ISSN 2007-9737

Continuous Testing and Solutions for Testing Problems in Continuous Delivery: A Systematic Literature Review

Maximiliano A. Mascheroni[1,2], Emanuel Irrazábal[1]

[1] Universidad Nacional del Nordeste, Facultad de Ciencias Exactas y Naturales y
Agrimensura, Departamento de Informática,
Argentina

[2] Universidad Nacional de la Plata, Facultad de Informática,
Argentina

{mascheroni, eirrazabal}@exa.unne.edu.ar

Abstract. Continuous Delivery is a software development discipline where quality software is built in a way that it can be released into production at any time. However, even though instructions on how to implement it can be found in the literature, it has been challenging to put it into practice. Testing is one of these biggest challenges. On the one hand, there are several Continuous Delivery testing problems related to Continuous Delivery reported in the literature. On the other hand, some sources state that Continuous Testing is the missing element in Continuous Delivery. In this paper, we present a systematic literature review. We look at proposals, techniques, approaches, methods, frameworks, tools and solutions for testing problems. We also attempt to validate whether Continuous Testing is the missing component of Continuous Delivery by analyzing the different definitions of it and the testing stages and levels in Continuous Delivery. Finally, we look for open issues in Continuous Testing. We have found 56 articles and the results indicate that Continuous Testing is straight related to Continuous Delivery. We also describe how solutions have been proposed to face the testing problems. Lastly, we show that there are still open issues to solve.

Keywords. Continuous delivery, continuous testing, systematic literature review, testing, software.

1 Introduction

Continuous Delivery (CD) is a software development discipline where the software is built in a way that it can be released into production at any time [1]. In today's agile age, CD is an increasingly critical concept. This discipline supports agile practices and cuts the time-to-release of websites and apps from several weeks to just a few hours. However, according to Prusak [2], it could be argued that "the industry has not yet closed the circle when it comes to realizing a full CD process". Even though the literature contains instructions on how to adopt CD, its adoption has been challenging in practice [3].

The first part of a CD process is Continuous Integration (CI) [4]. CI is a software development practice where developers integrate code frequently verified by an automated build (including test), to detect defects as quickly as possible [5]. The second part of a CD process is Continuous Deployment (CDP) [4]. CDP is the ability to deliver software more frequently to customers and benefit from frequent customer feedback [6]. However, according to non-academic articles, there's a missing part: Continuous Testing (CT) [2]. Two sources [7, 8], define CT as the process of executing automated tests as part of the software delivery pipeline to obtain immediate feedback on the business risks associated with a software release candidate.

Testing is considered by Humble and Farley [4] as the key factor for getting CD, and they present a Deployment Pipeline (DP) as a CD model composed with different testing stages. However, while instructions on how to adopt these stages are given by these authors, some organizations have not adopted this practice at large yet [9] and some of them have found it challenging [10, 11, 12, 13, 14, 15, 16].

doi: 10.13053/CyS-22-3-2794

This raises the question whether there is a lack of best practices or whether the implementation of the test stages is highly problematic and the benefits are lower than the mentioned by the proponents of CD. The reported testing problems are described in Table 1.

In this systematic literature review (SLR), we look at proposals, techniques, approaches, tools and solutions for the various mentioned problems. We attempt to create a synthesized view of the literature considering these topics.

Furthermore, our mission is not just to identify new tools or techniques, but also to understand their relationship with CT. We attempt to validate whether CT is really the missing component of CD. We also look for different testing levels or stages in CD. We want to dig into CT, looking for the different parts of it, its limitation, boundaries and whether open issues exist or not.

We believe this SLR can provide an important contribution for the field, because while different testing CD approaches have been successfully implemented in some companies such as Facebook [31] or Atlassian [32], it is not known how generally applicable they are. On the other hand, the CT approach may vary and follow diverse pathways to ensure products are delivered free of defects. Thus, in addition, for research communities, our attempt offers a good starting point for future research topics in CT by detecting open issues related to it.

Previous SLRs have focused on CD topics such as: characteristics [15, 33] benefits [33], [34], technical implementations [35], enablers [15, 36], problems and causes [3, 15, 33] and solutions [3]. Thus, there have been only three of these studies that are related to testing in CD. The first one of them [3], studied problems of the adoption of CD and it reported some of the aforementioned testing problems with partial solutions.

However, the authors mentioned that they are “tricky” solutions and that the biggest problem is time-consuming testing. The second paper [15], studied how the rapid releases have repercussions on software quality, but it does not analyze the possible solutions. The last one of them [33], considers CT as a key factor in CDP. However, it only describes challenges and needs.

Therefore, to our knowledge, this is the first SLR which studies CT approaches, stages, solutions, tools and techniques.

Apart from this introductory section, this paper is structured as follows. We introduce our research goals and questions and describe our methodology in Section 2. Next, we introduce the results, which we further discuss in Section 3. Finally, we present our conclusions and ideas for future work in Section 4.

2 Methodology

In this section, the research goals and questions are presented. We also describe the research method used in this SLR, the filtering and data extraction strategy.

2.1 Research Goal and Questions

The goal of this paper is to look for solutions that have been reported to face the different mentioned challenges. It also has been found in non-academic articles that CT is the missing part of CD. We believe that different CT approaches exist and they might be solutions for those challenges. We also attempt to investigate the meanings of CT for the industry and the different stages or testing levels that compose it. Thus, we propose the following research questions:

- RQ1. Is there a valid and accepted definition for CT?
- RQ2. What types of testing or testing levels have been implemented for continuous development environments?
- RQ3. What solutions have been reported to solve testing problems in CD?
- RQ4. Are there open issues related to CT?

With RQ1, it is intended to establish what exactly CT is and whether it has a formal and accepted definition for both academic and empirical studies. In the same context, with RQ2 it is intended to set the stages or testing levels for CD. The aim of RQ3 is to look for any kind of solution, such as approaches, tools, techniques or best practices that can be used to face the testing problems mentioned in Section 1.

Table 1. Testing problems at adopting CD

Ref	Problem	Description
[16, 17, 18]	Time-consuming testing	Testing is a process that takes too much time.
[16, 19, 20, 21, 22]	Automated flaky Tests	One of the main characteristics of CD is reliability, but it is difficult to get highly reliable tests when they fail randomly.
[23, 24, 25, 26]	User Interface Testing problems	The user interface (UI) is the part of an application that changes most frequently and it can drive to flaky automated tests.
[9, 20]	Ambiguous test results	Test results are not communicated to developers properly, indicating whether the tests have passed or not. There are also some reports where it is not clear what exactly has broken a build.
[26, 27]	Rich Internet Applications and modern web applications.	Modern web applications utilize new technologies like Flash, Ajax, Angular or they perform advanced calculations in the client side before carrying out a new page request. It is hard to automate test cases for these types of applications.
[28]	Big Data Testing	Big data is the process of using large datasets that cannot be processed using traditional techniques. Testing these datasets is a new challenge that involves various techniques and frameworks.
[29]	Data Testing	Data is very important for different types of systems and errors in these systems are costly. While software testing has received highly attention, data testing has been poorly considered.
[30]	Mobile Testing	Automated mobile testing brings with it a lot of challenges regarding the testing process in different type of devices.
[17, 21]	Continuous Testing of Non-functional Requirements	While unit, integration and functional tests have been extensively discussed in the literature and widely practiced in CD, testing non-functional requirements has been overlooked.

Finally, the aim of RQ4 is to compile a list of open issues related to CT (if they exist).

We answer the research questions using a SLR. Cruzes and Dybå define a SLR as “a rigorous, systematic, and transparent method to identify, appraise, and synthesize all available research relevant to a particular research question, topic area, or phenomenon of interest, which may represent the best available evidence on a subject” [37]. A SLR may serve as a central link between evidence and decision making, by providing the decision-maker with available evidence. The importance of SLR for software engineering has been deeply discussed by a reasonable amount of studies [37, 38, 39, 40]. Kitchenham and Charters [41] describe a set of reasons for performing a SLR, as follows:

– To summarize the existing evidence concerning a treatment or technology
– To identify any gaps in current research in order to suggest areas for further investigation.
– To provide a framework/background in order to appropriately position new research activities.

In this study, we followed Kitchenham and Charter’s guidelines [41] for performing SLRs in software engineering, as seen in Fig. 1.

2.2 Planning and Search Strategy

After the research questions have been set, the next step is to define the search criteria. In this study, we followed the phases suggested by Kitchenham and Charters [41]. First of all, a preliminary search was performed in order to know other researches in CD. From this, synonyms and alternatives to CD were identified.

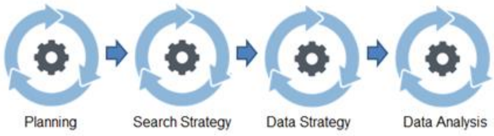

Fig. 1. Kitchenham's guidelines for performing SLRs in software engineering

Table 2. Search Engines used as data sources and obtained results

Engine	Total	Included after discards			Second Search	Included
		by repetition	by abstract	by irrelevance		
Scopus	272	243	172	26	3	29
IEEE Xplore	182	96	58	11	2	13
ISI Web of Science	81	33	22	5	1	6
ACM Digital Library	71	35	29	4	1	5
Science Direct	34	19	8	2	0	2
Research at Google	15	13	6	0	1	1
Total	655	439	295	48	8	56

According to [3], "CD is a fairly new topic" and there is not much mention in the literature concerning CD in the context of testing so we decided to include CI and CDP as they are claimed to be prerequisites and extensions of CD [4]. Thus, the obtained search query was:

("Continuous Development" OR "Continuous Integration" OR "Continuous Deployment" OR "Continuous Delivery" OR "Rapid Releases") AND (Testing OR Test) OR "Continuous Testing") AND Software

The first part of the query looks for studies in the field of CD, its synonyms (continuous development and rapid releases) and the other terms included (CI and CDP). The second part (CT) attempts to identify, appraise and synthesize all available literature relevant to CT. Finally, the "software" string was included in order to exclude articles that are not related to software engineering; the same approach was used in earlier SLRs [3, 35].

The search string was applied to titles, abstracts and keywords and it was executed on February 2017 and again on June 2017 in different data sources.

The second search was performed because there had been recent new publications in the area. The selection of the data sources is because they have been used in previous SLRs on Software Engineering [3, 39, 42] and they contain publications that are considered relevant to the area of interest. Using the mentioned search term, a lot of results were obtained, but many of them were considered as irrelevant to the purpose of this study. Table 2 shows the search engines used as data sources and the summary of the obtained results.

2.3 Filtering Strategy

The first search provided a total of 655 results as seen in Table 2. Those results were filtered using different inclusion and exclusion criteria. For the first discards, we used the following exclusion criterion:

1. Exclusion Criterion: duplicated studies are discarded.

From the 655 articles, we removed the duplicate studies, which left us with 439 articles.

Next, we studied the abstract of the remaining papers, and applied the following inclusion and exclusion criteria:

Table 3. Extraction form

#	Study Data	Description	Relevant RQ
1	Study identifier	Unique id for the study (S#).	Study overview
2	Authors, year, title		Study overview
3	Article source	Scopus, IEEE Xplore, ISI Web of Science, ACM Digital Library, Science Direct, Research at Google.	Study overview
4	Type of article	Journal, conference, symposium, workshop, book chapter.	Study overview
5	Application context	Industrial, academic, both.	Study overview
6	Research Type	Validation research, evaluation research, solution proposal, philosophical paper, experience paper.	Study overview
7	Evaluation method	Controlled experiment, case study, survey, ethnography, action research, systematic literature review, not applicable.	Study overview
8	Continuous Testing	Is there a valid and accepted definition for CT?	RQ1
9	Testing Stages	What stages or levels exist for CT?	RQ2
10	Solutions and Tools for Testing Problems	What solutions have been reported to solve testing problems in CD?	RQ3
11	Open issues in Continuous Testing	Are there CT related open issues?	RQ4

Table 4. Study quality assessment criteria

#	Question
Q1	Is there a clear statement of the goals of the research?
Q2	Is the proposed technique clearly described?
Q3	Is there an adequate description of the context in which the research was carried out?
Q4	Is the sample representative of the population to which the results will generalize?
Q5	Was the data analysis sufficiently rigorous?
Q6	Is there a discussion about the results of the study?
Q7	Are the limitations of this study explicitly discussed?
Q8	Are the lessons learned interesting and relevant for practitioners?
Q9	Is there sufficient discussion of related work?
Q10	How clear are the assumptions and hypotheses that have shaped the opinions described?

