- ORIGINAL ARTICLE -

Identifying Key Success Factors in Stopping Flaky Tests in Automated REST Service Testing

Identificación de Factores Clave de Éxito para Evitar las Pruebas Automatizadas no Determinísticas en Servicios REST

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

[1] *Departamento de Informática, Universidad Nacional del Nordeste, Corrientes, Argentina*
{mascheroni, eirrazabal}@exa.unne.edu.ar
[2] *Facultad de Informática, Universidad Nacional de La Plata*

Abstract

A flaky test is a test which could fail or pass for the same version of a certain software code. In continuous software development environments, flaky tests represent a problem. It is difficult to get an effective and reliable testing pipeline with a set of flaky tests. Also, according to many practitioners, despite the persistence of flaky tests in software development, they have not drawn much attention from the research community. In this paper, we describe how a company faced this issue, and implemented solutions to solve flaky tests for REST web services. The paper concludes proposing a set of key success factors for stopping flaky tests in this type of testing.

Keywords: Flaky tests, continuous integration, continuous deployment, continuous delivery, web service testing.

Resumen

Una prueba no determinística es una prueba que podría fallar o ser exitosa con la misma versión de un determinado código de software. En entornos de desarrollo de software continuo, las pruebas no determinísticas representan un problema. Es difícil obtener un proceso de pruebas efectivo y confiable con pruebas no determinísticas. Además, de acuerdo con muchos profesionales, a pesar de la persistencia de este tipo de pruebas, las mismas no han llamado mucho la atención de la comunidad científica. En este trabajo, describimos cómo una empresa se ha enfrentado este problema e implementado soluciones para resolver pruebas no determinísticas en servicios REST. Al final, se proponen un conjunto de factores clave de éxito para evitar este problema en pruebas de servicios.

Palabras claves: Pruebas no determinísticas, integración continua, despliegue continuo, entrega continua, pruebas de servicios web.

1. Introduction

An important characteristic of an automated test is its determinism. This means that a test should always produce the same result when the system under test (SUT) does not change. A test that fails randomly is not reliable and it is commonly called as "flaky test". Automated flaky tests slow down progress, cannot be trusted, hide real bugs and are not cost effective.

"Flaky tests" is not a new term. Some practitioners like Martin Fowler [1] have referred to flaky tests as non-deterministic tests. According to different authors [1,2,3,4,5,6], flaky tests are tests that have non-deterministic outcomes with respect to a given software version. During the last years, flaky tests have been a problem for several companies.

Google, for example, has a continual rate of about 1.5% of all test runs reporting a "flaky" result [7]. They have a Continuous Integration (CI) pipeline which identifies the moment when a passing test becomes a failure, so that they can investigate the checked-in code that caused that transition. Google statistics show that in practice 84% of the transitions from pass to fail involve a flaky test [7].

In the same way, many authors have reported testing problems related to flaky tests [3,8,9,10,11,12,13,14,15]. Nowadays, organizations invest a lot of effort to stop flaky tests. Thus, different proposals can be found in the literature with their corresponding pros and cons. Similarly, in this paper we describe how a company was able to

Citation: M.A. Mascheroni and E. Irrazábal. "*Identifying key success factors in stopping flaky tests in automated REST service testing",* Journal of Computer Science & Technology, vol. 18, no. 2, pp. 143-152, 2018.

DOI: 10.24215/16666038.18.e16

Received: February 10, 2018 **Revised:** August 30, 2018 **Accepted:** September 4, 2018.

stop flaky tests in automated REST service testing by applying a set of practices. In the end, we propose a list of key success factors that are derived from applying these practices to stop flaky tests.

Apart from this introductory section, common causes of flaky tests and existing proposals on how to deal with them are described in section 2. Section 3 describes the issues that the company had and the implemented solutions. The proposed key success factors are presented in Section 4, with a brief discussion of the applied steps. Section 5 describes threats to validity. Finally, we present our conclusions and ideas for future work in Section 6.

2. Background in flaky tests

2.1. Common causes

In [2], an empirical analysis of flaky tests is presented. The authors of that article classify the most common root causes of flaky tests and they describe strategies that developers use to fix flaky tests. Thus, the most common root causes of flaky tests are [2]:

1) **Asynchronous wait**: it happens when a test script makes an asynchronous call and does not wait for the results to be available before using them.

