins, Bamboo, TravisCI, SnapCI and TeamCity. The CI server executes multiple tasks representing a single stage that can be combined with other stages to establish a CD pipeline. The CD pipeline can visualize the building, testing and deployment status of the software under development for the team. Additionally, the CD pipeline facilitates the deployment of a workable release at any time and whenever needed. The pipeline and its corresponding tasks can include the following stages:

- **Build:** This stage can be triggered automatically after each commitment to source control by developers and the code can be cloned. The build stage may include tasks such as checking syntax errors, compiling the code, installing software dependencies, bashing the script to prepare permissions and optimizing files
- **Test:** In this stage, all automated tests are executed and a code coverage report of the tests is generated
- **Quality Assurance (QA):** At this stage, the software quality reports can be generated, such as a code style check, duplication code, mess code detection, code dependences, open tasks in code comments and lines of code (LOC)
- **Packaging and Deployment:** Software is packaged as needed and then associated by issuing a new version number to deploy a version of the software system to all different environments, such as the testing, staging and production environments. If a failure occurs in the deployment process, the stage should trigger a task, which automatically rolls back to the latest working version of the software

Visibility and quality assurance (QA): Visibility and QA are important aspects of CD that provide high-quality software and can visually indicate if any failures have occurred in the build, tests, packaging, or deployment processes. The team should review the reports generated by the CI pipeline (test coverage, mess detection, code style, duplicate code, automated code documentation) to improve the software. After the software is deployed for production, various components should be monitored, including the infrastructure layers running the software, such as servers, networks, databases and the load balancer. Furthermore, monitoring the application layers, i.e., APM, such as the response time of the application, memory usage, database transactions, queries performance and exceptions that occur to the end users, is vital for the successful implementation of CD. All monitors, dashboards and QA reports should be visible to all team members and stakeholders if necessary.

SURVEY FOR ADOPTING SOFTWARE CONTINUOUS DELIVERY (CD)

A semi-structured survey was designed based on Usman *et al.*[11] methodology and conducted to generate a taxonomy of the solutions for the identified CD characteristics. The related studies have been surveyed based on published articles and white papers. Four web-portal development organizations were selected for the investigation in this work because they were, at the time of this research, pioneering establishes of web-development environments in the software industry. The selected list is not intended to be exhaustive as those four vendors were specifically explored to demonstrate the methodology of the analysis followed by this work and the subsequent evaluation, which can be applied as an evaluation framework for a wider range of vendors in the future if sufficient details regarding their approaches were provided. Hence, it is believed that the selection made was adequate to validate the work at this stage.

The surveyed works were then analyzed in light of the key characteristics proposed by Fowler[1] in addition to the other related documents and manuals in the literature as a step to identify the common characteristics of CD at the conceptual level. Afterwards, low-level implementations were explored with a number of prominent web-portal solution providers in the industry that are recognized as CD adopters. The components of their solutions were investigated and highlighted. Finally, a taxonomy of the CD characteristics was generated to describe the similarity and variations in the surveyed work. The taxonomy has paved the way to identify the most appropriate components of a CD solution that an organization in the web-portal development domain needs to implement to successfully adopt full CD development practices in their organization. Based on this analysis and investigation, the core characteristics of CD practices were identified as a step toward establishing the foundations for defining a reference model for CD in practice.

Proposed Analysis: The proposed analytical framework was evaluated against a number of CD implementations reported by four web-development solution providers. A taxonomy of the various solutions was generated to identify the similarities and variations between the experimental samples. The purpose of this taxonomy is to validate the identified

Table 1: Taxonomy of the CD models

Characteristics	CD Book[2]	Zend[3]	XebiaLabs[4-6]	Duvall[7] and DZone[8]
Organizational culture				
Agility	✔	✔	✔	✔
DevOps	✔	✔	✔	✔
SCM and frequently commits	✔	✔	✔	✔
Flexible software architecture	✔	-	-	-
Feature toggling	✔	-	-	-
Continuous planning	✔	✔	✔	✔
Self-organized team	✔	✔	✔	✔
Automation and continuous integration				
Staging/test environment	✔	✔	✔	✔
Maintained databases	✔	✔	✔	✔
Automated unit test	✔	✔	✔	✔
Automated functional test	✔	✔	✔	✔
UAT	✔	✔	✔	✔
Smoke test	-	✔	✔	✔
Infrastructure automation	✔	✔	✔	✔
Server triggered	✔	✔	✔	✔
Automated build and compile	✔	✔	✔	✔
Release automation	✔	✔	✔	✔
Automated DB	✔	✔	✔	✔
Automated rollback	✔	✔	✔	✔
Visibility and quality assurance (QA)				
Build status visibility	✔	✔	✔	✔
Auditing	✔	✔	-	-
Code coverage	-	✔	✔	✔
Analysis of code quality	✔	✔	✔	✔
APM	✔	✔	-	-
Infrastructure monitors	✔	✔	-	-

characteristics to establish a base for the definition of common CD characteristics, hence leading to the definition of a standard reference model for CD in the future. Table 1 lists four key corporations based on the observations that were commonly recognized as CD adopters in their development environments.