2. Inclusion Criterion: articles that propose tools, frameworks or any kind of solution for a continuous software development practice (CD, CDP, CI, and CT) are included.

3. Inclusion Criterion: articles that study CT are included.

4. Exclusion Criterion: if a continuous software development practice or a CT topic is not mentioned in the abstract, then it is discarded.

A total of 295 articles passed the criteria. Next, full-text versions of the studies were acquired. Finally, we applied the following exclusion criteria:

5. Exclusion criterion: articles that do not answer any of the research questions are discarded.

After the five criteria were applied, we got a total of 48 articles. As this filtering process was applied during the months of February, March, April, May and June, we repeated the process for those months.

Thus, new papers that were published within that period were included for the SLR. Finally, a total of 56 papers were included.

2.4 Data Extraction

We prepared forms to accurately record any information needed to answer the research questions. We extracted the data described in Table 3 from each of the 56 studies included in this systematic review. The extraction form was used in other SLR of software engineering [43].

3 Results and Analysis

This section describes the results of our study. We discuss the answers of each research question separately.

Our selection process resulted in 56 studies that met the inclusion criteria and we extracted the data following the extraction form described in Table 3. The articles are listed in Table A.1. (Appendix A).

Before presenting the results and analysis for each research question, we depict the quality assessment results and provide an overview of the general characteristics of the included articles.

3.1 Quality Assessment Results

The quality assessment is a key factor to increase the reliability of the conclusions. It has helped to ascertain the credibility and coherent synthesis of results [44].

We present the results of the quality assessment of the selected studies in Table B.1. (Appendix B), according to the assessment questions described in Table 4. These 10 questions were proposed by [44] and they provided a measure of the extent to which we could be confident that a particular study can make a valuable contribution to our review. The results show that the overall quality of the included studies is reasonable since the mean of quality was 76%.

3.2 Overview of the Studies

The selected studies were published between 2001 and 2017. In Fig. 2, the number of studies are presented by year of publication. An increasing number of publications can be noticed in the context of this review from 2015.

After analyzing this temporal view of the articles, we can conclude that the number of studies about CT and testing in CD is minimal throughout the years. Although the apparent increasing number of the studies on this topic from 2013, this result corroborates with the statement that testing practices in continuous software development have been somewhat neglected.

We used three categories for the application context of the studies: industrial (empirical), academic, and both. On the one hand, the studies that were published by authors affiliated to a University are considered as academic studies. On the other hand, the articles that explicitly state that they were performed in a real company or from an author's work in the industry, they are considered as industrial studies. If academic studies have experimentation or case studies in real working environments, we classified them as academic & industrial studies (both). However, articles that have experimentation in laboratories or non-real company environments are considered just as academic studies.

The results show that 16 studies (29%) belong to the academic context. 13 studies (23%) were conducted in industrial settings and 27 studies (48%) belong to the academic & industrial studies category (see Fig. 3). Most of the studies were applied in the industry (71%). From those articles that were applied in the industrial context, most of them are also academic studies.

This may indicate that practices that are emerging from universities and researchers are attempting to solve challenges faced by the industries. Furthermore, it also may point out that there is some approximation between industries and universities.

We categorized the evaluation method based on the following categories: controlled experiment, case study, survey, ethnography and action research. These categories for evaluation method were proposed by Easterbrook et al. [45]. In addition, we adopted two extra categories:

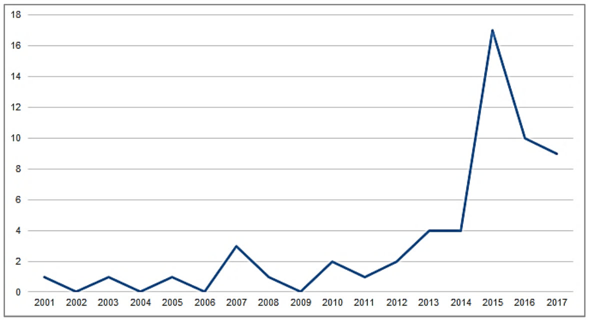

Fig. 2. Temporal view of the studies

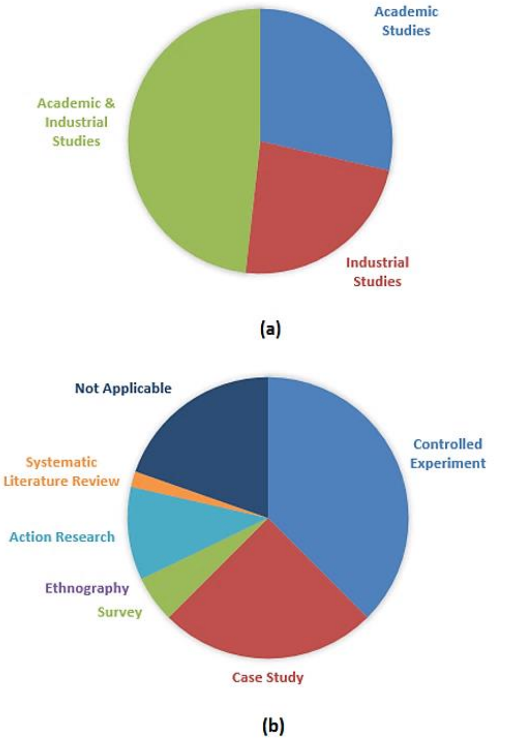

Fig. 3. Application context of analyzed articles (a), and evaluation method (b)

'systematic literature review' and 'not applicable'. The first category is used to classify studies that collect and critically analyze multiple research studies or papers in order to get conclusions. The second category refers to the articles that do not contain any kind of evaluation method in the study. Results of this classification can be seen in Fig. 3.

Most of the studies were evaluated empirically: controlled experiment (38%) and action research methods (11%). 14 of the articles were case studies (25%). On the other hand, there were 3 surveys (5%), only 1 SLR (2%), and 11 studies (20%) did not mention any kind of evaluation method or they are just opinion papers. Ethnography studies were not found.

Finally, the selected studies were categorized according to the applied research types defined by Wieringa et al [46], as can be seen in Fig. 4.

The most adopted research type is Solution Proposal with 26 studies (46%) followed by Experience Papers with 15 studies (27%). This is very related to our research questions because we are looking for new approaches, solutions, tools and techniques. On the other hand, Validation Research type has 6 studies (11%) and Evaluation Research has 5 studies (9%). Finally, 4 of the selected studies (7%) belong to Philosophical Papers category of research types.

In the next sections, we present and analyze the results of each research question. Discussions about the obtained results are presented at the end of each topic analysis.

3.3 RQ1. Is There a Valid and Accepted Definition for Continuous Testing?

The term was mentioned for the first time by Smith in 2000 [47], as a part of the Test-Driven Development (TDD), process at running unit tests. It was a testing process that has to be applied during the development and execution stages as automated regression testing 24 hours a day. However, the results obtained from the selected papers show that this concept has been evolving during recent years.

In 2003, Saff and Ernst in S42 introduced the concept of CT as "a means to reduce the time wasted for running unit test". It used real-time integration with the development environment to asynchronously run tests that are applied to the last version of the code, getting efficiency and safety by combining asynchronous testing with synchronous testing. Later, in 2004 and 2005 (S17), the same authors (Saff and Ernst) proposed an eclipse IDE plugin which used excess cycles on a developer's workstation to continuously run regression tests in the background while the developer edited code. These tests were composed by automated integration and unit tests.

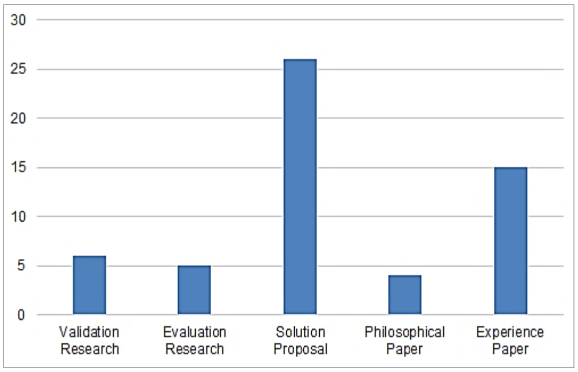

Fig. 4. Research types of the selected studies

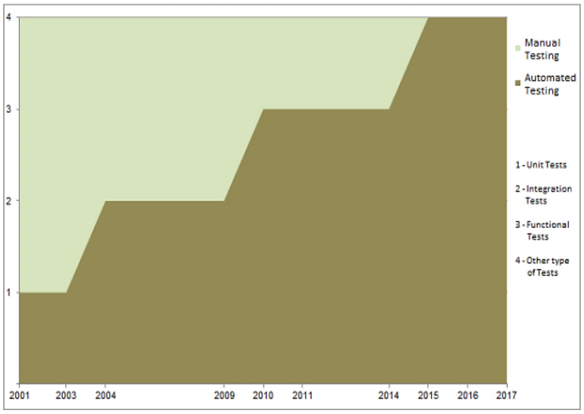

Fig. 5. Evolution of automated testing during the years

They named this process as "Continuous Testing". It provided rapid feedback to developers regarding errors that they have inadvertently introduced. This CT definition is the base of other CT definitions and it is also used for other authors nowadays. For example, in 2016, S25 uses the same term to refer "a process that provides fast feedback about the quality of the code by running regression tests in the background automatically while the developer is changing the source code".

In 2010, S29 takes the same CT definition from S42 and S17: "running test cases all the time during development process to ensure the quality of the software system that is built". However, they mentioned that CT cannot be completed before software goes to users. Thus, the authors in S29 presented a new concept: life-long total continuous testing. It includes not only the unit testing stage, but also the following testing stages: specification testing, design testing, coding testing, validation testing, functional testing, non-functional testing, installation testing, operation testing, support testing, and maintenance testing.

In 2011, using the CT process proposed by Smith (2000), S6 presents CT for cloud computing. The authors state that CT can be used to test SaaS applications. As applications may be composed from services, CT can be applied before and after application and service composition, and even during its execution like a monitoring service. Later, in 2013, for Google (S30), CT means "running any kind of test as soon as possible, and detecting any type of issues related to a change made by a developer".

Between 2015 and 2016, several new CT approaches appeared: S10, S13, S14, S53 and S55. In S10, CT means "using automated approaches to significantly improve the speed of testing by taking a shift-left approach, which integrates the quality assurance and development phases". This approach may include automated testing workflows that can be combined with metrics in order to provide a clear picture of the quality of the software being delivered.

Leveraging a CT approach provides project teams with feedback on the quality of the software that they are building. It also allows them to test earlier and with greater coverage by removing testing bottlenecks such as access to shared testing environments and having to wait for the UI to stabilize. According to S10, "CT relies on automating deployment and testing processes as much as possible and ensuring that every component of the application can be tested as soon as it is developed".

For S14, CT involves testing immediately at integrating the code changes into the main trunk, and running regression test suites at any time to verify that those changes do not break existing functionalities. For S53, CT means to automate every single test case. Manual testing processes must be evaluated for possibilities of automation. Software delivery processes should be able to execute the test suite on every software build without any user intervention thereby moving towards the last goal of being able to deliver a quality release quickly. This whole principle of CT presented in S53 not only moves the testing process early in the cycle but it also allows the tests to be carried out on production-like systems (complemented by CDP).

S13 presents CT for mobile development as "the process of running all of the tests continuously (every few hours) on both the Master and the Release branch". The most important of these tests are the full build tests, integration regression tests, and performance tests in a mobile device lab.

Finally, in S55, the authors cited a part from a CI book [48], where the authors state that "in a continuous integration setting, which most of the organizations either adopted it or are trying to adopt it, testing should be run continuously". This CT approach includes unit tests, integration tests, functional tests, non-functional tests, and acceptance tests.

Discussion of RQ1

The results show that the concept of CT has been evolving during the years. At the beginning, it was only applied to the execution of unit tests continuously, especially in the developer's workstation while he/she codes in the background. Now, it does not apply only to unit testing, but also to every type of test case that can be automated. This may indicate that the inclusion of different testing stages during the years in CT definitions is related to the emergence of automation tools that allow teams to automate different types of test cases (see Fig. 5).