2) **Concurrency**: flaky tests that are caused by different threads interacting in a non-desirable manner (data races, deadlocks, atomicity violations, etc.)

3) **Test order dependency**: according to the best practices of automated testing, all tests in a test suite should be independent of one another and the order in which they are run should not affect their outcomes. However, in practice, it is not the case: flaky tests can be produced by test outcomes that depend on the order in which the tests are run.

4) **Resource leak**: test failures may also occur whenever the application does not properly manage one or more of its resources such as database connections, memory allocations, etc.

5) **Network**: the network is a resource that is difficult to control, so tests whose execution depends on it can be flaky.

6) **Time**: relying on the system time may introduce flakiness. For example, a test may fail when the midnight changes in the UTC time zone.

7) **IO**: I/O operations may also cause flaky tests similarly to resource leaks.

8) **Randomness**: the use of random numbers generator (without accounting for all the possible values that may be generated) may also cause some tests to be flaky.

9) **Floating point operations**: tests that performs these operations may become flaky.

10) **Unordered collections**: when the tests iterate over unordered collections (e.g., lists, maps, sets), and they assume that the elements are returned in a particular order, then the test outcome can become non-deterministic because different executions may have a different order.

According to [2], there are more causes of flaky tests that depends on each individual project. A project where a big data application is being developed may have a certain type of flaky tests, different from a project for a microservices based application. Also, we have found more causes like:

11) **Servers problems**: when the server is down or unstable, automated tests may fail [16].

12) **Having user interface (UI) testing**: the UI is the part of an application that changes most frequently, and it can drive to flaky tests [12,17].

2.2. Common fixes

There are many workarounds for flaky tests that can be found in the literature. We will list the most common fixes for the aforementioned root causes:

1) **Fixes for asynchronous wait failures**: using *waitFor* calls [2,18,19]; using *sleep* calls [2], reordering code [2].

2) **Fixes for concurrency failures**: adding locks [2,18]; making code deterministic [2,13,20]; changing concurrency guard conditions [2]; changing assertions [2,20].

3) **Fixes for test order dependency failures**: setting up/cleaning up states [2,17]; removing dependency [2,18]; merging tests [2].

4) **Fixes for resource leak failures**: managing relevant resources through resource pools [1].

5) **Fixes for network failures**: using mocks [2,21], using *waitFor* calls [2]; adding connection retries [18].

6) **Fixes for time failures**: avoiding the use of platform dependent values (e.g. time) [2].

7) **Fixes for I/O failures**: closing any opened resource [2,18]; using proper synchronization between different threads sharing the same resource [2].

8) **Fixes for randomness failures**: controlling the seed of the random generator and the boundary values that the random number can return [2]; modifying assertions [18].

9) **Fixes for floating point operations failures**: making assertions more flexible in to accepting a range of values instead of just one [18].

10) **Fixes for unordered collections**: writing tests that do not assume any specific ordering on collections [2]; using pointers [18].

11) **Fixes for server problems**: using *waitFor* calls [18].

12) **Fixes for UI false positive failures**: using *waitFor* calls [18, 19]; adopting model-based UI testing [22]; adopting Visual GUI testing (VGT) [23]; adding image comparison [24]; including crowdsourced GUI testing [25].

A summary of the reported common causes for flaky tests and their common fixes are shown in Table 1.

Table 1 Common causes and fixes for flaky tests.

Common Causes	**Common Fixes**
Asynchronous wait	*waitFor* and *sleep* calls, code reordering.
Concurrency	Locks, deterministic code, concurrency guard conditions and assertions improvement.
Test order dependency	States cleaning up, dependency removal.
Resource leak	Managing relevant resources through resource pools.
Network	Mocks, Retries, *waitFor* calls.
Time	Avoid time as platform dependent values
I/O operations	Closing resources after using them, adding synchronization.
Randomness	Managing the seed and boundaries of random values generators.
Floating point operations	Flexible Assertions.
Unordered collections	Dependency removal on collections that need to be ordered.
UI false positive failures	*waitFor* calls, model-based testing, VGT, image comparison, crowdsourced UI testing.
Server problems	*waitFor* calls.

Apart from these workarounds, other solutions for general flaky test issues can be found.

In Table 2 solutions for general flaky test issues are presented with their pros and cons. However, we have found more flaky tests root causes in automated REST service testing. In the next section we will describe them and how the company has mitigated them.

