UNIVERSITY OF MINNESOTA

SENG 5852

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

RESEARCH PAPER

LUE XIONG

May 10, 2019

Contents

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 Constitution of Continuous Software Engineering Practices

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.

Continuous integration is a practice entrusted to the active engagement of the members of a team developing code, of which they will participate by integrating and merging code to a source control management service such as GitHub; typically on a frequent basis. It contains an autonomous 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, Babar, & Zhu, 2017, p. 3910).

It is absolutely crucial to note 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. 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 it to 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 deliver 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. 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 profes-

sor, 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 value of software engineering. Developers by nature of their work are expected to produce software in a timely business-oriented manner, whereas researchers are focused on the insights that data analysis can accomplish.

There has been effort to bridge the gap between the two groups but this effort has been far from trivial.

2.1 Differences of Interpretation & Implementation

2.1.1 Viewpoint of Software Professionals

- How do software professionals interpret and implement CI/CD/CDE?
  - (Atkinson & Edwards, 2018) & (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018)

2.1.2 Viewpoint of Academic Researchers

- How do academic researchers interpret and believe how CI/CD/CDE should be implemented?
  - (Bosch, 2014), (Shahin et al., 2017), & (Stahl, 2017)

2.1.3 Collaboration Effort

- What effort is there to bridge the phenomena of non-collaboration between developers and researchers?
  - (Bosch, 2014) & (Stahl, 2017)

2.2 Benefits of Continuous Integration, Delivery, & Deployment

2.2.1 Self-healing Systems

- What are the metrics and tools that software professionals use to mitigate having to manually fix software issues?
  - (Bosch, 2014)

- How do these self-healing systems work?
  - (Bosch, 2014)

2.2.2 Risk Reduction

- How does continuous software engineering reduces risk in systems?
  - (Atkinson & Edwards, 2018), (Bosch, 2014), (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018), & (Stahl, 2017) (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018)

2.2.3 Faster Release Cycles

- How are faster release cycles are achieved?
  - (Atkinson & Edwards, 2018), (Bosch, 2014), (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018), & (Stahl, 2017) (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018)

2.2.4 Overall Cost Reduction

- Why will all of the above will reduce cost?
  - (Atkinson & Edwards, 2018), (Bosch, 2014), (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018), & (Stahl, 2017) (*Continuous Delivery, Deployment & Integration: 20 Key Differences*, 2018)

2.3 Struggles of Traceability

2.3.1 Importance

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

2.3.2 Problem of Mapping

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

2.3.3 Eiffel Framework

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

2.4 Transition an Agile Environment

2.4.1 The Effect of Organizational Change to Agile

- What are the problems that businesses face in attempt to switch to Agile practices?
  - (Bosch, 2014) & (Meyer, 2014)

2.4.2 Roles in Agile

- What are typical roles that each individual plays in an Agile environment?
  - (Bosch, 2014) & (Meyer, 2014)
- Why do these roles exist?
  - (Bosch, 2014) & (Meyer, 2014)

2.4.3 Paradigm Shift in Leadership

- How has leadership changed as a result of Agile?
  - (Bosch, 2014)

3 Conclusion

3.1 Rephrase Thesis Statement

3.2 Closing Statement

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