However, most of these articles based their CT approaches on Saff and Ernst definition (S42) and some of them use the definition proposed by Smith [47]. Thus, it can be concluded that CT is the process of running any type of automated test case as quickly as possible in order to provide rapid feedback to the developer and detecting critical issues before going to production.

3.4 RQ2. What Types of Testing or Testing Levels Have Been Implemented for Continuous Development Environments?

The International Software Testing Qualifications Board [49] proposes 4 testing levels: unit testing, integration testing, system testing and acceptance testing. At the same time, these testing levels have testing sub-levels.

These levels are used in CD through the implementation of specific testing stages that include different types of testing.

The earlier first stage that appears in the results of the SLR is *peer review*. It was proposed as a quality assurance stage in CD by S2. It's a manual stage that can be supported by tools. For example, S2 uses Gerrit for peer reviewing. Similarly, S13 uses *code review* as a requirement before any code can be pushed to the mainline trunk.

The second stage is *build and unit testing*. This testing stage has been applied by most of the studies included in this SLR (S2, S6, S9, S11, S12, S13, S14, S22, S32, S39, and S47). S22 has extended this stage by using automated mutation testing. Mutation testing is a process by which existing code is modified in specific ways (e.g., reversing a conditional test from equals to not equals, or flipping a true value to false) and then the unit tests are run again. If the changed code, or mutation, does not cause a test to fail, then it survives. Tools reported for unit testing are: JUnit (S2, S13), Robolectric (S13), NUnit (S9), Xcode (S13), Microsoft MSTest (S9), Android Studio (S13), XCTest (S13). Tools used for mutation testing are: PIT Mutation Testing (S22), Ninja Turtles (S22), and Humbug (S22). S9, S11, S12, S22 and S47 have also added to this level, a *code coverage* stage which is ran at the same time with unit tests. Unit test code coverage is measured using tools such as JaCoCo (S22), CodePlex Stylecop (S9), Coverage.py (S22), or NCover (S22). The third stage is *static code analysis* or simply static analysis. It was implemented by S2, S12, S13 and S47.

It is another automated stage that examines the code without running it, detecting coding style issues, high complexity, duplicated code blocks, confusing coding practices, lack of documentation, etc. S22 states that "static analysis allows manual code reviews to concentrate on important design and implementation issues, rather than enforcing stylistic coding standards". An alternative name to this stage is Code Verification (S47). The most implemented tool for this stage is SonarQube (S2, S12, S13, and S22).

The fourth stage is *integration testing*. It is an automated testing stage implemented by S2, S6,

Table 5. Solutions for time-consuming unit testing

Solution	Articles that have implemented the solution	Degree to which the solution solves the problem
Test case generation	S1, S15	Partial
Test case prioritization	S6	Partial
Running unit tests in the background at coding	S16, S17	Partial
Running groups of unit tests in the background at coding	S25	Total

S13, S14, S32 and S47. In this stage, individual software modules are combined and tested as a group. JUnit (S2) and TestNG (S2) have been reported as tools for supporting this stage.

The fifth stage is called by some authors as *Installation or Deployment testing*. It was implemented by S14 and S22. The goal of this stage is to verify whether software installation or deployment to a specific environment was made properly. It is a very short verification stage and the involved tools are the same as those that are used for unit or integration testing.

The sixth stage is *functional testing*. The goal of this stage is to verify that the functionalities of the system work as expected. This stage has been implemented using automated testing tools by S2, S9, S11, S12, S13, S14, S22, S39, S43 and S47. However, some of the studies (S9 and S13) were not able to automate all of the functional test cases. Thus, they use both manual and automated testing for this level. The stage has been named also as conformance testing (S13 and S43), feature verification (S14), system functional testing (S14), and functional acceptance testing (S9, S11, and S22). S22 states that it’s also important to add negative testing to this stage. The negative testing ensures that the system can handle invalid inputs or unexpected user behaviors. Furthermore, it has been proposed other testing sub-stages as part of the functional testing stage, like snapshot testing (S13). The goal of snapshot testing is to generate screenshots of the application, which are then compared, pixel by pixel, to previous snapshot versions. Testing tools used at this level are: JUnit (S2), NUnit (S9), MbUnit (S9), XUnit (S9), or Borland Sil4Net (S9).

The seventh stage is *security testing*. It is used to verify that security requirements such as confidentiality, integrity, authentication, authorization, availability and non-repudiation are met. This stage was implemented by S9 and S39.

The eighth stage is *performance, load and stress testing*, and it was implemented by S2, S9, S11 S13, S14, S39, S43 and S47. This stage is performed to determine a system's behavior under both normal and anticipated peak load conditions. The tools used for this stage are Jmeter (S2) and Borland Silk Performer (S9). An additional stage at this level is *capacity testing*. Capacity testing is targeted at testing whether the application can handle the amount of traffic that should handle. It was implemented by S9, S39 and S13. We will name this stage as CPLS testing (capacity, performance, load and stress).

Finally, the last stage is *exploratory manual testing*. Manual testing becomes more important since automated tests will cover the simple aspects, leaving the more obscure problems undiscovered. This stage was implemented by S9, S11 and S22.

Discussion of RQ2

These stages have been implemented for different types of platforms: web applications, mobile applications, cloud computing, web services, big data applications, etc. The testing stages implemented in the studies are shown in Fig. 6.

Unit testing, functional testing and CPLS testing are the most used stages in continuous software development environments.

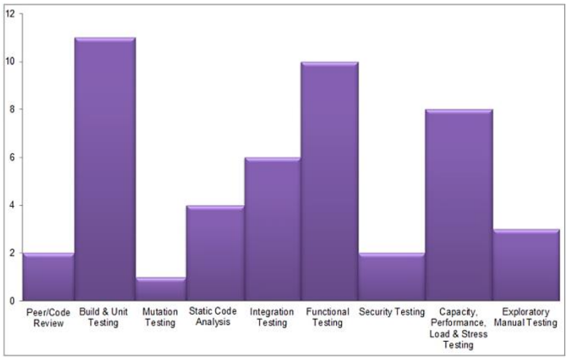

Fig. 6. Testing stages implemented by the studies in Continuous Software Development

3.5 RQ3. What Solutions Have Been Reported to Solve Testing Problems in CD?

Different solutions have been proposed to solve the testing problems presented in Table 1. We analyze those solutions for each problem separately.

It is very important to highlight that more solutions can be found in the literature, but we just describe those which are related to continuous software development environments.

3.5.1 RQ3-P1. Time-Consuming Testing

In any continuous software development environment, changes are introduced more frequently to the repository, so it is necessary to run regressions as quickly as possible. However, the execution of a huge suite of test cases takes too much time, even if the test cases are automated. Furthermore, this problem is not tied up to one single testing level, but to all of the different mentioned testing stages. Thus, we analyze the different solutions by grouping them according to the testing levels.

Unit testing. S1 and S6 have proposed the use of automatic test case generation techniques to face this problem. By having a complete automated unit test case generation system, it is possible to reduce the cost of software testing and it also facilitates the test case writing. S1 presents an automatic test case generation mechanism for javascript programs. One of the main problems of this tool is that it is not applicable for object-oriented languages. On the other hand, S15 proposes an automatic test case generation system for object-oriented languages using search-based testing and a mechanism called continuous test generation as a synergy of automated test generation with CI. For this purpose, it is presented a tool called EvoSuite. However, the authors describe that it is not applicable for inner classes and generic types.

S6 presents a unit test prioritization technique, where test cases can be ranked to help the users to select the most potent test cases to run first and often. However, the developer has to make the prioritization manually, and that is a time-consuming task.

S16 and S17 propose a mechanism that consists in running unit tests in the background while the developer is coding. S17 presents an eclipse plugin for Java projects that automatically compiles the code when the user saves a buffer, and then it indicates compilation errors in the eclipse text editor and in the task list, providing an integrated interface for running JUnit test suites. Similarly, S16 uses the same approach for .Net code but combined with TDD, where the developer has to write the tests before writing the code. Finally, S25 improves this approach by adding an oriented test selection strategy using plugins. Each module has its own plugin, and it is not necessary to run all of the unit tests but only the ones related to the affected module. By using naming conventions, the proposed framework can find the test plugin for a specific module, and code coverage tools can detect modified classes.

Thus, this approach impacts on the execution times of the unit tests, reducing them and it does not impact on the test case writing stage.

A summary of the proposed solutions for time-consuming unit testing is shown in Table 5.

Functional Testing. S6, S27 and S41 propose test case grouping and segmentation techniques to face the time-consuming functional testing stage. Tests are grouped in different suites based on functionality and speed. In this way, most critical tests can be run first and developers get fast feedback from them. Non-critical and slower tests run later (only if the first ones have passed). Thus, the test segmentation partially solves the time-consuming testing problem.

Table 6. Solutions for time-consuming functional testing

Solution	Articles that have implemented the solution	Degree to which the solution solves the problem
Test case grouping/segmentation	S6, S27, S41	Partial
Test case parallelization	S6, S13, S22, S27, S41, S43, S50, S56	Total
Automatic test case selection	S25, S44	Partial
Automatic test case prioritization	S23, 24, S34, S45, S46, S48, S52	Partial
Automatic test case selection and prioritization	S31	Partial
Running tests continuously in a build server	S29	Partial
Testing as a service (TaaS)	S26, S36	Total
Automatic test case selection, prioritization, and parallelized run in TaaS	S30	Total
Test case optimization	S52	Partial
Browser rotation (web only)	S27	Partial
Use of APIs	S27	Partial

Another alternative for this problem is the use of parallelization (S6, S13, S22, S27, S41, S43, S50 and S56). Executing automated tests in parallel (instead of serially), decreases the amount of time at running the tests. The tests execution is distributed through different computers. S56 proposes the use of virtualization as an alternative for having just one computer and distributing the test execution through virtual machines. However, both approaches require considerable hardware.

S25 presents a framework called Morpheus which reduces the feedback time and provides test results for only changes committed by the developer. It reduces the number of tests to be executed by selecting only the ones that are related to the changed source code. One of the strategies they have implemented for the framework is the Requirement Oriented Test Selection Strategy: the changed source code can be linked with the user story, bug fixing requirement or feature, which can be also linked with the test cases. Another test case selection technique is proposed by S44, where the selection is based on “the analysis of correlations between test case failures and source code changes”. However, none of the two approaches have solved the problem for changes that have an effect on the entire system.

S23, 24, S34, S45, S46 and S48 propose the automatic test case prioritization technique to face the time-consuming problem, using different prioritization methods. S23, S24 S45 and S48 implemented prioritization based on history data to determine an optimal order of regression tests in the succeeding test executions.

On the other hand, S34 uses prioritization based on business perspective, performance perspective and technical perspective. Finally, S46 presents a tool called Rocket which executes the tests that take a shorter time earlier. However, the results of the prioritization techniques’ implementation show that while they solve the early detection of critical defects, they do not solve the time-consuming problem at running the whole suite of test cases.

Other alternatives were implemented in S26, S27, S29, S52 and S56. S29 proposes an approach called long-life CT that uses Artificial intelligence (AI) methods at different levels. The approach consists of running test cases all the time on a build server and detecting issues via AI techniques. S52 proposes a test suite optimization technology called TITAN. S27 proposes browser rotation as an alternative for increasing speed in the execution of web UI testing. Rotating the browsers between consecutive builds can gradually achieve the same coverage as running the tests in every single browser for each build. S27 also propose the use of REST APIs in tests when some of the test cases require configuration initialization by using the UI in the application. Utilizing REST APIs can reduce the testing duration of some tests scripts because the operations are considerably faster when compared with performing them through the UI.

S26 and S56 propose Testing as a Service (TaaS). Both S26 and S56 state that running automated tests in parallel is one of the best solutions for time-consuming testing, but it requires hardware resources. One of the main advantages of TaaS over traditional testing is its scalable model via the cloud: it utilizes computing power, disk space and memory as per current requirements but it has the ability to ramp up on demand very quickly. Thus, running tests in parallel is not a problem. Furthermore, TaaS supports multiple types of automated tests and reduces the cost of in-house testing.