Table 2 Pros and cons of solutions for general automated flaky tests.

Solution	Ref.	Pros	Cons
Test prioritization and test selection	[26, 27, 28]	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.	[27]	1) Flaky tests are easier to identify and ignore.	1) Flaky tests still exist.
Test the automated test scripts for flakiness.	[2,29]	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 2) A lot of execution time.
Re-running tests.	[17,30]	1) It reduces the number of failures due to flaky tests.	1) Longer execution times. 2) Flaky tests still exist.
Postpone tests re-runs till the end of the execution.	[17]	1) It reduces the number of failures due to flaky tests. 2) It is possible to determine the cause of flakiness.	1) Longer execution times. 2) Flaky tests still exist.

3. Facing flaky tests in a REST service architecture

In this section we describe the project background, the problems with flaky tests and the solutions that have been applied in order to face them.

3.1. The project

The company where the project is being developed, is a digital marketing agency and an Interactive Investment Management (IIM) firm which specializes in digital media and analytics for different clients worldwide. Its services include Paid Search, SEO, Affiliate Marketing, Web Analytics, Link Building, Display, Email, Mobile, Affiliate and Social Media. It uses a cloud-based, or software as a service model. It operates offices in the United States, Canada, Europe and Latin America.

The project consists in a backend architecture, which connect the frontend of the application with several big data technologies such as Hadoop, HBase, MongoDB, Elasticsearch and Spark Streaming. This connection is made by using RESTful web services.

Currently, there are 4 teams working on that project. Each team is composed by 6 Java developers, 1 manual tester and 1 test developer[1]. Thus, the project is supported by 24 developers, 4 manual testers and 4 test developers.

3.1.1. The testing process

While the feature is being developed, manual testers write test scenarios using ubiquitous language. At the same time, test developers prepare the necessary components (drivers, dependencies, etc.) before developing the test scripts for that feature.

When the feature is completed, it is deployed to an environment for developers (dev environment). There, developers verify that the feature is working with the other components of the application. Then, when all the features of the current sprint are completed, they are deployed to a QA environment, where the manual testers verify whether they satisfy

[1] The company uses this term to refer to a person who only develops test scripts.

the acceptance criteria and they also run the regression tests. At the same time, the test developers start to develop the test scripts for the features. It is very important to highlight that for the development of test scripts, it was used a QA framework. In that framework, test developers added classes that interacted with the different databases (Mongo, Hbase, etc.) and Elasticsearch. In the same way, they created classes that represented the requests and the responses for the different REST services of the application. Finally, the test classes were composed by methods which interacted with the mentioned classes and made assertions to verify expected conditions. The architecture of the QA framework can be seen in Fig. 1.

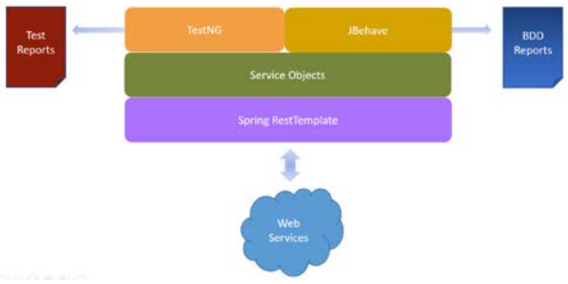


Fig. 1 Architecture of the QA Framework.

3.1.2. Flaky tests in the project

Execution of regular and regression tests are performed using a CI server. The CI server contains plugins which provide developers with metrics such as tests duration, number of failed tests, number of passed tests, number of skipped tests, build status trend, and similar metrics.

However, regression testing was not performed immediately after developers checked-in changes into the repository, because the test results were not reliable. The reports generated by the CI server helped the team to analyze root causes of failures, and then identify 4 reasons (see Fig. 2):

Reason 1 (R1): Failures produced by unavailable or inconsistent test data. For example, given a database which contains city names associated with country codes, a list of city names can be retrieved using a web service. The request to the server must contain the country code as a parameter. If the requested country code does not exist in the database or it is incorrect, then the server will not return the expected number of city names and the test at verifying this scenario will fail.

Reason 2 (R2): Internal server errors (HTTP status code 500) not related to a server which is down or unavailable. An internal server error indicates that there was a problem with the server. However, sometimes the server returned this error but it was produced by other completely different cause (for example, a bad request).

