Testing in Continuous Integration and Deployment

Project in ETSN20 Software Testing

Andreas Warvsten – ine15awa@student.lu.se
Alexander Goobar – ine15ago@student.lu.se
Anton Engström - ine15aen@student.lu.se

Abstract—**This report explores testing in a Continuous Integration and Deployment setting. It provides an overview of Continuous Integration and Deployment and maps testing done in that specific context. It is time consuming and costly to manually deliver new stable releases of a software to a customer. As a result, tools and frameworks have been developed to automate the build, testing and deployment of software into production. The importance of test automation was identified using models and frameworks describing the Continuous Integration and Deployment pipeline process. Key aspects and challenges in implementing this automated process are analyzed, such as having an adequate testing strategy and investing in test automation.**

I. Introduction

The terms "Continuous Integration" and "Continuous Deployment", henceforth referred to as CI and CD, are often mentioned in the context of modern software development, especially when speaking of agile practices. To fully understand these terms and their background, a brief history of agile and software development methodologies will be presented.

Agile programming has grown to become a preferred way of working in many modern software projects. It focuses on short iterations with quick feedback loops. As the use of agile programming methodologies grew with the demand for faster delivery, a need for automation of build, test and deployment processes emerged. It was crucial for the agile programmer to minimize time spent on waiting for feedback from testing and integration teams for every change or added functionality they submitted, especially since these could take days to review [4]. Speeding up the process of testing the software, building the application and deploying it to the production environment, without sacrificing quality, is what the CI/CD process aims to achieve [7].

To understand CI, and the issues it deals with, one should understand its origins in the agile movement. The concept of CI was first written about in Kent Beck's book "Extreme Programming Explained" released in 1999 [11], and emerged as the benefits of agile software development were becoming increasingly apparent. Before the rise of agile programming, the waterfall model was widely used [1]. Issues with the waterfall model was, however, that it was not until the end of development, during the integration phase, that it became apparent whether or not the system was operational. Not building the whole system until the developers had finished developing all components and functionality, also meant that if the build was not successful, it was not obvious what caused the problem and where to look for bugs. There may be many components of the build that do not interact with each other as one had predicted. It was from this issue that the agile ethos of regular integration of the codebase was born. This way, the feedback of whether or not the system works comes earlier as one develops smaller parts of it at a time. With smaller additions to the system, it is easier to identify what has changed and what might be causing any integration issues or bugs. This was taken one step further when Kent Beck posed the question "if regular integration is good, why not do it all the time?", i.e. why not build the system every time anyone commits a change to it? That way it is very easy for developers to gather quick feedback on their developed code. They also avoid the problem of having to search many possible sources when integration fails. The goal of CI is to reduce the amount of bugs, the time it takes to locate and fix them and as a consequence deliver the software system faster. [1] In this report, we aim to investigate what characterizes testing in these extremely iterative environments.

II. Description

A. The CI/CD Pipeline

Figure 1 illustrates the fundamental components of the Continuous Integration, Delivery and Deployment pipelines. As can be seen, CI is the precursor to deployment and delivery. When working according to CI principles, the developers frequently commit their code to the source repository, as this helps keep the scale of the potential merge conflicts to a minimum. The build is then constructed on a CI server, where automated tests are run, and the results are used as feedback to the developers. In both Continuous Deployment and Continuous Delivery, all tests need to pass, in order to move further down the pipeline. The tests employed in the CI/CD pipeline constitute what's known as a quality gate. In order for the build to be promoted to production, it has to pass the quality gate.[7]

It can be difficult to distinguish Continuous Deployment from Continuous Delivery, as the terms are quite similar, and are often expressed in the same context. The difference is that Continuous Deployment aims to deploy updates to production as soon as they have passed the quality gate, while Continuous Delivery only requires that the system

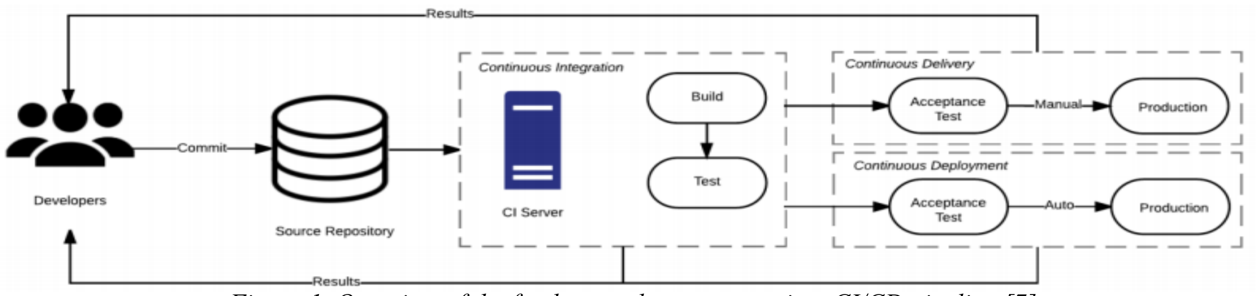

Figure 1. Overview of the fundamental components in a CI/CD pipeline [7]