Furthermore, combinations of the aforementioned techniques are proposed. S31 presents an approach that uses test selection and test prioritization techniques and integrates different machine learning methods. These methods are: test coverage of modified code, textual similarity between tests and changes, recent test-failure or fault history, and test age. However, the approach carries with it the problems of test selection and test prioritization. Finally, S30 shows how Google faced these problems by adding to test selection and test prioritization, the execution of test cases in the cloud (TaaS).

A summary of the proposed solutions for time-consuming functional testing is shown in Table 6.

Manual Testing. Automated testing has been the solution for time-consuming manual testing for years. There are lots of tools that allow developers to automate both functional and non-functional test cases. However, apart from functional and non-functional automated testing, two new approaches were found in the literature:

1. Automation of negative scenarios (S22): manual testing is used to perform exploratory testing that includes negative scenarios (non-happy path scenarios). However, negative testing can be automated and that reduces the time in manual testing stages.
2. Prioritization techniques for manual black-box system testing (S40): coverage-based testing, diversity-based testing and risk-driven testing. The results show that the risk-driven approach is more effective than the others, in the context of continuous software development environments. The risk-driven approach uses the historical fault detection information and it requires access to the previous execution result of the test cases.

Discussion of RQ3-P1

There are many studies which have proposed solutions for time-consuming testing. For unit testing, running the tests in the background while the developer codes seems to be the best solution. As an addition to this technique, test cases can be generated automatically in order to decrease the unit tests creation process.

Regarding functional testing, the use of parallelization decreases the amount of time at running the tests. However, it requires considerable hardware resources. TaaS can solve this problem because it uses resources on demand, but it is more costly.

Finally, if test selection and test prioritization techniques can be added to parallelization or TaaS, they will improve the testing process.

For manual testing, negative scenarios should be automated. Also, test prioritization techniques can be incorporated for the manual testing process.

3.5.2 RQ3-P2. Automated Flaky Tests

An important characteristic of an automated test is its determinism. This means that a test should always have the same result when the tested code does not change. A test that fails randomly is not reliable and it is commonly called “flaky test”. Automated flaky tests slow down progress, cannot be trusted, hide real bugs and cost money.

Table 7. Pros and cons of solutions for automated flaky tests

Solution	Pros	Cons
Test prioritization and test selection.	(1) It reduces the number of flaky tests in the test-suite execution.	(1) Flaky tests still exist. (2) Flaky tests are not identified.
Running tests only for new or modified code.	(1) Flaky tests are easier to identify and ignore.	(1) Flaky tests still exist.
Test the tests for flakiness.	(1) Flaky tests can be identified and ignored. (2) It is possible to determine the cause of flakiness. (3) Flaky tests can be removed or fixed.	(1) Cost and Time
Re-running tests.	(1) It reduces the number of failures due to flaky tests.	(1) Time (2) Flaky tests still exist.
Postpone tests re-runs to the end of the execution.	(1) It reduces the number of failures due to flaky tests. (2) It is possible to determine the cause of flakiness.	(1) Time (2) Flaky tests still exist.

S25 proposes a framework which runs only tests related to a new feature or a modified functionality. In this way, the number of flaky tests is significantly reduced. However, this framework only reduces the number of flaky tests, but it does not skip them or fix them. Similarly, S31 proposes an approach that performs test selection and test prioritization using different techniques: coverage of modified code, textual similarity between tests and changes, recent test-failure or fault history, and test age. The approach tracks the failures by coverage, text and history. When it finds failures that are not related to the new or modified code in terms of coverage or text similarity, it means that those failures belong to an automated flaky test. Thus, it can detect flaky tests that can be skipped. S45 also uses test prioritization and test selection techniques, avoiding breaking builds and delaying the fast feedback that makes CI desirable.

Automated tests also can be tested for flakiness (S41). For example, S7 proposes a mechanism for flaky tests classification, so that it is better to analyze them. The authors of S7 have studied common root causes of flaky tests and fixes for them. The goal of the authors is to identify approaches that could reveal flaky behaviors, and describe common strategies used by developers to fix flaky tests. However, testing tests introduces effort and time.

Another common strategy is re-running tests. Google for example (S30) has a system which collects all the tests that fail during the day and then it re-runs them at night. It is possible to see whether they really are flaky tests or not.

They also use a notification system where the running history of the test that has failed is listed, so the developer can see if it is a flaky test or not. Google also has implemented flakiness monitoring: if the flakiness is too high, the monitoring system automatically quarantines the test and then it files a bug for developers.

Finally, S21 proposes and evaluates approaches to determine whether a test failure is due to a flaky test or not:

- Postponing the test re-runs to the end of the test-suite execution. At this time, more information is available and re-runs can be avoided altogether.
- Re-running the tests in a different environment.
- Intersecting the test coverage with the latest change: If a test that has passed in a previous revision fails in a new one, and if the test execution does not depend on the changes introduced in that revision, it can definitely be concluded that the test is flaky. If the test

Table 8. Solutions for User Interface Testing Problems

Solution	Benefits	Drawbacks
Use of APIs for preconditions instead of UI testing steps.	(1) Execution speed and robustness. (2) Decreasing of flakiness. (3) Reduction in the amount of UI testing steps.	(1) At running regression suites, changes in the UI element to verify may cause test to fails.
Model-based approach.	(1) Flexibility. (2) Execution speed and robustness. (3) If changes are made to the application's source code that breaks the model, the developer will receive a compilation error.	(1) It does not verify that the GUI rendering is correct. (2) Interactions with the application during testing are not performed in the same way as a user interacts with the software. (3) It needs synchronization between the test cases and the application under test.
Visual GUI Testing.	(1) Tests are easy to create. (2) Flexible. (3) It can be used on any GUI-based system regardless of its implementation or even platform. (4) Changes in the code of the GUI will not make the test fail.	(1) Synchronization between test script and the application state transition. (2) Images dependency. (3) Lack of functionality or instability. (4) Limited online support available for VGT tools.
Image Comparison.	(1) Easy to implement. (2) It is very accurate at detecting UI changes.	(1) If it is used as a testing tool, it may cause false-positives because of minor differences caused by UI rendering or other components like advertisements.
Crowdsourcing GUI testing.	(1) No automation testing needed. (2) There are not flaky tests for GUI. (3) Maintenance is not required.	(1) Dependency on external users. (2) The users do not know about the business. (3) Indeterminate testing time.

depends on the change, it cannot be concluded whether the test is flaky or not.

Discussion of RQ3-P2

There are different solutions for automated flaky tests. However, according to the analyzed articles, all of the solutions have pros and cons (see Table 7), and there is no a perfect and accepted solution for flakiness yet.

3.5.3 RQ3-P3. User Interface Testing Problems

"High-level tests such as UI acceptance tests are mostly performed with manual practices that are often costly, tedious and error prone" [23]. Test automation has been proposed as an alternative to solve these problems. However, the UI is the part of an application that changes most frequently and it can drive to flaky automated tests. Thus, several solutions were proposed in order to face this problem. We analyze them, we summarize them with their benefits and drawbacks in Table 8.

In a CD pipeline, where automated tests are executed many times per day, test stability is a key factor for achieving sufficient throughput. S9 presents a problem related to the low stability of tests interacting with UI elements.

The authors of S9 state that "stable tests demand a minimum level of testability, which is sometimes hard to achieve when testing at the UI level". They have improved the testability by providing additional interfaces for accessing the application under test at API level. Preconditions steps can be performed using APIs and the particular behavior to be tested is performed through the UI.

This solution reduces the number of unnecessary UI testing steps. The same approach is proposed by S27, where the authors state that "configuration steps can be executed by using

REST APIs (if possible) to make them more reliable and thus reduce unnecessary failures".

In S33, UI testing problems are tackled by "adopting a model-based approach, centered on models that represent the data manipulated by the UI and its behavior". The models are read by a compiler, which after a set of tasks produces the java source code of a test harness. Using the test harness, the developer can write high level test cases, which can be ran using Selenium as a driver. This approach moves most of the fragility factors from the source code to the models, where handling them is more effective, and lets the compiler to generate an up to date test harness.

On the other hand, S8 presents Visual GUI Testing (VGT). VGT is "a test technique that uses image recognition in order to interact and assert the correctness of a system under test through the bitmap GUI that is shown to the user on the computer monitor". The use of image recognition allows the technique to be used on any GUI-based system regardless of its implementation or platform.

Different from second-generation GUI-based testing tools (like Selenium or Sahi), changes in the UI code will not make the test fail. However, several challenges for VGT are presented in S8:

- Maintenance of test scripts (not only for VGT, but for automated testing in general).
- Synchronization between test script and the application state transition.
- Image Recognition: According to the authors of S8, it has been empirically observed that VGT tools sometimes fail to find an image without reason, producing false-positive test results.
- Instability.
- Lack of online support.

In S37, it is presented the results of a case study focused on the long-term use of VGT at Spotify. Due to the challenges mentioned in S8, they decided to abandon VGT in most of the projects and they started to use a model-based approach by implementing a tool called GraphWalker. In S37, the benefits and drawbacks in relation to these techniques are mentioned, which are detailed in Table 8.

S38 proposes an approach called "Perceptual Difference (PD) for Safer Continuous Delivery in UI applications". PD combines Computer Vision concepts with CI to enable recognition of UI-based changes, assisting testers to check development branches before deploying the application. Image Difference is defined by S38 as "a simple Image Processing technique that involves subtracting one image from the other". According to the authors, "this process is very useful in identifying changes in an image". They propose a tool called pDiff in order to incorporate Image Difference into the CD pipeline. As UI changes are difficult to manually keep track of via code, pDiff adopts a solution which consists in storing screenshots of the live and staged versions of the application. After the results of the comparison are generated, the tester can access the pDiff web application in order to check its status and manually approve or reject each difference marked by the tool.

Another completely different approach was presented in S18. It presents a crowdsourcing GUI test approach. The authors of this article state that "it is possible to outsource GUI testing to a very large pool of testers (users) scattered all over the world". They have implemented a prototype implementation on Amazon's Mechanical Turk (MTurk). MTurk is a crowdsourcing marketplace that allows requesters to submit Human Intelligence Tasks (HITs) which will be performed by users against a fee.

When users accept a GUI testing task through the MTurk website, they are presented with a web page that shows the display of a VM running the GUI under test, allowing mouse and keyboard interactions with it. The VM runs on a remote server and it is instantiated automatically. The users are asked to execute a sequence of steps described in the task and then to report the results. The interaction of the users with the VMs is captured by recording the displays of the VMs, allowing developers to analyze and reproduce reported problems.

Discussion of RQ3-P3

A vast amount of approaches has been proposed to face UI testing problems. However, all of them have benefits and drawbacks that can be seen in Table 8.

Nevertheless, the use of APIs (like REST services) for running test preconditions instead of UI steps can significantly contribute to the test robustness and thus reduces flakiness. It also decreases execution time. Thus, using APIs for precondition steps might be the best solution for UI tests.

3.5.4 RQ3-P4. Ambiguous Test Results

Test results have to be communicated to developers properly, indicating whether the tests have passed or not. It has to be clear what exactly has broken a build. When a test result does not fulfill these requirements, then it is an ambiguous test result.

S8 presents techniques that automate the results analysis process. It uses examination and analysis of crash dump files and log files to extract consistent failure summaries and details. The authors of S8 called this set of techniques as "Automated Test Results Processing" and achieving it requires three steps:

1. Applying consistent methods that improve the effectiveness of the automated tests.
2. Designing concise problem reports to allow testers to identify duplicates problems quickly. Concise problem reports also provide developers with the information they need to isolate defects.
3. Automating the test results analysis to collect the data required to build concise problem reports.

Another approach is presented in S25. It presents a solution that improves the quality of the feedback with the test results by:

1. Executing tests with the product build in a production-like environment.
2. Providing only test results for the changed code of the developer.
3. Providing information whether a test has failed because of the developer's last change or because of a previous change.

These feedback reports include:

- Information about the change set of the commitment which triggered the test run: change log, committed files, etc.
- An overview about the executed tests with their results (success or failure). It is also provided data about how often a test has already failed.
- URLs to different web pages with detailed information about the failed tests, including which exception has been thrown and its stack trace.
- A code coverage report of the executed tests, so that the developer will be able to verify whether the tests have really executed the modified source code and whether the code coverage of the written test is good enough.