Reason 3 (R3): Failures produced by REST API requests/responses which have changed because of business requirements. The requests and the responses of the web services are represented using POJOs in the QA framework. When these requests and responses change because of business requirements, the POJOs need to be refactored. However, this refactoring is performed once the code is deployed in the QA environment, causing tests to fail in previous stages.

Reason 4 (R4): Real Failures. Bugs introduced by developers.

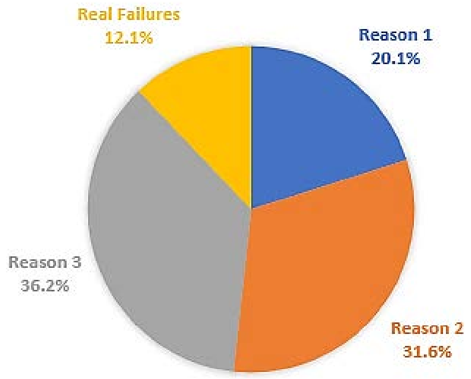


Fig. 2 Failures reasons in the project by percentage.

As it can be seen, R1, R2 and R3 are causes of flaky tests and they represent almost a 90% of the failures. In Fig. 3 it is also presented the build status trend report generated by the CI server.

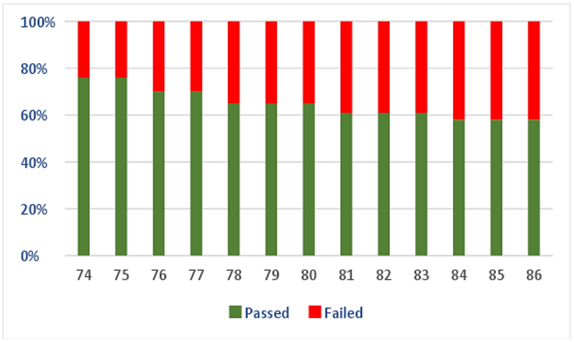


Fig. 3 Build status trend report of the builds #74 to #86.

3.2. The solution

The solution approach consisted of a set of steps which were applied gradually. Some of the steps were taken from unit tests principles [31]. Others were taken from continuous software development practices [32,33,34]. Finally, the rest of them were taken from a set of papers whose main focus is software testing [35,36,37]. We will detail each solution for every failure reason.

3.2.1. R1: Failures produced by unavailable or inconsistent test data

S1.1: Preconditions verification for test data

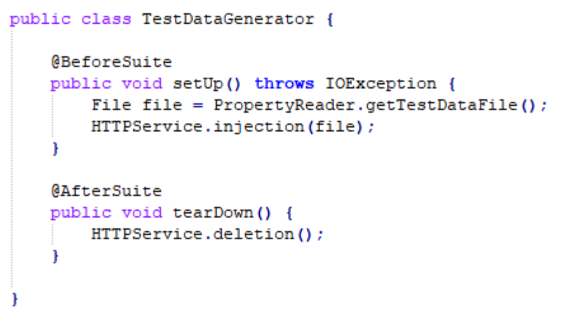
The presence of test data is verified before running the test cases which use that data. Also, if some scenarios have to be ran before others in order to generate specific data, they are verified in precondition steps. The result is a number of skipped tests, instead of failures. However, not having a pass/fail result decreases the coverage of the testing stage.

S1.2: Automated test content injection/deletion

As S1.1 avoids flaky tests produced by unavailable or inconsistent test data, but it does not generate a pass/fail result, an automated test data generator was developed. Developers in collaboration with test engineers, made two endpoints in the application: one for injecting test data and the other one for deleting it.

Thus, a file containing valid test data is prepared and then injected automatically as part of a *beforeSuite* method by using the injection endpoint. Finally, after the test cases run, the deletion endpoint is called as part of an *afterSuite* method and the test data is deleted from the databases and the search engine.

A screenshot of the TestDataGenerator class is shown in Fig. 4.



```
public class TestDataGenerator {

    @BeforeSuite
    public void setUp() throws IOException {
        File file = PropertyReader.getTestDataFile();
        HTTPService.injection(file);
    }

    @AfterSuite
    public void tearDown() {
        HTTPService.deletion();
    }

}
```

Fig. 4 TestDataGenerator class.

