UNIVERSITY OF MINNESOTA

SENG 5852

Continuous Integration, Delivery, & Deployment: What Do They Mean?

RESEARCH PAPER

LUE XIONG

May 10, 2019

1 Introduction

The software industry is transforming at a rapid pace to accommodate the dynamic nature of the market and as a result, it continues to struggle to find process-identity with continuous software engineering.

The software engineering community has for a few decades experimented with the concept of distributing software in faster release cycles; endeavoring to do so without sacrificing reliability and security. To achieve such a goal, there has been a widespread movement in the technical community to advocate for using Agile practices, and in particular: continuous integration, delivery, and deployment. The traditional methods of software development no longer meet the need of businesses that – now more than ever – want to proactively engage and retain their customers. The organizational transition to Agile practices demands a large mentality change and require individuals to recognize software as incremental features developed with cross-collaboration of small comprehensive team units, as opposed to large modules developed by siloed units.

2 Continuous Software Engineering

The word 'Agile' can be quite a jumbled terminology depending whom a person talks to. Software professionals have their own experiences and stories of using it in their day-to-day work with varying degrees of opinions and outcomes. The Agile Manifesto is a declaration of the principles of Agile, which attempts to alleviate any misunderstanding of what it means to be Agile in a software context. One principle of the Agile Manifesto states that the highest priority is satisfying customers with early and often continuous delivery (Meyer, 2014, p. 50). It is apparent that at the heart of Agile is the value of producing software quickly and efficiently, without sacrificing a team's sustainability. One such software development method that has been in existence for over two decades, and fundamentally a practical expression of the principles of the Agile Manifesto, is Extreme Programming (XP).

Extreme Programming involves a set of practices including test-first development, automated testing, fast release cycles, refactoring, continuous integration, and among other things. Software professionals have embraced these set of practices for its effectiveness, even if arbitrarily and selectively picking from its' set. Bertrand Meyer – a well-respected and influential individual in the software engineering community – considers many of the XP practices to be 'brilliant.' He goes as far as to say that the practices of continuous integration and testing are major factors for the success of modern software projects (Meyer, 2014, p. 154). It is no wonder that many software professionals and organizations have turned to Agile and now embody Agile practices to keep up with the demands of the software market.

Of the ideal software practices that are commonly known and trending vocabulary – continuous integration, delivery, and deployment – come with the responsibility of defining them. Many software professionals, however, have misunderstood these development practices and at times, infuse them as one concept without careful and thoughtful distinction. It would, therefore, be wise to define each component for what they represent and why it matters.

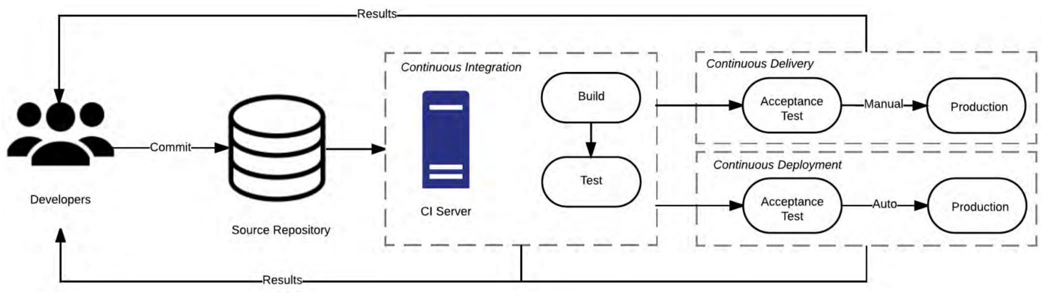

Figure 1: The relationship between continuous integration, delivery and deployment, Adapted from Shahin et al. (2017, p. 3911)

Continuous integration is a practice entrusted to the active engagement of the members of a team developing code, of which they participate by integrating and merging code to a source control management service such as GitHub; typically on a frequent basis. It contains an automated process that builds the software based on the new state of the code and verifies the software against a collection of tests. The important thing to note is that there is a quick feedback loop to the maintainers of the code based on the tests, determining whether any action is necessary to move forward. The main benefits to software organizations using continuous integration are shorter and faster release cycles, improvement of software product quality, and an increase of team productivity (Shahin et al., 2017, p. 3910).

It is absolutely crucial to note from Figure 1 that using continuous delivery is state-dependent practice on continuous integration. The purpose of continuous delivery is to deploy a software product to a production-like environment meant for acceptance testing by stakeholders such as the customers, but only after it has been verified to work by the continuous integration process. The stakeholders are the main drivers for the decision of whether the software product can go into a real production environment. In the case that it is accepted, a manual process will be used to deploy the software into production. The benefits of continuous delivery include reduced deployment risks, faster user feedback, and overall lower project costs (Shahin et al., 2017, p. 3910-3911).