S27 recommends improving also the failure messages and the name of test classes/methods for Selenium UI tests. The authors of S7 state that exact error messages could make test result failure analysis easier. Sometimes the tests fail because an exception thrown from a Page Object. The exception messages from these Page Objects range from custom messages to detailed exceptions from Selenium. The Selenium exceptions often reveal the root cause, for example, a certain HTML element was not found on the page. However, the person doing the failure analysis may not be able to recognize the element by the web element selector that is mentioned in the error. Thus, adding custom messages to exceptions may better serve the clarity if they are precise enough. The name of the test is also import because it is used in the test results and a descriptive name would greatly improve the readability of these results.

Finally, S41 reports two techniques to face ambiguous test results:

1. Test adaptation: the test suites are segmented and then adapted based on the history of test runs. According to the authors, it solves time-consuming testing by running the most critical tests first and others later only if the first tests pass. They also state that "when a high-level test fails, it might be difficult and time-consuming to find out why the fault occurred". Therefore, it is advised that low-level tests should be able to give the cause of the failure.
2. Commit-by-commit tests: every introduced change in the repository should be tested individually, so when tests fail it can be directly detected which change caused the failure.

Discussion of RQ3-P4

According to the literature, the ambiguity of the results can be improved by using reports where:

- The cause of the failure is described in a precise way.
- It is shown only the status of the tests related to the developer's change.
- Flaky test results are not shown.
- Custom messages are presented instead of exceptions and stack traces.
- URLs to web pages and screenshots where the tests have failed are also provided.

3.5.5 RQ3-P5. Rich Internet Applications and Modern Web Applications

Modern web applications utilize new technologies like Flash, Ajax, Angular or they perform advanced calculations in the client side before carrying out a new page request. Testing these dynamic technologies is challenging but some solutions have been proposed.

For the Ajax challenges, S5 recommends a CT process as a best practice. To accomplish this, the required infrastructure should facilitate three activities:

1. Testing the application code while it is being built on the server.
2. Testing the server by mimicking client behavior.
3. Testing not only each component of the browser, but also their interactions.

These activities should be performed every night as part of an automated build process. Since many of the tests in this process require code deployment to a production or production-like server, deployment should be also automated.

Furthermore, S54 presents a solution for websites with accessibility. Accessibility is a non-functional requirement for web applications. However, according to the authors of S54, "current accessibility automatic evaluation tools are not capable of evaluating DOM dynamic generated content that characterizes Ajax applications and Rich Internet Applications (RIA)". In this context, S54 describes an approach for testing accessibility requirements in RIA, by using acceptance tests. The approach adds a set of assistive technology user scenarios to the automated acceptance tests, in order to guarantee keyboard accessibility in web applications. These tests provide an end-to-end accessibility analysis, from server-side to client-side implementations (javascript and dynamically generated DOM elements) in RIA. As the tests are automated, they can be incorporated in a CD process.

Discussion of RQ3-P5

Even though both of the aforementioned studies present approaches for problems based on RIA and modern web applications, they do not present solutions for specific dynamic content challenges.

In [50] for example, a framework that is composed by Selenium and TestNG is proposed. It shows that Selenium has a feature that allows tests to implement three different waiting and timeouts configurations, so that they will not fail because of dynamic content. This feature will wait until the application gets its final state before continuing with the next step or verification.

3.5.6 RQ3-P6. Big Data Testing Problems

Big data is the process of using large datasets that cannot be processed using traditional techniques. Testing these datasets is a new challenge that involves various tools, techniques and processing frameworks.

S3 presents two problems with big data testing, and hadoop-based techniques that may be solutions for them:

1. "Processing big data takes a long time". One possible solution is test data generation using Input Space Partitioning with parallel computing. The process starts with an input-domain model (IDM). Then, the tester partitions the IDM and selects test values from the partitions. Finally, a combinatorial coverage criteria is applied to generate tests.
2. "Validation of transferred and transformed data is difficult to implement". Transferred data can be tested by checking the number of columns and rows, the columns' names, and the data types. If the data source and the target data are provided, this validation can be automated. On the other hand, there are some workarounds to test transformed data, but it is still a challenge. Some of the approaches are:

   a. To validate whether the target data has correct data types and value ranges at a high level by deriving data types and value ranges from requirements, then generating tests to validate the target data.

b. To compare the source data with the target data to evaluate whether or not the target data was transformed correctly.

In S4, quality assurance techniques for big data applications are presented. First, the authors of S4 list quality attributes for big data applications: data accuracy, data correctness, data consistency and data security. Then they present the quality factors of big data applications: performance, reliability, correctness and scalability. Finally, they discuss the methods to ensure the quality of big data application: model-driven architecture (MDA), monitoring, fault-tolerance, verification and prediction.

Discussion of RQ3-P6

Trivial solutions for transferred data and transformed data were proposed. One of the studies has also proposed test data generation using “Input Space Partitioning” with parallel computing in order to decrease the processing data time testing. However, it was not considered other big data stages such as data streaming, data enrichment, data storing in distributed nodes, data analysis or graph processing.

Finally, it has been proposed quality attributes and quality factors that can be considered at the time of testing big data systems.

3.5.7 RQ3-P7. Data Testing Problems

Data is very important for different types of systems and errors in these systems are costly. While software testing has received highly attention, data testing has been poorly considered.

S41 reports a partial solution for data testing problems using database schema changes testing. In S19, the authors propose an approach called Continuous Data Testing (CDT), in which a tool run test queries in the background, while a developer or database administrator modifies a database. This technique notifies the user about data bugs as quickly as they are introduced, leading to three benefits:

1. The bug is discovered quickly and can be fixed before it causes a problem.
2. The bug is discovered while the data change is fresh in the user’s or administrator’s mind, increasing the chance to fix the bug quickly.
3. Contribute to poor data documentation.

According to the authors of S19, “CDT can discover multiple kinds of errors, including correctness errors and performance-degrading errors”. They conclude that “the goal is not to stop errors from being introduced, but to shorten the time to detection as much as possible”.

S49 presents an approach called TDD for Relation Databases. To extend TDD practice to database development, database tasks equivalent to regression testing, refactoring and CI are necessary:

– In database regression testing, the database is validated by running a comprehensive test suite that includes:

– Interface testing. From the database’s viewpoint, these are black-box tests that verify how systems will access the database.

– Internal testing. There are tests that verify data and database behavior.

– In database refactoring, a simple change is made to a database that improves its design (while keeping its behavioral and informational semantics).

Table 9. Non-Functional Requirements considered for CD

Non-Functional Requirement	Tool/Technique/Approach	Article
Maintainability	SonarQube, Gerrit	S2, S13, S22, S47, S9, S11, S39, S12
Performance	JMeter	S2, S13, S47, S43, S9, S11, S39
Correct Installation/Deployment	JUnit, TestNG, XUnit, NUnit	S14, S22
Compatibility	Cross-Browser Testing	S14, S39
Localization and Internalization	G11N/L10N Testing	S14
Load	JMeter, Grinder, Gatling	S14, S43, S9, S11
Stress	JMeter, Gatling	S14, S9
Documentation	Own Framework	S14
Security	Own Framework	S22
Accessibility	Own Framework	S54
Usability	Own Framework	S22

– In continuous database integration, developers integrate their changes to their local database instances, including structural, functional, and informational changes. In any case, whenever someone submits a database change, tests should verify that the database keeps stable. After that, everyone else working with the same database should download the change from the configuration management system and apply it to their own local database instance as soon as possible.

The authors of S49 also present the best practices for continuous database integration:

– Automate the build.

– Put everything under version control (data scripts, database schemas, test data, data models, and similar artifacts).

– Give developers their own database copies.

Discussion of RQ3-P7

According to the literature, data testing can be performed in the same way as in conventional software testing. Data testing can be automated and incorporated into the CI server. Data artifacts such as scripts, schemas or models have to be added to the version control system.

Every change on these artifacts should trigger the automated tests (including data testing) as soon as possible.

Also, we believe that the quality attributes and quality factors proposed for Big Data testing problems can be considered for Data testing.

3.5.8 RQ3-P8. Mobile testing problems

Mobile testing has brought with it a lot of challenges regarding the testing process, the testing artifacts, the testing levels, the type of testing, the different type of devices, and the costs of automated testing. Because of these challenges, a few proposals for mobile testing have emerged.

S13 presents the continuous deployment process of mobile applications at Facebook. In that article, the authors mention that testing is particularly important for mobile apps because:

1. Many mobile software updates are made each week.
2. There are hundreds of mobile devices and operating systems where the software has to run on.
3. When critical issues arise in production, there are just a few options to deal with them.

According to S13, “Facebook applies numerous types of tests, including unit tests, static analysis

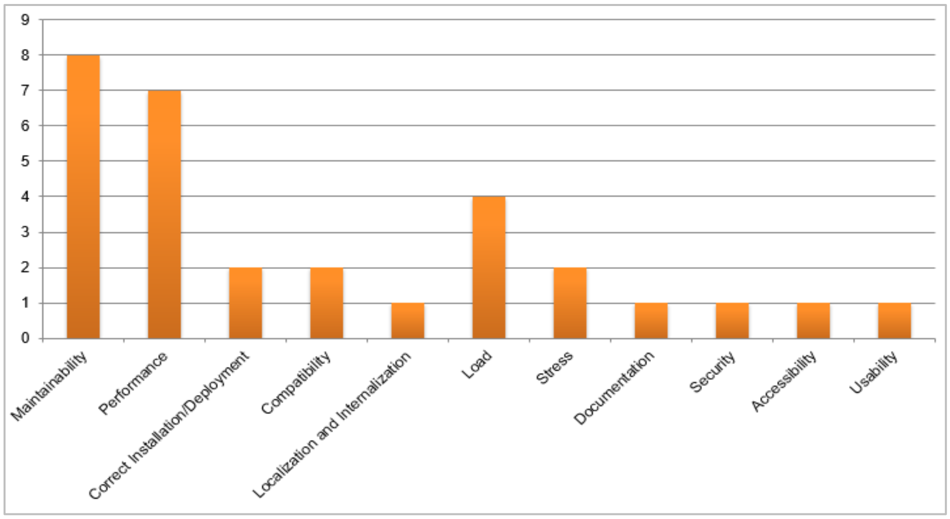

Fig. 7. Non-Functional requirements in CD

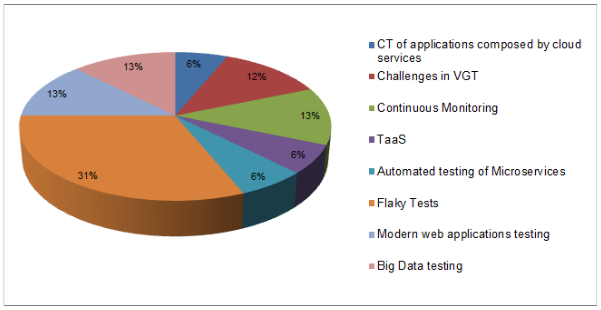

Fig. 8. Open Issues in Continuous Testing

tests, integration tests, screen layout tests, performance tests, build tests, and manual tests". These tests are automated and they are run in hundreds of nodes using simulated and emulated environments. For performance testing on real hardware, Facebook uses a mobile device lab, with a primary focus on characteristics of the application such as speed, memory usage, and battery efficiency. The mobile device lab contains electromagnetically isolated racks. Each rack contains multiple nodes which are connected to real mobile devices with different operating systems. Thus, Facebook's testing strategy encompasses the following principles:

- Coverage: testing has to be performed as extensively as possible.
- Responsive: regression tests have to be run in parallel. The goal is to provide the developer with the results from smoke-tests within 10 minutes of his/her actions.
- Quality: tests should identify issues with precision. Flaky tests need to be minimized.
- Automation: automate as many tests as possible.
- Prioritization: tests have to be prioritized since they use too many computing resources in regressions.

S35 proposes a novel framework that can be applied to test different mobile browsers and applications using a tool called Appium. The novel framework works for native, hybrid and mobile-web applications for iOS and Android systems. It is a data driven test automation framework that uses the Appium test library to automatically test mobile applications. According to a study made by the authors of S35, "the framework will improve the testing process for mobile applications, save time and speed up the process of testing and release any mobile application in a short period of time".