After applying S1.1 and S1.2 flaky tests produced by unavailable or corrupted test data were fixed.

3.2.2. R2: False Internal server errors

S2.1: Negative testing

It includes the verification of negative scenarios which produce errors like bad requests, not found resources, unauthorized access, etc. Thus, it’s possible to distinguish between real failures produced by the server (internal server errors) and errors produced by negative scenarios. In order to do this, developers have to follow HTTP standards and fix all the incorrect errors in the responses. Then, if a non-expected error appears, it is considered as a defect.

S2.2: Health suite

Negative scenarios fix the false internal server errors, by adding a verification of real expected errors. However, in order to avoid failures caused by problems with the servers, a health suite has been added. The health suite is ran before any test script, even before the injection of the test data. It verifies that the environment is up and running. The scope of this verification includes the server, databases, streams and the search engine.

3.2.3. R3: Failures produced by REST API requests/responses

S3.1: Integration of the QA framework as another module of the REST application project.

The integration of the QA framework with the other project modules, allows test developers to reuse the same classes used by developers, to be mapped with the requests and responses.

Thus, if changes in the requests or responses of the service are made, they will not have an effect in the requests or responses classes (POJOs) handled by the test scripts. Also, the test code can be refactored at the same time the code base is modified.

4. Discussion

After applying the 5 mentioned solutions, the different teams working on the project were able to fix the flaky tests. The results of the progress can be seen in Fig. 5.

The first step was the analysis of the root causes of the failures which was described in section 3.1.2. Then, the test developers started to share the problems across the developers of the different teams by using presentation slides.

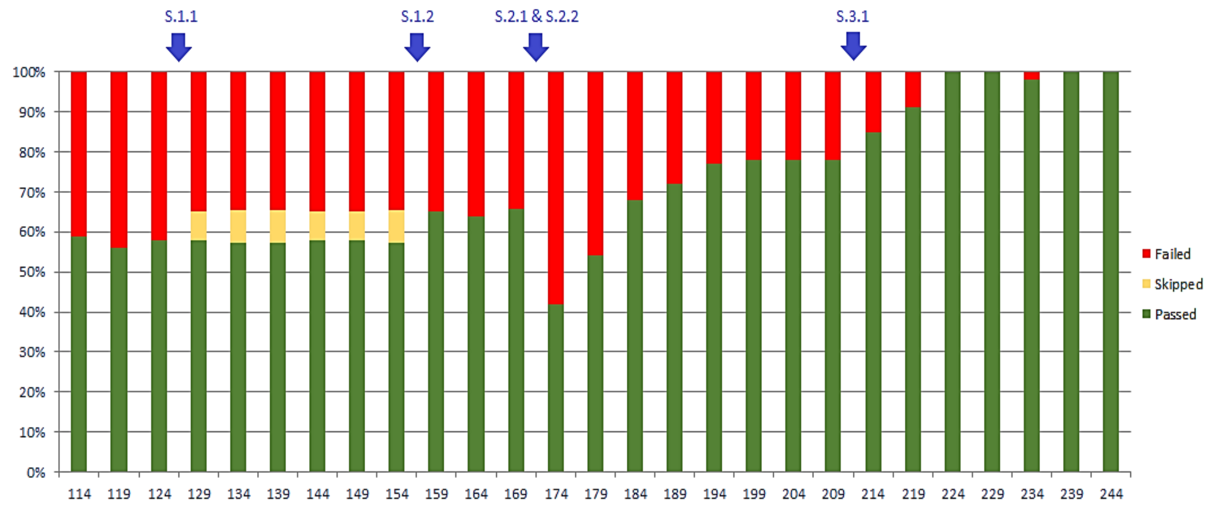


Fig. 5 Build status trend report showing the last 100 results and the time where solutions were applied.

S.1.1 was implemented first in build #126, skipping the number of flaky tests caused by R1. After one week of analyzing the best option to fix those skipped tests, and three weeks of developing the injection and deletion endpoints for test data, S.1.2 was implemented in build #157. The percentage of passed test scripts increased from 57% to 65% approximately. Two weeks later, negative scenarios (S.2.1) were added, at the same time with the health suite (S.2.2). The negative scenarios increased the number of test cases and the coverage of the REST service test stage. However, they decreased the number of passed test scripts. These failures represented just incorrect error responses, so developers started to fix them gradually. After two weeks, the project increased the percentage of passed tests to 78%. Finally, in build #212, S.3.1 was implemented. After a couple of days of refactoring, the test build was completely stabilized. New failures after that, represented real defects. Thereby, the test stage gained reliability by eliminating false positives.