The same state-dependence goes for continuous deployment as well; it cannot exist without continuous integration. There is also the implication that continuous delivery is a major part of continuous deployment. The distinguishing factor of continuous deployment versus continuous delivery is the extra step of an automated process of the deployment to production (Shahin et al., 2017, p. 3911). For this to be possible, the software must be verified against a robust and fully automated collection of test suites, assuring with high confidence that the software works as it is intended to by design. Rod Cope, CTO of Rogue Wave Software says that companies that use continuous deployment can push hundreds or even thousands of releases into production per day (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018). The intent behind continuous deployment is delivering a working product to customers as fast as possible, hence the heavy emphasis on automation. Upon asking a software architect about the practice, who chose not be to named, said, "let me put it this way, if I want a pony, then give me a damn pony. That's how simple it should be" (anonymous, personal communication, April 30, 2019). Although spoken quite metaphorically, the architect does get across the idea that if the continuous deployment system was built the way it was supposed to, a 'want' should, in fact, result in a ready-to-use product.

Software professionals, however, often get these three continuous software engineering practices confused or often times do not understand the deeper details that differentiate one from another. For example, Kofi Senaya, Director of Product from Clearbridge Mobile says "continuous Integration means that when [developers] are working on the same code, they are building and unit-testing it to prevent any integration problems" (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018). The first misunderstanding from Kofi is that he implies continuous integration only runs unit tests against the source code. Other levels of testing such as integration and system testing can also be implemented for continuous integration. The second misunderstanding is the implication that unit tests will cover all cases of integration functionality. There is a reason that integration tests exist, and that is for testing that module-to-module interaction works as intended. The third misunderstanding is stating that the developers are building and unit testing the software. The statement assumes that continuous integration is a manual process by which the developers partake in compiling and then running tests against the software. Kofi then goes on to say that "the code is verified by the automated build which allows teams to identify problems" (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018), which conflicts with the previous statement. The conceptual integrity of continuous software engineering practices has yet to mature within the software industry. It is clear that explaining the intricacies of these continuous software engineering practices is a challenge for software professionals.

An added layer of complexity that makes defining what it means to do continuous software engineering more difficult is the well-documented phenomena of non-collaboration between software developers and academic researchers. Computer scientist and software engineering professor, Jan Bosch, states that even the greatest developers in the software industry view top academic researchers as people who give insignificant learning value and the converse being true as well (Bosch, 2014, p. 29). In particular, top researchers do not believe that top developers contribute much to their body of research. The intrinsic difference between the developer and researcher is in how they view the values of software engineering as a means to a different end. Developers by nature of their work are expected to produce software in a timely business-oriented manner, wherein researchers are focused on the insights that data analysis can accomplish.

The implementation of continuous integration, delivery, and deployment is time-intensive and difficult as well as complex and inherently abstract. Among the many identified challenges for continuous practices are (Shahin et al., 2017, p. 3925-3927): lack of expertise and skill, lack of appropriate technologies, coordination and collaboration, increased business pressure and workload, improper test strategies, stateful design and code dependencies, resistance to change, and the list goes on. One of the distinct challenges of the identified challenges is the lack of expertise and skill. The traditional method of development typically required a developer to work isolation on features with a limited and familiar stack of technology. Developing continuous practices in contrast requires new and different technical skills, a heap of new and unfamiliar technology, and soft skills for an unprecedented level of collaboration across teams. Transitioning to such a different way of development is daunting for many and carries with it, a set of new engineering issues for developers to solve.

A particular issue that plagues the software industry is the traceability of requirements in the context of using continuous practices.

On the topic of continuous integration alone, Daniel Stahl points out the problems that the software industry is facing (Stahl, 2017, p. 60):

As the size of a software development effort increases several things tend to happen. The size of the code base increases, implying longer build times. The number and the scope of tests tend to increase, implying longer test times. This in turn leads to a decreased time frame in which to integrate one's changes... Furthermore, a great number of developers involved implies both an increased rate of changes and more people impacted by failed integrations.

The software engineering community is still in the early stages of understanding and implementing continuous practices. It will take time for the community to find engineering standards for continuous practices to mitigate the risks while taking advantage of its' benefits.

2.1 Struggles of Traceability

2.1.1 Importance

- What is the importance of traceability for the software engineering community?
  - (Stahl, 2017) & (D. Stahl, Hallen, & Bosch, 2016)

2.1.2 Problem of Mapping

- What is the problem of mapping requirements to implemented code and the converse?
  - (Stahl, 2017) & (D. Stahl et al., 2016)

2.1.3 Eiffel Framework

- What is the proposed solution to address traceability issues in CI/CDE/CD environments?
  - (Stahl, 2017) & (D. Stahl et al., 2016)

3 Conclusion

4 Bibliography

References

Atkinson, B., & Edwards, D. (2018). *Generic pipelines using docker: The devops guide to building reusable, platform agnostic ci/cd frameworks*. Apress. doi: 10.1007/978-1-4842-3655-0

Bosch, J. (2014). *Continuous software engineering*. Springer International Publishing. doi: 10.1007/978-3-319-11283-1

Continuous delivery, deployment & integration: 20 key differences. (2018). Retrieved from `https://stackify.com/continuous-delivery-vs-continuous-deployment-vs-continuous-integration/`

Meyer, B. (2014). *Agile! the good, the hype and the ugly*. Springer International Publishing. doi: 10.1007/978-3-319-05155-0

Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, *5*, 3909–3943. doi: 10.1109/access.2017.2685629

Stahl. (2017). *Large scale continuous integration and delivery: Making great software better and faster*. University of Groningen.

Stahl, D., Hallen, K., & Bosch, J. (2016). Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the eiffel framework. *Empirical Software Engineering*, *22*(3), 967–995. doi: 10.1007/s10664-016-9457-1