Finally, S20 presents a combination of Appium and TaaS. The authors of S20, describe a case study of the MedTabImager tool using this framework to run UI tests. The tests ran against a cloud service (Sauce Labs). According to the authors of S20, "setting up Sauce Labs on the CI server with a specific plugin did not require much effort". A successful build (which includes unit tests) triggers the UI tests on Sauce Labs automatically. Even though tests are automated, manual testing cycles are required before releasing. To automate the app distribution S20 used "distribution of beta builds" using a cloud-based service called TestFairy4. For MedTabImager, the authors state that "the TaaS solution requires less setup effort and works fairly well".

Discussion of RQ3-P8

All testing levels and stages discussed for RQ2 can be considered for mobile testing. Mobile test cases can be automated using existing tools like Appium. They also can be parallelized and there are three approaches for mobile testing environments:

1. Emulated devices with different operating systems – for development environments.
2. Physical mobile devices – for production like environments.
3. Cloud service (like Sauce Labs).

3.5.9. RQ3-P9. Continuous Testing of Non-functional Requirements

While unit, integration and functional tests have been extensively discussed in the literature and widely practiced in CD, testing non-functional requirements has been overlooked. In Table 9, we present a list of the different non-functional requirements that were considered by the studies in the implementation of the CD pipelines. Fig. 7 also shows the amount of times they were considered, as a brief *discussion of RQ3-P9*.

3.6 RQ4. Are there open issues related to CT?

After the analysis of the 56 articles, we found different challenges in terms of open issues for CD. Some of these challenges are related to testing. We list them as follow:

- CT of applications composed by cloud services (S6): The number of available cloud services and the size of data they need to handle are often large. Testing workflows composed of those services is a challenge. Another concern is the necessity to provide Quality of Service (QoS) guarantees [51].
- TaaS in CD (S26): TaaS is a model in which testing is performed by a service provider rather than employees. The most popular TaaS tools are Sauce Labs, BlazeMeter and SOASTA CloudTest. BlazeMeter has presented in 2015, a solution for continuous delivery using TaaS called "Continuous Testing as a Service" (CTaaS) [52].
- Continuous Monitoring (S12, S53): with the capability to test early on a production like system, there is an opportunity to monitor several quality parameters throughout and hence ability to react to sudden issues in timely manner.
- Challenges in VGT (S28, S37): Currently no research has explored the benefits of using VGT for CT. However, used in long-term (years) projects, VGT has still many challenges to solve:
- Test scripts have limited use for applications with dynamic/non-deterministic output
- Test scripts require image maintenance.
- VTG tool scripts have limited applicability for mobile applications
- VTG tool scripts lock up the user's computer during test execution.
- Automated testing of Microservices (S32): According to the authors of S32, "Microservices concept is relatively new, so there are just few articles in the field of microservice validation and testing right now".

Discussion of RQ4

While open issues were found in the field of CT, some of the problems aforementioned in this article have not been fully realized. For that reason, we have added those problems to the list of open issues. This can be seen in Fig. 8.

It is also important to determine the maturity of automated testing in every test stage. In [53], a model to determine the maturity of test automation is proposed as an area of research and development in the software industry.

4 Conclusions and Future Works

In this paper, we have presented a SLR focused on CT and solutions for testing problems in CD.

Our first goal was to validate if an accepted definition for CT existed and if it was related to CD, like the other C-approaches (CI, CDP). The results show that the concept of CT has been evolving over the years. At the beginning, it was only applied to the execution of unit tests continuously and now it does not apply only to unit testing, but also to every type of test case that can be automated. Thus, it was validated that CT is the process of running any type of automated test case as quickly as possible in order to provide rapid feedback to the developers and detecting critical issues before going to production. This CT definition is one of the CD main goals, so we concluded that CT is directly related to CD.

We also looked for different testing levels or stages in CD. Unit testing, functional testing and performance, load and stress testing are the most used stages in continuous software development environments.

Our third goal was to look at proposals, techniques, approaches, tools and other kind of solutions for the different existing testing problems in CD. We found that many solutions have been proposed to face the mentioned problems. Time-consuming testing has been the most discussed problem by the articles, where new techniques, approaches and tools have been proposed to solve it. On the other hand, flaky tests, big data testing and the testing of modern web applications that use Flash, AJAX, Angular or similar technologies, are problems that were not completely solved yet. Nevertheless, we believe that the different studied solutions may mean a contribution to CD and a list of key success factors can be set by combining them properly.

Finally, we wanted to check whether open issues exist or not for CT. We set a list of 5 open issues for CT: CT of applications based on cloud services, challenges with VGT, continuous monitoring, TaaS in CD and automated testing of microservices. However, we consider that flaky tests, big data testing and modern web applications testing are challenges that need to be faced.

As future work, we will research on new approaches to face the mentioned open issues, and we will set a list of key success factors for testing in CD.

In addition, we will design standardized testing models for CD, using the different stages found for CT. We will work on a framework that will implement these models using the different existing testing approaches and the different solutions for testing problems that were found in this SLR.

Acknowledgements

This research was supported by GICS (Grupo de Investigación de Calidad de Software - F010-2013 Project) at Universidad Nacional del Nordeste.

We also thank Guy Finkill who provided language help for the preparation of this paperAppendix A. Selected Papers.

References

1. **Fowler, M. (2014).** https://martinfowler.com/bliki/ContinuousDelivery.html.
2. **Prusak, O. (2015).** *The Missing Link in the Continuous Delivery Process.* BlazeMeter. https://www.blazemeter.com / blog / continuous-testing-missing-link-continuous-delivery-process.
3. **Laukkanen, E., Itkonen, J., & Lassenius, C. (2017).** Problems, causes and solutions when adopting continuous delivery - A systematic literature review. *Information and Software Technology*, Vol. 82, pp. 55–79. DOI: 10.1016/j.infsof.2016.10.001.
4. **Humble, J. & Farley, D. (2010).** *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.*

Appendix A. Selected Papers

Table A.1. List of included articles

#	Ref	Name of the article
S1	[54]	A complete automation of unit testing for JavaScript programs
S2	[55]	A Practical Approach to Software Continuous Delivery Focused on Application Lifecycle Management
S3	[56]	A Scalable Big Data Test Framework
S4	[57]	A survey on quality assurance techniques for big data applications
S5	[27]	Ajax best practice: Continuous testing
S6	[58]	An approach for service composition and testing for cloud computing
S7	[59]	An Empirical Analysis of Flaky Tests
S8	[60]	Automated Test Results Processing
S9	[61]	Automated testing in the continuous delivery pipeline: A case study of an online company
S10	[62]	Continuous Architecture and Continuous Delivery
S11	[10]	Continuous Delivery: Huge Benefits, but Challenges Too
S12	[63]	Continuous delivery practices in a large financial organization
S13	[64]	Continuous Deployment of Mobile Software at Facebook
S14	[65]	Continuous Software Testing in a Globally Distributed Project
S15	[66]	Continuous Test Generation on Guava
S16	[67]	Continuous test-driven development: A novel agile software development practice and supporting tool
S17	[68]	Continuous testing in eclipse
S18	[69]	Crowdsourcing GUI tests
S19	[29]	Data debugging with continuous testing
S20	[70]	Designing an Android continuous delivery pipeline
S21	[71]	Determining flaky tests from test failures
S22	[72]	DevOps Advantages for Testing: Increasing Quality through Continuous Delivery
S23	[73]	Effect of time window on the performance of continuous regression testing
S24	[74]	Failure history data-based test case prioritization for effective regression test
S25	[75]	Fast feedback from automated tests executed with the product build
S26	[76]	Hard Problems in Software Testing Solutions Using Testing as a Service (TaaS)
S27	[77]	Improving Web User Interface Test Automation in Continuous Integration
S28	[78]	Industrial application of visual GUI testing: Lessons learned
S29	[79]	Intelligent Testing System
S30	[80]	Large-scale test automation in the cloud
S31	[81]	Learning for Test Prioritization: An Industrial Case Study
S32	[82]	Microservices validation: Mjolnirr platform case study
S33	[83]	Model-Based Continuous Integration Testing of Responsiveness of Web Applications
S34	[84]	Multi-Perspective Regression Test Prioritization for Time-constrained Environments
S35	[85]	Novel Framework for Automation Testing of Mobile Applications using Appium
S36	[86]	O!Snap: Cost-Efficient Testing in the Cloud
S37	[87]	On the long-term use of visual GUI testing in industrial practice: A case study
S38	[88]	Perceptual Difference for Safer Continuous Delivery
S39	[89]	Principles of Continuous Architecture
S40	[90]	Prioritizing Manual Test Cases in Traditional and Rapid Release Environments
S41	[3]	Problems, causes and solutions when adopting continuous delivery—A systematic literature review
S42	[91]	Reducing wasted development time via continuous testing
S43	[92]	Regression and Performance Testing of an e-learning Web Application - dotLRN
S44	[93]	Supporting Continuous Integration by Code-Churn Based Test Selection
S45	[94]	Techniques for Improving Regression Testing in Continuous Integration Development Environments
S46	[95]	Test case prioritization for continuous regression testing: An industrial case study
S47	[96]	Test Orchestration: A framework for Continuous Integration and Continuous Deployment
S48	[97]	Test prioritization with optimally balanced configuration coverage
S49	[98]	Test-Driven Development of Relational Databases
S50	[99]	Testing in parallel: A need for practical regression testing
S51	[100]	The State of Continuous Integration Testing @Google
S52	[101]	TITAN: Test Suite Optimization for Highly Configurable Software
S53	[102]	Understanding DevOps & bridging the gap from continuous integration to continuous delivery
S54	[103]	Using acceptance tests to validate accessibility requirements in RIA
S55	[104]	Verdict Machinery: On the Need to Automatically Make Sense of Test Results
S56	[105]	Virtual to the (Near) End: Using Virtual Platforms for Continuous Integration

5. **Fowler, M. & Foemmel, M. (2015).** Continuous integration. Online: https://www.thoughtworks.com / continuous-integration.

6. **Olsson, H. H., Alahyari, H., & Bosch, J., (2012).** Climbing the stairway to heaven - A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of