The adoption of these solutions took almost 4 months. However, the most important step was bridging the gap between test engineers and developers. Without that collaboration, most of these solutions could not have been implemented. Thus, we propose a list of key success factors to stop flaky tests in automated REST service testing.

Also, the company have improved the test processes by adding continuous deployment as follows:

1) Developers check-in new code into the mainline trunk.

2) The CI server builds the code and runs the unit tests.

3) If step 2 passes, then the CI server deploys the changes automatically into the developer’s environment.

4) The CI server runs the automated REST service test scripts.

5) If step 4 passes, then the code is deployed automatically into the QA environment.

Thus, the QA environment is ready for a manual testing stage where manual testers only need to verify that the newly developed features satisfy the acceptance criteria.

The implementation of the solutions fixed the problems related to unreliable tests, but the use of continuous deployment and the execution of the test scripts earlier in the pipeline, helped the teams to improve the speed of the release process.

4.1.1. Key success factors to stop flaky tests in automated REST service testing

Based on the experience we acquired by applying the solutions to different problems and attaining the results, we propose the following 10 key success factors. We consider that they can be applied to automated REST service testing in order to avoid flaky tests.

1) Bridge the gap between DEV and QA. If your team has different roles for testing and developing tasks, then improve the collaboration between test engineers and developers.

2) Don’t create a testing framework, but add a testing module to the code base.

3) Reuse as much code base as possible: entities, database and search engine connectors, configuration files, scripts, etc.

4) Create a mechanism to create test data before your test scripts run.

5) Create a mechanism to delete the created test data after your test scripts run.

6) Verify that the test data required for the test scripts is available before running them. If the test data is not available, then skip the tests that depend on it.

7) Create a health suite in order to verify the stability of your environment and run it before any other suite (smoke, regression, etc.). If the health suite fails, then skip all of your test scripts.

8) Add negative scenarios.

Additional good practices for continuous software development environments:

9) Add continuous deployment.

10) Run your automated REST service testing suites in the corresponding environment, immediately after the changes in the code base are introduced.

5. Threats to validity

The validity of this proposal is threatened by the following issues:

- The scope of the proposed key success factors is the testing of RESTful web services. Even though it might be applied to the testing of other web services architectures like SOAP or WSDL, they have not been contemplated in the experiment.

- The injection/deletion of test data is not always possible. If it is not possible, then the key success factors cannot be applied.

- Sometimes, the servers do not have an API to verify their status. Verifying the status of a server is very important for the creation of the health suite.

- The implementation of the mentioned solutions took almost 4 months, but that time is directly related to the amount of people working on the project and the size of the project.

- The experiment was performed in a project where developers and testers were able to work together. As it was mentioned above, collaboration between them is very important. If the collaboration between developers, test engineers and other similar roles is something hard to achieve, then the adoption of the proposed solutions might be hard to achieve as well.

6. Conclusions

Creating stable automated tests is a difficult goal to accomplish. Flaky tests are present at all testing levels such as unit tests, integration tests, functional tests and non-functional tests. We have studied that there are many causes for non-deterministic tests. Additionally, there are some workarounds that may help software development teams to fix them. However, according to different authors, flaky tests are still a problem for organizations which try to get a continuous software development approach like continuous delivery.

In this paper, we have described the experience of a company that has faced this issue and that has implemented solutions to solve flaky tests for REST services. Based on the successful implementation of these solutions, we have proposed a set of key success factors for stopping flaky tests in this type of testing. We believe that this might be a little contribution and a starting point for avoiding flaky tests in one type of testing.

Finally, in future works we will continue working on exploring solutions to stop flaky tests in other types of testing like UI testing. We also will work on finding more solutions that may contribute to the improvement of the testing process in continuous software development environments.

Acknowledgements

This work has been supported by two research projects. One of the projects is “Metodologías y herramientas emergentes para contribuir con la calidad del software” (PI 17F018 SCyT UNNE). The other project is “Análisis e Implementación de tecnologías emergentes en sistemas computacionales de aplicación regional” (PI 17F017 SCyT UNNE).