shall be *ready* for deployment. A way to implement Continuous Delivery is to first deploy onto a staging server, where new releases can be tried in a simulated environment, until someone decides to manually deploy the feature into production [1].

B. Role of Testing in CI/CD

As the frequency of commits increases with CI/CD, the number of tests that need to be performed does as well. Much effort is therefore being put into trying to minimize time in the build process and test phase, as this would improve the efficiency of the CI/CD process [7]. Shahina et al. mentions that "one of the most prominent roadblocks to adopting CI, reported by several studies, were the challenges associated with the testing phase" [7]. They also present six aspects that are of great importance for the success of a CI/CD process. Of the six described, three are directly attributed to testing, and these are: *Reduce build and test time in CI*, *Increase visibility and awareness on build and test results in CI* and lastly *Support (semi-) automated continuous testing* [7].

Challenges with implementing a CI process are also described, and when it comes to testing, the two main ones are: lack of a proper test strategy and poor test quality. The lack of a proper test strategy can often be attributed to either lack of fully automated testing, lack of Test Driven Development or both [7].

C. Tools

There are several tools used throughout the CI/CD pipeline. The ones most pertinent to testing in a CI/CD context are the server coordinating tools, the build tools and testing tools. Jenkins, a widely used tool, can be used to set up both the CI and CD servers in both Continuous Delivery as well as Continuous Deployment pipelines. The CI/CD pipeline also uses other tools for task automation. These include building tools, such as Ant and Maven, as well as testing tools, such as JUnit and Selenium. [7] Since Jenkins does not provide effective versioning control [12], it is not the end all be all tool for full implementation of a CI/CD pipeline. For example, one of the previously mentioned testing tools, supported by Jenkins, could be used to organize unit tests. Jenkins would then schedule, run and coordinate the other steps of the CI. These include building the application and interacting with the code repository. Jenkins also provides plugins for monitoring the various steps in the pipeline, and helps visualizing the results. [12]

D. Models and frameworks for testing in CI/CD

To get a deeper understanding of the processes that make up a CI/CD pipeline, especially in regards to testing, the Test Orchestration framework will be described. Furthermore, the Test Activity Stakeholders model presents how CI/CD pipelines can include certain test activities to support stakeholder interests.

1) Test Orchestration Framework

The Test Orchestration framework was designed by Nikhil Rathod and Anil Surve in 2015 to give a general description of the CI/CD process, by mapping a streamlined release pipeline. The framework has four main components: build automation, testing automation, reporting and deployment automation. While build automation and deployment automation are important, this section will be focusing on the testing automation and reporting components. [2]

Test automation is described as the use of testing frameworks to automatically execute tests. With many test activities being time consuming to complete manually, it is a key factor to build a foundation of automated testing when adopting CI principles. [2]

Once the automated testing process has matured, it should be able to be run at any time and frequently. In an ideal situation, all regression tests are automated. Since CI/CD implies rapid release processes and gradual updates, automated regression tests are essential to fully be able to take part of the benefits of CI/CD. [2]

Reporting is the process of analyzing the test results and the result of that analysis. While testing is important to find and identify errors, the analysis part is necessary to understand the errors and the potential impact they can have. Therefore, it is important that the CI/CD pipeline can present the findings of the automated tests in a generated test report. It is then the responsibility of either developers or testers to read these reports and take appropriate action if there are critical errors. [2]

2) *TAS Model*

The Test Activity Stakeholders (TAS) model was presented by Torvald Mårtensson, Daniel Ståhl and Jan Bosch in 2018. It is based on 25 interviews done at four different companies developing large-scale systems. The model describes how CI pipelines can be formed to include certain test activities that can support four main stakeholder interests: checking quality, securing stability, measuring progress and verifying compliance. [3]

The model identifies three phases in the pipeline:

• Early in the pipeline: In this phase, unit tests and system tests are executed to check the new software changes. Testing vital functions with system tests is also done to confirm the system stability and integrity, likely monitored by a test manager.