The investigation had highlighted the essence of standardizing CD as a development process model to help developers, practitioners and even educators thoroughly understand CD. Consequently, CD can be treated as a standard development model in the same manner as other software process models defined by software engineering standards.

Hence, this work had highlighted the conceptual characteristics that a CD model must exhibit. In addition to the building blocks of the CD pipeline, it is believed that the characterization presented in this work can help to establish a common reference model for CD. Potentially, a maturity model can subsequently be developed to measure the readiness of organizations to adopt CD and the maturity levels they have reached.

Generally, it was observed from Table 1 that the different CD approaches available in the literature agreed conceptually on the importance of having many of the identified characteristics and components of CD proposed in this work. Nevertheless, vendors tend to interpret their implementation of the characteristics differently, which has slightly affected the other components of their pipelines. Some variations are observed in the taxonomy that has been generated in this study.

Although a significant characteristic of CD, feature toggling was not covered by the other three solutions, namely, Zend Technologies Inc.[3], XebiaLabs[4-6], Duvall[7] and DZone[8]. It is believed that this feature would necessitate the high modularity of the system's components, which could be relatively expensive to develop. With regard to the QA dimension, it was observed that XebiaLabs[4-6] and DZone[8] lack auditing, APM and infrastructure monitoring. At this stage of the research, it was not possible to identify why they had not implemented these solutions.

From the literature perspective, Chen[12] described the ASRs as establishing the boundary of CD in an abstract manner without further details on how every ASR can be adapted in a practical development environment. Vost and Wagner[13] described CD for the automotive industry. Nevertheless, their approach partially adopted the CD concepts identified by Fowler[1] in terms of software integration practices. Makinen *et al.*[14] conducted a survey to identify the common tools used in the 18 companies' toolchains; these companies were interested in improving their delivery frequency. Their survey was based on semi structured interviews with the corresponding Finnish software-intensive organizations to identify their delivery approaches. Although this survey covered many aspects related to software delivery pipelines, they briefly covered the cultural aspects that a software development environment must fulfill to facilitate automation. It is believed that culture building is a key attribute for successful implementation of a CD model. Shahin *et al.*[15] studied the impact of the system architecture on continuous deployment and delivery and identified that challenges for adopting CD might be attributed to several factors, including the availability of monolithic systems, highly coupled code, less reuse practices and team dependencies. Their work mainly examined the architectural aspects of software systems, which is one of the characteristics defined in this work. Schermann *et al.*[16] argued that there is a trade-off between the velocity of delivering releases of software and the quality of the delivered release. Thus, they proposed a confidence-velocity model for enhancing software delivery to balance the delivery speed and the quality of the releases. Their investigation considered the organizational policies for releasing software and the interaction activities with customers for obtaining fast feedback, which are related to the cultural dimension of this work's taxonomy. Stewart *et al.*[17] investigated the relation between code quality and frequency of software delivery in a number of open source software projects in terms of size, coupling and cohesion on the overall speed of release delivery. It is believed that Stewart[17] work investigated the architectural aspects of source code projects, which is one of the characteristics covered by this work. However, their work is more detailed in terms of the inspection at the source code level, while this work explored the generic patterns and tools utilized for CD. Bellomo *et al.*[18] proposed a framework to address issues of software CD in terms of identifying software reliability factors (e.g., architecture) and monitoring development (e.g., APM). Their main hypothesis of the investigation asserted that the architecture of the system is the key factor that impacts CD. In this work, it was assumed that a flexible software architecture is key to promoting CD practices, which is more general than Bellomo's view. Martensson *et al.*[19] examined the behavior of developers in two organizations that adopted CI practices. These researchers identified 12 factors that were believed to impact the CD of releases and related these factors

to the organizational structure of the system, which is influenced by the architecture of the system under development.

Based on the reviewed work from the literature, it was observed that the key characteristics needed for successful adoption of CD were mainly related to cultural aspects and the architecture of the system in the project. Both aspects were covered in detail by this work and the underlying tools were listed as per the surveyed solutions presented by the taxonomy work.