Appendix B. Quality assessment of the included studies

Table B.1. List of papers and quality scores

ID	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Total Score	Quality	Citations
S1	1	1	1	0	0.5	1	0.5	0	0	1	6	60%	11
S2	1	1	1	0	1	0.5	0.5	0	1	1	8	70%	0
S3	1	1	1	0	0	0	0	0	0	1	4	40%	5
S4	1	1	1	1	1	1	1	0.5	0.5	0.5	8.5	85%	0
S5	0	0	1	0	0	0	0	0	0	0.5	1.5	15%	0
S6	1	1	0.5	1	1	0	0	1	1	1	7.5	75%	37
S7	1	1	1	1	1	1	1	1	0	1	9	90%	31
S8	1	1	0	0.5	1	0.5	0	0.5	0	1	5.5	55%	1
S9	1	1	1	1	1	1	1	1	0	1	9	90%	8
S10	1	1	1	0	0	1	1	1	0	1	7	70%	3
S11	1	1	1	0	0	0	1	1	1	1	7	70%	73
S12	1	1	1	0.5	1	1	1	1	1	1	1	95%	4
S13	1	1	1	0	0	0	1	1	1	1	7	70%	1
S14	1	0.5	1	0	1	1	1	1	1	1	8.5	85%	6
S15	1	0.5	1	0	1	1	1	1	1	1	8.5	85%	13
S16	1	1	1	0	1	1	1	1	1	1	9	90%	8
S17	1	1	1	0	0.5	0.5	0.5	1	1	1	7.5	75%	58
S18	1	1	1	0.5	1	0.5	1	1	1	1	8.5	90%	18
S19	1	0.5	1	0	0	0	0	0	1	1	4.5	45%	13
S20	1	0.5	0.5	1	1	1	1	0.5	1	0.5	8	80%	3
S21	1	1	1	0	1	1	1	0.5	1	1	8.5	85%	0
S22	1	0.5	1	0	1	1	1	1	1	1	8.5	85%	2
S23	1	1	1	0	1	1	1	1	1	1	7	90%	0
S24	1	1	1	0	0	0	1	1	1	1	7	70%	0
S25	1	0.5	1	0	0.5	1	1	1	1	1	8	80%	2
S26	1	1	0	1	0	0	1	1	0.5	1	4	65%	3
S27	1	0	0	0	0	0	1	1	0	1	4	40%	0
S28	1	1	0	1	0	0	1	1	0.5	1	4	65%	0
S29	1	1	0.5	0	0.5	0.5	1	1	0.5	1	7	70%	2
S30	1	1	1	0	1	1	1	1	1	1	9	90%	2
S31	1	1	1	0.5	1	1	1	1	0	1	8.5	85%	1
S32	1	0.5	1	0.5	1	1	1	1	0	1	8	80%	11
S33	1	1	0.5	0	0	0.5	0.5	0.5	0.5	1	5.5	55%	1
S34	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	2
S35	1	1	1	0.5	0	1	1	1	0	1	7.5	75%	0
S36	1	1	1	0.5	1	1	1	1	0	1	8.5	85%	0
S37	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	0
S38	0	0.5	0.5	0	1	0	0	0.5	0.5	1	4	40%	0
S39	1	1	0	0	0	0	1	1	0	1	5	50%	3
S40	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	11
S41	1	1	1	1	1	1	1	1	1	1	10	100%	4
S42	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	128
S43	1	1	1	0.5	1	0.5	1	0.5	1	1	8.5	85%	5
S44	1	1	1	0.5	0.5	1	1	0.5	0	1	7.5	75%	6
S45	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	48
S46	1	1	1	0.5	1	0.5	1	1	1	1	9	90%	25
S47	1	1	1	0.5	0.5	0.5	1	0.5	1	1	8	80%	5
S48	1	1	1	1	1	1	1	1	1	1	10	100%	0
S49	1	1	1	0	0	0	0.5	1	0	1	5.5	55%	22
S50	1	1	1	0	0	0	0.5	1	1	1	6.5	65%	3
S51	1	0.5	1	0	1	1	1	1	1	1	8.5	85%	0
S52	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	0
S53	1	1	0.5	0	0.5	0	0	1	1	1	6	60%	18
S54	1	1	1	0.5	1	1	1	1	0.5	1	9	90%	15
S55	1	1	1	1	1	1	1	1	0	1	9	90%	1
S56	1	1	1	0	0.5	0	0.5	1	1	1	7	70%	1
Total												76%	

software. *38th EUROMICRO, Conference on Software Engineering and Advanced Applications* (SEAA), pp. 392–399. DOI:10.1109/SEAA.2012.54.

7. **Auerbach, A. (2015).** Part of the Pipeline: Why Continuous Testing Is Essential. TechWell Insights.

https://www.techwell.com / techwell-insights / 2015 /08/part-pipeline-why-continuous-testing-essential.

8. **Philipp-Edmonds, C. (2014).** The Relationship between Risk and Continuous Testing: An Interview with Wayne Ariola. https://www.stickyminds.com / interview/relationship-between-risk-and/continuous continuous-testing-interview-wayne-ariola.
9. **Ståhl, D. & Bosch, J. (2014).** Automated software integration flows in industry: A multiple-case study. *Companion Proceedings of the 36th International Conference on,* pp. 54–63. DOI: 10.1145/2591062. 2591186.
10. **Software Engineering (2014).** htpp://www.allengineeringschools.com/engineering-careers/computer-software-engineer/famous-software-engineers/, pp. 54–63.
11. **Chen, L. (2015).** Continuous Delivery: Huge benefits, but challenges too. *IEEE Software*, Vol. 32, No. 2, pp. 50–54. DOI: 10.1109/MS.2015.27.
12. **Debbiche, A., Dienér, M., & Svensson, R. B. (2014).** Challenges When Adopting Continuous Integration: A Case Study. *International Conference on Product-Focused Software Process Improvement,* Springer International Publishing, pp. 17–32. DOI: 10.1007/978-3-319-13835-0_2.
13. **Fitzgerald, B. & Stol, K. (2014).** Continuous software engineering and beyond: Trends and challenges. *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pp. 1–9. DOI:10.1145/2593812.2593813.
14. **Claps, G. G., Svensson, R. B., & Aurum, A. (2015).** On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software technology*, Vol. 57, pp. 21–31. DOI: 10.1016/j.infsof.2014.07.009.
15. **Fitzgerald, B. & Stol, K. J. (2017).** Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, Vol. 123, pp. 176–189. DOI: 10.1016/j.jss.2015.06.063.
16. **Mäntylä, M. V., Adams, B., Khomh, F., Engström, E., & Petersen, K. (2015).** On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, Vol. 20, No. 5, pp. 1384–1425. DOI: 1007/s10664-014-9338-4
17. **Neely, S. & Stolt, S. (2013).** Continuous delivery? easy! just change everything. *IEEE Agile Conference* (AGILE'13), pp. 121–128. DOI: 10.1109 /AGILE.2013.17.
18. **Chen, L. (2017).** Continuous Delivery: Overcoming adoption challenges. *Journal of Systems and Software,* Vol. 128, pp. 72–86. DOI: 10.1016/ j.jss.2017.02.013.
19. **Brooks, G. (2008).** Team pace keeping build times down. *IEEE Agile Conference (AGILE'08),* pp. 294–294. DOI: 10.1109/Agile.2008.41.
20. **Laukkanen, E., Lehtinen, T.O., Itkonen, J., Paasivaara, M., & Lassenius, C. (2016).** Bottom-up adoption of continuous delivery in a stage-gate managed software organization. *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 45. DOI: 10.1145/2961111.2962608.
21. **Cannizzo, F., Clutton, R., & Ramesh, R. (2008).** Pushing the boundaries of testing and continuous integration. *IEEE Agile Conference* (AGILE'08), pp. 501505. DOI: 10.1109/Agile.2008.31.
22. **Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V. P., Itkonen, J., Mäntylä, M. V., & Männistö, T. (2015).** The highways and country roads to continuous deployment. *IEEE Software*, Vol. 32, No. 2, pp. 64–72. DOI 10.1109/MS.2015.50.
23. **Debbiche, A. & Dienér, M. (2014).** *Assessing challenges of continuous integration in the context of software requirements breakdown: A case study.* Master Thesis, University of Gothenburg, Gothenburg, Sweden, pp. 1–65.
24. **Alégroth, E., Feldt, R., & Ryrholm, L. (2015).** Visual GUI testing in practice: challenges, problems and limitations. *Empirical Software Engineering*, Vol. 20, No. 3, pp. 694–744. DOI: 10.1007/s10664-013-9293-5.
25. **Borjesson, E. & Feldt, R. (2012).** Automated system testing using visual GUI testing tools: A comparative study in industry. *IEEE Fifth International Conference on Software Testing, Verification and Validation* (ICST), pp. 350–359. DOI 10.1109/ICST.2012.115.
26. **Pradhan, L. (2012).** *User Interface Test Automation and its Challenges in an Industrial Scenario.* Master Thesis. School of Innovation, Design and Engineering. Mälardalen University Sweden, Sweden.
27. **Suwala, P. (2015).** *Challenges with modern web testing.* Master Thesis. Institutionen för datavetenskap. Department of Computer and Information Science. Linköpings universitet, Linköping, Sweden. No. LIU-IDA/LITH-EX-A--15/013—SE.
28. **Huizinga, D. & Kolawa, D. (2017).** AJAX best practice: Continuous testing. *Automated Defect Prevention: Best Practices in Software Management,* Anonymous Wiley-IEEE Computer Society Press, pp. 391–393.

29. **Garg, N., Singla, S., & Jangra, S. (2016).** Challenges and Techniques for Testing of Big Data. *Procedia Computer Science*, Vol. 85, pp. 940–948. DOI: 10.1016/j.procs.2016.05.285.

30. **Muşlu, K., Brun, Y. & Meliou, A. (2013).** Data debugging with continuous testing. *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pp. 631–634. DOI: 10.1145/2491411. 2494580.

31. **Muccini, H., Di-Francesco, A., & Esposito, P. (2012).** Software testing of mobile applications: Challenges and future research directions. *Proceedings of the 7th International Workshop on Automation of Software Test,* IEEE Press, pp. 29–35.

32. **Feitelson, D. G., Frachtenberg, E. & Beck, K. L. (2013).** Development and deployment at Facebook. *IEEE Internet Computing*, Vol. 17, No. 4, pp. 8–17. DOI: 10.1109/MIC.2013.25.

33. **Schumacher, J. (2011).** Continuous Deployment at Atlassian. https://www.atlassian.com/blog/archives/ continuous _ deployment _ at _ atlassian.

34. **Rodríguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J. M., & Oivo, M. (2017).** Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, Vol. 123, pp. 263–291. DOI:10.1016/j.jss.2015. 12.015.

35. **Ståhl, D. & Bosch, J. (2013).** Experienced benefits of continuous integration in industry software product development: A case study. *Proceedings of the 12th IASTED international conference on software engineering*, pp. 736–743. DOI: 10.2316/P. 2013.796-012.

36. **Ståhl, D. & Bosch, J. (2014).** Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, Vol. 87, pp. 48–59. DOI: 10.1016/j.jss.2013.08.032.

37. **Eck, A., Uebernickel, F., & Brenner, W. (2014).** Fit for continuous integration: How organizations assimilate an agile practice. *Proceedings of the 20th Americas Conference on Information Systems*, pp. 1–11.

38. **Cruzes, D. S. & Dybå, T. (2011).** Research synthesis in software engineering: A tertiary study*.* *Information and Software Technology*, Vol. 53, No. 5, pp. 440–455.

39. **Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., & Linkman, S. (2009).** Systematic literature reviews in software engineering – a systematic literature review. *Information and software technology*, Vol. 51, No. 1, pp. 7–15. DOI: 10.1016/j.infsof.2008.09.009.

40. **Dybå, T. & Dingsøyr, T. (2008).** Strength of evidence in systematic reviews in software engineering. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pp. 178–187. DOI: 10.1145/1414004.1414034.

41. **Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., & Khalil, M. (2007).** Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, Vol. 80, No. 4, pp. 571–583. DOI: 10.1016/j.jss.2006.07.009.

42. **Kitchenham, B. & Charters, S. (2007).** *Guidelines for performing systematic literature reviews in software engineering.* Keele University and Durham University. Report No. EBSE 2007-001.

43. **Do Carmo-Machado, I., Mcgregor, J. D., Cavalcanti, Y. C. & De Almeida, E. S. (2014).** On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, Vol. 56, No. 10, pp. 1183–1199. DOI: 10.1016/j.infsof.2014.04.002.

44. **Vilela, J., Castro, J., Martins, L. E. G., & Gorschek, T. (2017).** Integration between requirements engineering and safety analysis: A systematic literature review. *Journal of Systems and Software*, Vol. 125, pp. 68–92. DOI:10.1016/j.jss. 2016.11.031.

45. **Dermeval, D., Vilela, J., Bittencourt, I. I., Castro, J., Isotani, S., Brito, P., & Silva, A. (2016).** Applications of ontologies in requirements engineering: a systematic review of the literature. *Requirements Engineering*, Vol. 21, No. 4, pp. 405–437. DOI: 10.1007/s00766-015-0222-6.

46. **Easterbrook, S., Singer, J., Storey, M. A. & Damian, D. (2008).** Selecting empirical methods for software engineering research. *Guide to advanced empirical software engineering*, pp. 285–311. DOI: 10.1007/978-1-84800-044-5_11.

47. **Wieringa, R., Maiden, N., Mead, N., & Rolland, C. (2006).** Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements Engineering*, Vol. 11, No. 1, pp. 102–107. DOI: 10.1007/s00766-005-0021-6.

48. **Smith, E. G. (2000).** Continuous testing. *Proceedings of the 17th International Conference on Testing Computer Software*.

49. **Duvall, P. M., Matyas, S., & Glover, A. (2007).** *Continuous integration: improving software quality and reducing risk.* Pearson Education.