• Later in the pipeline: Here, a wider range of system tests that support multiple stakeholder interests are run, and they are followed up by a developer if necessary.

• Release pipeline: In this phase, the project is approaching a release. Here, test cases based on the requirements are executed on real hardware to verify compliance. [3]

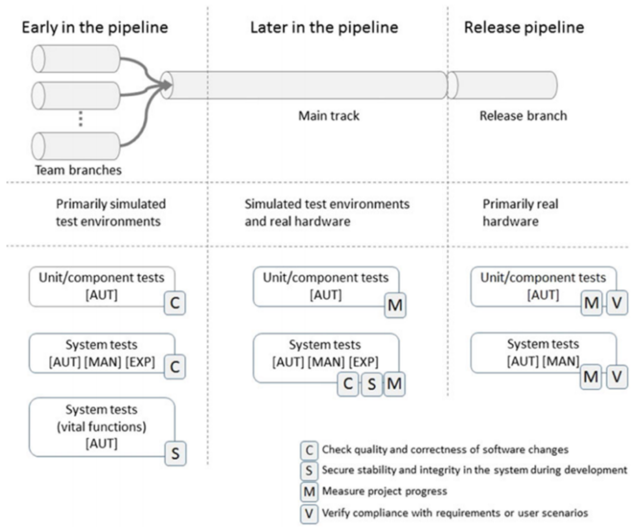
Early in the pipeline
Later in the pipeline
Release pipeline
Main track
Release branch
Team branches
Primarily simulated test environments
Simulated test environments and real hardware
Primarily real hardware
Unit/component tests [AUT] C
System tests [AUT] [MAN] [EXP] C
System tests (vital functions) [AUT] S
Unit/component tests [AUT] M
System tests [AUT] [MAN] [EXP] C S M
Unit/component tests [AUT] M V
System tests [AUT] [MAN] M V
C Check quality and correctness of software changes
S Secure stability and integrity in the system during development
M Measure project progress
V Verify compliance with requirements or user scenarios

Figure 2. Overview of The TAS model [3]

In Figure 2, the model is summarized graphically, also marking how different test techniques support the stakeholder interests. [AUT] means automatic testing, [MAN] means manual testing and [EXP] means exploratory testing. [3]

E. Test Driven Development

The The authors behind the Test Orchestration framework describe Test Driven Development (TDD) as a key feature of agile software development, also enabling an efficient way of working with a CI pipeline [2]. Shahina et al also mentions this as a key aspect for successful CI implementation [7]. A case study done at a Dutch SME confirmed the combination of TDD and CI to be a favorable approach in terms of both quality and productivity for software production [5]. In TDD, low-level tests based on the requirements are written before the production code. A few unit tests are coded initially, followed by a simple and partial implementation to pass the tests. Afterwards, one more unit test is added, followed by code implementation to pass the test, but not more. This process of incrementally adding unit tests and code is then repeated until there is nothing left to test, see Figure 3. [6]

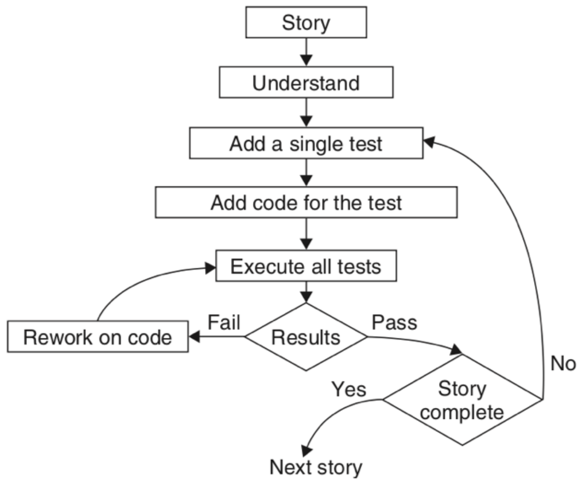

Figure 3. The TDD process [6]

III. Analysis

To further understand the impact of testing and various important aspects surrounding it, in a CI/CD context, the following questions will be discussed:

- What are the key factors of testing in a CI/CD process?
- What areas could be improved for a more efficient CI/CD process?
- What are the main organizational challenges?

A. Importance of Test Automation and Infrastructure

As can be seen in both the Test Orchestration framework and the TAS model, test automation plays an essential role in CI/CD. In the Test Orchestration framework, test automation is a main component in the pipeline, which can be interpreted as a hard requirement for CI/CD. In the TAS model, automatic testing is presented as a possible test technique for all test activities. However, in this model exploratory and manual testing are presented as viable options for some of the phases and test types, such as system tests. This would imply that while test automation plays a key role in the pipeline, it is not a hard requirement for each testing activity.