CONCLUSION

This paper presents an analysis of the available software CD approaches in the software industry from a practical perspective by identifying their key characteristics as a step toward standardizing the identified CD practices for generating a common reference model that contributes to the software engineering paradigm. The next stage of this work will be to apply the identified CD components on a use case to investigate their impact on improving the development process and to establish a guideline for the adoption of CD by software development organizations.

SIGNIFICANCE STATEMENT

This study provides a comparative framework for evaluating the applicability of CD within software development organizations. The study investigated the conceptual characteristics and the corresponding implementation details of CD approaches in the market and hence can be used to establish a reference model for CD as a significant contribution to standard software engineering models and knowledge areas.

ACKNOWLEDGMENT

The authors of this work would like to thank the Deanship of Information Technology at Umm Al-Qura University for providing the necessary support, development environment and infrastructure, which was vital to the completion of this work.

REFERENCES

1. Fowler, M., 2013. Continuous delivery. May 30, 2013. http://martinfowler.com/bliki/ContinuousDelivery.html.
2. Humble, J. and D. Farley, 2010. Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation. 1st Edn., Addison-Wesley Professional, Massachusetts.
3. Zend Technologies Inc., 2013. Zend blueprint for continuous delivery. http://www.zend.com/whitepapers/zend-blueprint-for-continuous-delivery.pdf.
4. XebiaLabs, 2018. Introducing continuous delivery in the enterprise. White Paper. https://xebialabs.com/resources/whitepapers/introducing-continuous-delivery-in-the-enterprise/.
5. XebiaLabs, 2018. Preparing for continuous delivery in the enterprise. White Paper. https://xebialabs.com/resources/whitepapers/preparing-for-continuous-delivery-in-the-enterprise/.
6. XebiaLabs, 2018. Best practice for continuous delivery automation in the enterprise. White Paper. https://xebialabs.com/resources/whitepapers/best-practice-for-continuous-delivery-automation-in-the-enterprise/.
7. Duvall, P.M., 2017. Continuous delivery patterns and antipatterns in the software lifecycle. DZone Inc. https://dzone.com/storage/assets/3989241-dzonerefcardz145-cdpatternsantipatterns.pdf.
8. DZone Inc., 2015. The DZone's guide to continuous delivery. https://dzone.com/asset/download/1934.
9. Laukkanen, E., J. Itkonen and C. Lassenius, 2017. Problems, causes and solutions when adopting continuous delivery-A systematic literature review. Inform. Software Technol., 82: 55-79.
10. UQU., 2018. UQU web-portal. Umm Al-Qura University. https://uqu.edu.sa.
11. Usman, M., R. Britto, J. Borstler and E. Mendes, 2017. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. Inform. Software Technol., 85: 43-59.
12. Chen, L., 2015. Towards architecting for continuous delivery. Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture, May 4-8, 2015, Montreal, QC, Canada, pp: 131-134.
13. Vost, S. and S. Wagner, 2016. Toward continuous integration and continuous delivery in the automotive industry. https://arxiv.org/ftp/arxiv/papers/1612/1612.04139.pdf.
14. Makinen, S., M. Leppanen, T. Kilamo, A.L. Mattila, E. Laukkanen, M. Pagels and T. Mannisto, 2016. Improving the delivery cycle: A multiple-case study of the toolchains in Finnish software intensive enterprises. Inform. Software Technol., 80: 175-194.
15. Shahin, M., M.A. Babar and L. Zhu, 2017. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access, 5: 3909-3943.

16. Schermann, G., J. Cito, P. Leitner and H.C. Gall, 2016. Towards quality gates in continuous delivery and deployment. Proceedings of the 24th IEEE International Conference on Program Comprehension, May 16-17, 2016, Austin, Texas, USA.
17. Stewart, K.J., D.P. Darcy and S.L. Daniel, 2005. Observations on patterns of development in open source software projects. Proceedings of the 5th Workshop on Open Source Software Engineering, May 17, 2005, St. Louis, MO, USA., pp: 1-5.
18. Bellomo, S., N. Ernst, R. Nord and R. Kazman, 2014. Toward design decisions to enable deployability: Empirical study of three projects reaching for the continuous delivery holy grail. Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 23-26, 2014, Atlanta, GA, USA.
19. Martensson, T., D. Stahl and J. Bosch, 2017. Continuous integration impediments in large-scale industry projects. Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA), April 3-7, 2017, Gothenburg, Sweden.