50. **Graham, D., Van Veenendaal, E., & Evans, I. (2008).** *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA.
51. **Bindal, P. & Gupta, S. (2012).** Test Automation Selenium WebDriver using TestNG. *Journal of Engineering Computers & Applied Sciences* (JECAS), Vol. 3, No. 9, pp. 18–40.
52. **Barquet, A. L., Tchernykh, A., & Yahyapour, R. (2013).** Performance Evaluation of Infrastructure as Service Clouds with SLA Constraints. *Computacion y Sistemas*, Vol. 17, No. 3, pp. 401–411.
53. **Blaze Meter. (2015).** BlazeMeter Introduces Continuous-Testing-as-a-Service to Marketplace. https://www.blazemeter.com/blazemeter-newsnews/blazemeter-introduces-continuous-testing-service-marketplace.
54. **Serna, M. E., Martínez, M. R., & Tamayo, O. P. A. (2017).** A Model for Determining the Maturity of Automation of Software Testing as a Research and Development Area. *Computacion y Sistemas*, Vol. 21, No. 2, pp. 337–352.
55. **Alshraideh, M. (2008).** A complete automation of unit testing for javascript programs. *Journal of Computer Science*, Vol. 4, No. 12.
56. **Gomede, E., Da Silva, R. T. & De Barros, R. M. (2015).** A practical approach to software continuous delivery focused on application lifecycle management. *Proceedings of the 27th Software Engineering and Knowledge Engineering (SEKE)*, pp. 320–325. DOI: 10.18293/SEKE2105-126.
57. **Li, N., Escalona, A., Guo, Y., & Offutt, J. (2015).** A scalable big data test framework. *Proceedings of the IEEE 8th International Conference on Software Testing,* Verification and Validation (ICST), pp. 1–2. DOI: 10.1109/ICST.2015.7102619.
58. **Zhang, P., Zhou, X., Gao, J., & Tao, C. (2017).** A survey on quality assurance techniques for big data applications. *Proceedings of the 3rd International Conference on Big Data Computing Service and Applications*. DOI:10.1109/BigDataService.2017.42
59. **Tsai, W. T., Zhong, P., Balasooriya, J., Chen, Y., Bai, X., & Elston, J. (2011).** An approach for service composition and testing for cloud computing. *Proceedings of the 10th International Symposium on Autonomous Decentralized Systems (ISADS)*, pp. 631–636. DOI: 10.1109/ISADS.2011.90.
60. **Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014).** An empirical analysis of flaky tests. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 643–653. DOI: 10.1145/2635868.2635920.
61. **Smith, E.G. (2001).** Automated Test Results Processing. *Proceedings of the STAREAST`01 Conference*, pp. 1–13.
62. **Gmeiner, J., Ramler, R., & Haslinger, J. (2015).** Automated testing in the continuous delivery pipeline: A case study of an online company. *Proceedings of the IEEE 8th International Conference on Software Testing,* Verification and Validation Workshops (ICSTW), pp. 1–6. DOI: 10.1109/ICSTW.2015.7107423.
63. **Erder, M. & Pureur, P. (2015).** Continuous architecture and continuous delivery. *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric*, Morgan Kaufmann (Elsevier), pp. 103–129.
64. **Vassallo, C., Zampetti, F., Romano, D., Beller, M., Panichella, A., Di Penta, M., & Zaidman, A. (2016).** Continuous delivery practices in a large financial organization. *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 519–528. DOI: 10.1109/ICSME.2016.72.
65. **Rossi, C., Shibley, E., Su, S., Beck, K., Savor, T., & Stumm, M. (2016).** Continuous deployment of mobile software at Facebook (showcase). *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 12–23. DOI: 10.1145/2950290.2994157.
66. **Moe, N. B., Cruzes, D., Dybå, T., & Mikkelsen, E. (2015).** Continuous software testing in a globally distributed project. *Proceedings of the IEEE 10th International Conference on Global Software Engineering (ICGSE)*, pp. 130–134. DOI: 10.1109/ICGSE.2015.24.
67. **Campos, J., Fraser, G., Arcuri, A., & Abreu, R. (2015).** Continuous Test Generation on Guava. *International Symposium on Search Based Software Engineering*, Springer, pp. 228–234. DOI: 10.1007/978-3-319-22183-0_16.
68. **Madeyski, L. & Kawalerowicz, M. (2013).** Continuous Test-Driven Development - A Novel Agile Software Development Practice and Supporting Tool. *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*, pp. 260–267.
69. **Saff, D. & Ernst, M. D. (2005).** Continuous testing in eclipse. *Proceedings of the 27th International Conference on Software Engineering*, pp. 668–669.
70. **Dolstra, E., Vliegendhart, R., & Pouwelse, J. (2013).** Crowdsourcing GUI tests*. Proceedings of the IEEE 6th International Conference on Software Testing,* Verification and Validation (ICST), pp. 2159–4848. DOI: 10.1109/ICST.2013.44.

71. **Zachow, M. (2016).** Designing an Android Continuous Delivery pipeline. *Software Engineering (Workshops)*, pp. 160–163.

72. **Eloussi, L. (2015).** *Determining flaky tests from test failures.* PhD Thesis. University of Illinois at Urbana-Champaign, Urbana, Illinois.

73. **Gotimer, G. & Stiehm, T. (2016).** DevOps Advantages for Testing: Increasing Quality through Continuous Delivery. *CrossTalk Magazine*, May/Jun Ed., pp. 13–18.

74. **Marijan, D. & Liaaen, M. (2016).** Effect of time window on the performance of continuous regression testing. *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 568–571. DOI: 1109/ ICSME.2016.77.

75. **Kim, J., Jeong, H., & Lee, E. (2017).** Failure history data-based test case prioritization for effective regression test. *Proceedings of the Symposium on Applied Computing*, pp. 1409–1415. DOI:10.1145/ 3019612.3019831

76. **Eyl, M., Reichmann, C., & Müller-Glaser, K. (2016).** Fast feedback from automated tests executed with the product build. *Proceedings of the International Conference on Software Quality*, Springer, Cham, pp. 199–210. DOI: 10.1007/978-3-319-27033-3_14.

77. **Tilley, S. & Floss, B. (2014).** Hard Problems in Software Testing: Solutions Using Testing as a Service (TaaS). *Synthesis Lectures on Software Engineering*, Vol. 2, No. 1, pp. 1–103.

78. **Sinisalo, M. T. (2016).** *Improving web user interface test automation in continuous integration.* Master Thesis. Tampere University of Technology. Finland.

79. **Alégroth, E. & Feldt, R. (2014).** Industrial Application of Visual GUI Testing: Lessons Learned. *Continuous Software Engineering*, Springer International Publishing, pp. 127–140, pp. DOI: 10.1007/978-3-319-11283-1_11.

80. **Burgin, M. & Debnath, N. (2010).** Intelligent testing systems. *IEEE World Automation Congress* (WAC), pp. 1-6.

81. **Penix, J. (2012).** Large-scale test automation in the cloud. *Invited Industrial Talk of IEEE 34th International Conference on Software Engineering* (ICSE), pp. 1122–1122.

82. **Busjaeger, B. & Xie, T. (2016).** Learning for test prioritization: an industrial case study. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 975–980. DOI: 10.1145/2950290. 2983954.

83. **Savchenko, D. I., Radchenko, G. I., & Taipale, O. (2015).** Microservices validation: Mjolnirr platform case study. *Proceedings of the IEEE 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 248–253.

84. **Brajnik, G., Baruzzo, A., & Fabbro, S. (2015).** Model-based continuous integration testing of responsiveness of web applications. *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–2. DOI: 10.1109/ICST.2015.7102626.

85. **Marijan, D. (2015).** Multi-perspective regression test prioritization for time-constrained environments. *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 157–162. DOI: 10.1109/QRS.2015.31.

86. **Alotaibi, A. A. & Qureshi, R. J. (2017).** Novel Framework for Automation Testing of Mobile Applications using Appium. *International Journal of Modern Education and Computer Science*, Vol. 9, No. 2, pp. 34–40. DOI: 10.5815/ijmecs.2017.02.04.

87. **Gambi, A., Gorla, A., & Zeller, A. (2017).** O! Snap: Cost-Efficient Testing in the Cloud. *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 454–459. DOI: 10.1109/ICST.2017.51.

88. **Alégroth, E. & Feldt, R. (2017).** On the long-term use of visual GUI testing in industrial practice: a case study. *Empirical Software Engineering*, Vol. 22, No. 6, pp. 2937–2971.

89. **Ramakrishnan, A. & Manjula, R. (2016).** Perceptual Difference for Safer Continuous Delivery. *International Research Journal of Engineering and Technology (IRJET)*, Vol. 3, No. 11, pp. 793–798.

90. **Erder, M. & Pureur, P. (2015).** Principles of continuous architecture. *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric*, Morgan Kaufmann (Elsevier), pp. 21–37.

91. **Hemmati, H., Fang, Z., & Mantyla, M. V. (2015).** Prioritizing manual test cases in traditional and rapid release environments. *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, DOI: 10.1109/ICST.2015.7102602.

92. **Saff, D. & Ernst, M.D. (2003).** Reducing wasted development time via continuous testing. *Proceedings of the IEEE 14th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 281–292.

93. **Cavalli, A., Maag, S., & Morales, G. (2007).** Regression and Performance Testing of an e-learning Web application: dotLRN. *Proceedings of*

the 3rd IEEE International Conference on Signal-Image Technologies and Internet-Based System (SITIS), pp. 369–376. DOI:10.1109/SITIS.2007.129.

94. **Knauss, E., Staron, M., Meding, W., Söder, O., Nilsson, A., & Castell, M. (2015).** Supporting continuous integration by code-churn based test selection. *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, IEEE Press, pp. 19–25.
95. **Elbaum, S., Rothermel, G., & Penix, J. (2014).** Techniques for improving regression testing in continuous integration development environments. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 235–245. DOI: 10.1145/2635868.2635910.
96. **Marijan, D., Gotlieb, A., & Sen, S. (2013).** Test case prioritization for continuous regression testing: An industrial case study. *Proceedings of the IEEE 29th International Conference on Software Maintenance (ICSM)*, pp. 540–543, DOI: 10.1109/ICSM.2013.91.
97. **Rathod, N. & Surve, A. (2015).** Test orchestration a framework for Continuous Integration and Continuous deployment. *Proceedings of the IEEE International Conference on Pervasive Computing (ICPC)*, pp. 1–5, DOI:10.1109/PERVASIVE.2015.7087120.
98. **Marijan, D. & Liaaen, M. (2017).** Test Prioritization with Optimally Balanced Configuration Coverage. *Proceedings of the IEEE 18th International Symposium on High Assurance Systems Engineering (HASE*), pp. 100–103. DOI: 10.1109/HASE.2017.26.
99. **Ambler, S. W. (2007).** Test-driven development of relational databases. *IEEE Software*, Vol. 24, No. 3, pp. 37–43. DOI: 10.1109/MS.2007.91.
100. **Zhang, Z., Tong, Z., & Gao, X. (2010).** Testing in Parallel - A Need for Practical Regression Testing. *Proceedings of the 5th International Conference on Software Engineering and Systems Development, Software Systems and Applications,* Foundational and Trigger Technologies (ICSOFT), pp. 344–348.
101. **Micco, J. (2017).** The state of continuous integration testing @ google. *11th International Conference on Sensing Technology (ICST)*.
102. **Marijan, D., Liaaen, M., Gotlieb, A., Sen, S., & Ieva, C. (2017).** TITAN: Test Suite Optimization for Highly Configurable Software. *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 524–531. DOI: 10.1109/ICST.2017.60.
103. **Virmani, M. (2015).** Understanding DevOps & bridging the gap from continuous integration to continuous delivery. *Proceedings of the 2015 5th International Conference on Innovative Computing Technology (INTECH)*, pp. 78–82. DOI: 10.1109/INTECH.2015.7173368.
104. **Watanabe, W. M., Fortes, R. P., & Dias, A. L. (2012).** Using acceptance tests to validate accessibility requirements in RIA. *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*, Art. 15 DOI: 10.1145/220 07016.2207022.
105. **Fagerström, M., Ismail, E. E., Liebel, G., Guliani, R., Larsson, F., Nordling, K., Knauss, E., & Pelliccione, P. (2016).** Verdict machinery: on the need to automatically make sense of test results. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 225–234. DOI: 10.1145/2931037.2931064.
106. **Engblom, J. (2015).** Virtual to the (near) end: Using virtual platforms for continuous integration. *Proceedings of the 52nd Annual Design Automation Conference*, Art. No. 200. DOI: 10.1145/2744769.2747948.

Article received on 30/08/2017; accepted on 20/10/2017.
Corresponding author is Maximiliano A. Mascheroni.