Many companies seek to achieve automated tests, but the process of getting there can be difficult. Implementing automated tests may require a large initial investment, by getting the infrastructure and code to support the automation. The developers writing the test code would need knowledge of testing and development to ensure quality tests, and employees with this cross competence can be costly to acquire.

The infrastructure, including automation and CI/CD tools, also play a key role. Without the help of automation tools, integrating and testing every commit becomes a tedious task in a rapid iterative development process. As such, this is an area where automation needs to take center stage and where much of the effort is spent. As mentioned in section 2.C, there are many tools available to help set up the testing in the CI/CD pipeline. The infrastructure enables teams to work efficiently and spend less time on repetitive manual tasks.

Although the infrastructure does most of the work, if set up correctly, it is crucial that the correct routines support it to extract the most value out of it. The important routine to be noted on this subject is to commit early and often. It is the high frequency of code commits, which results in integration and build of the application through the CI pipeline, that keeps iterations short and provides quick feedback.

B. Areas of Improvement in the Testing Process

As stated in section 2.B, aspects related to the test phase are what generate some of the biggest challenges for adopting a CI/CD process. Apart from automation, two important aspects related to testing were also presented. These were: *Reduce build and test time in CI* and *Increase visibility and awareness on build and test results in CI.* In order to continue improving the process of CI/CD, it is crucial to investigate areas and aspects of testing that can be improved, as they can have a huge potential impact on the effectiveness of CI/CD.

Two proposed methodologies for reducing build and test time are VMVM and VMVMVM. The first abbreviation stands for *virtual machine in a virtual machine* and the second one takes it one step further with *virtual machine in a virtual machine on a virtual machine*. The idea behind these approaches is to isolate in-memory and external dependencies between test cases, which would enable them to be run simultaneously, in parallel processes. By having many test cases running in parallel, the time of testing could be reduced significantly [9].

Another approach is based on using historical data from the test suite, in order to construct a set of test cases that should be prioritized. The prioritized test cases, i.e. the most critical ones, can then be used early in the testing process. The effect of this is that the developers can receive feedback on these tests much quicker than they otherwise would, which speeds up the entire process [7].

An issue that comes as a by-product of the increased number of tests in a CI environment is the increasing difficulty in handling all that test data [7]. As mentioned in 2.D, the Test Orchestration framework also emphasizes the importance of reporting the data, generated from the tests, in a comprehensible manner. Without an effective plan for structuring it, the feedback process becomes significantly slower.

Furthermore, Shahina et al. have summarized two approaches that can be utilized in order to increase the visibility and awareness on build and test results in CI. The first one is CIViT, Continuous Integration Visualization Technique [7]. This is a technique that helps visualizing the entire testing process, in terms of time and extent of the testing. It also gives developers a way of visualizing the status of different components that will be or have been tested, which could help avoid spending time on performing duplicate testing [7]. For the second approach, Shahina et al. reference Brandtner et al. [8], who describe a possible way of making the data easier to comprehend by the use of SQA-mashup, a mashup framework. The idea behind this is to create a platform where the data from single testing tools can be integrated and visualized in a way that can facilitate the data analysis [8]. These types of improvements could help various stakeholders get an overview of the massive amounts of data generated in the CI/CD pipeline.

C. Organizational Challenges

While it might seem beneficial to move to a process built on CI/CD, many companies that are suited for this way of producing software still have not taken the initiative. For companies that already have adopted CI/CD, there are also challenges in improving the process, due to factors regarding the testing. Both of these situations can stem from the organizational challenges, in particular regarding the testing domain.

The transition to CI/CD might require the organization to adopt a more robust test strategy. As previously mentioned, Test Driven Development is in many cases regarded as a good test strategy to enable CI/CD for a project. If a certain organization does not currently use TDD, there can be several challenges related to the test strategy. Firstly, they could try to use CI/CD without a TDD strategy, which could be doable but not optimal. This could in turn end up with the CI/CD process not achieving the intended results, due to an inefficient way of working with the tests. Managers might then assume that the CI/CD process is flawed, and not connect the root cause to be the test strategy. Secondly, employees are often opposed to change [10]. This can result in a new test strategy not being executed to its fullest potential, hurting the overall QA performance and efficiency of the project.

Adapting the organization to support CI/CD can be a costly process. This was previously mentioned with regards to implementing automated tests, but this also applies to organizational challenges. For example, a company might feel pressured to keep manual tests to not put the employees performing these tests out of work. In addition, forming a new test strategy and building out the supporting infrastructure might also require a large initial investment. These factors could make the management refrain from initializing test automation, even though there are both long-term financial and productivity incentives to do so.

The organizational challenges may be most prevalent in big companies with large-scale applications, due to these companies often requiring longer processes to change strategies, project structures and infrastructure. However, larger projects can potentially mean larger benefits from a successful CI/CD process. Not only because of the significant reduction of overhead and cost savings, but also from the additional customer value that is created by providing new features at a more rapid pace. This is why it is important for organizations to realize the long term benefits of enabling the transition to CI/CD by initializing change and making initial investment towards a more CI-compatible testing process.

IV. Conclusion

The use of CI and CD principles is becoming more prevalent in the software development world, leading to more rapid iterations of software releases. With this trend, the need for robust and efficient testing procedures increases. Testing plays an important role in CI/CD pipelines. Therefore, it is crucial that the testing is carefully planned and structured. To further support the processes of CI/CD, software tools such as Jenkins are often used for task automation.

As could be seen in the Test Orchestration framework, it is very important to automate tests and also provide a way of presenting the findings in a generated report. When constructing the tests, it is important to have stakeholder interests in mind, which was described by the TAS model.

An adequate development process is needed to support a CI/CD pipeline. Optimally, the project should use Test Driven Development, where unit tests are written before the code implementation. This supports a test-focused mindset that is important when working with CI/CD, reducing the risk of deploying bugs into production.

As CI/CD pipelines heavily depend on test automation, it is apparent that this is a key requirement for the CI/CD process. The automated tests also have to be accompanied with the right infrastructure of software tools and routines to achieve an efficient process. In the efficiency factor lies some challenges in reducing build and test times, as organizations strive to have pipelines with shorter lead times, making this an appealing area for future research. Moreover, organizations might also face challenges making the transition to a CI/CD process, such as forming a new test strategy and investing in test automation, while potentially facing backlash from employees opposed to change or a reduced number of positions in manual testing. However, for managers, it is important to realize the long term benefits of CI/CD, and to invest in robust testing to support it.

V. Contribution Statement

The contributions in the report have been mapped in the following list:

Introduction - Engström
The CI/CD Process - Warvsten, Engström
Role of Testing in CI/CD - Warvsten
Tools - Engström
Models and frameworks for CI/CD - Goobar
Test Driven Development - Goobar
Importance of Test Automation and Infrastructure - Goobar, Engström
Areas of Improvement in the Testing Process - Warvsten
Organizational Challenges - Goobar
Conclusion – Goobar

The writing process was conducted with the authors being gathered. The purpose and content of each section was discussed and agreed upon in the group, and input from everyone was taken into account when required during the writing process.

References

[1] J. Humble and D. Farley. *Continuous Delivery - Reliable Software Releases Through Build, Test And Deployment Automation*. Pearson Education. 2010.

[2] N. Rathod and A. Surve. *Test Orchestration - A framework for Continuous Integration and Continuous Deployment* International Conference on Pervasive Computing (ICPC). 2015.

[3] T. Mårtensson, D. Ståhl, and J. Bosch. *Test activities in the continuous integration and delivery pipeline*. Journal of Software: Evolution and Process, Vol 31, Issue 4, April. 2019.

[4] S. Neely and S. Stolt. *Continuous Delivery? Easy! Just Change Everything*. In proceedings of Agile 2013, Nashville, Tennessee, August 5-9. 2013.

[5] C. Amrit and Y. Meijberg. *Effectiveness of Test Driven Development and Continuous Integration – A Case Study*. IT Professional. 2017.

[6] K. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley. 2008. pp. 71-72.

[7] M. Shahina, M. A. Babara, and L. Zhub. *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*. IEEE Access. 2017.

[8] M. Brandtner, E. Giger, and H. Gall. *SQA-Mashup: A mashup framework for continuous integration*. Information and Software Technology, Vol 65, September, pp 97-113. 2015.

[9] J. Bell, E.Melski, and M. Dattatreya. *Vroom: Faster Build Processes for Java.* Electric Cloud Gail E. Kaiser, Columbia University. 2015.

[10] M. Saari. *Implementing New Continuous Integration Tool*. University of Oulu. 2017. pp. 45.

[11] K. Beck. *Extreme Programming Explained*. Addison-Wesley Educational Publishers Inc. 1999.

[12] V. Armenise, *Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery*. 2015 IEEE/ACM 3rd International Workshop on Release Engineering, Florence. 2015. pp. 24-27