We also thank Atif Chaudhry who provided language help for the preparation of this paper.

Competing interests

The authors have declared that no competing interests exist.

References

[1] M. Fowler, “Eradicating non-determinism in tests”, 2011. Available at: `https://martinfowler.com/articles/nonDeterminism.html` Accessed on 2018-01-27

[2] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests”, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 643-653, 2014.

[3] F. Lacoste, “Killing the gatekeeper: Introducing a continuous integration system,” in *Agile Conference ‘09*, pp. 387-392, 2009.

[4] P. Sudarshan. “No more flaky tests on the Go team”. Available at: `https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team` Accessed on 2018-01-27

[5] TotT. “Avoiding flakey tests”. Available at: `https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html` Accessed on 2018-01-27

[6] “Flakiness dashboard HOWTO”. Available at: `http://www.chromium.org/developers/testing/flakiness-dashboard` Accessed on 2018-01-27

[7] J. Micco, “Flaky Tests at Google and How We Mitigate Them”, 2016. Available at: `https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html` Accessed on 2018-01-27

[8] L. Hukkanen. *Adopting Continuous Integration – A Case Study*. M.Sc. thesis, Aalto University, 2015.

[9] E. Laukkanen, T. O. A. Lehtinen, J. Itkonen, M. Paasivaara, and C. Lassenius, “Bottom-up Adoption of Continuous Delivery in a StageGate Managed Software Organization,” in *10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1-10, 2016.

[10] F. Cannizzo, R. Clutton and R. Ramesh, "Pushing the boundaries of testing and continuous integration," in *IEEE Agile Conference '08*, pp. 501-505, 2008.

[11] J. Downs, J. Hoskins and B. Plimmer, “Status Communication in Agile Software Teams: A Case Study”, in *Fifth International Conference on Software Engineering Advances*, pp. 82-87, 2010.

[12] G. Gruver, M. Young, and P. Fulghum, *A Practical Approach to LargeScale Agile Development: How HP Transformed LaserJet FutureSmart Firmware*, New York, NY, USA: Addison-Wesley Professional, 1 ed., 2012

[13] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *IEEE Agile Conference '13*, pp. 121-128, 2013.

[14] J. Süß and W. Billingsley, “Using continuous integration of code and content to teach software engineering with limited resources,” in *Proceedings of the 34th International Conference on Software Engineering*, pp. 1175-1184, 2012.

[15] A. Debbiche, M. Diener, and R. Berntsson Svensson, “Challenges When Adopting Continuous Integration: A Case Study,” *Product-Focused Software Process Improvement, ser. Lecture Notes in Computer Science*, vol. 8892, pp. 17–32, 2014.

[16] L. Elloussi, *Determining flaky tests from test failures*. Master Thesis, University of Illinois at Urbana-Champaign, 2015.

[17] A. Miller, “A hundred days of continuous integration,” in *IEEE Agile Conference ‘08*, pp. 289-293, 2008.

[18] Q. Luo, L. Eloussi, F. Hariri and D. Marinov, “Can We Trust Test Outcomes?”, 2014. Available at: `https://pdfs.semanticscholar.org/a4b2/f4b9bcfdd0e83323570c40b893310f41e979.pdf` Accessed on 2018-01-27

[19] M. Collin, *Mastering Selenium WebDriver*, Birmingham, UK, Packt Publishing, 2015.

[20] F. Palomba and A. Zaidman, “Does refactoring of test smells induce fixing flaky tests?” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSME), pp. 1-12, 2017.*

[21] A. Gyori, *Proactively detecting unreliable tests.* PhD Thesis, University of Illinois at Urbana-Champaign, 2017.

[22] G. Brajnik, A. Baruzzo and S. Fabbro, “Model-based continuous integration testing of responsiveness of web applications,” in *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1-2, 2015.

[23] E.G. Smith, “Automated Test Results Processing,” in *Proceedings of the STAREAST 2001 Conference*, pp. 1-13, 2001.

[24] M.A. Mascheroni, M.K. Cogliolo and E. Irrazábal. “Automatic detection of Web Incompatibilities using Digital Image Processing,” *Electronic Journal of Informatics and Operations Research (SADIO EJS), Special Issue dedicated to JAIIO 2016*, Vol. 16, No. 1, pp. 29-45, 2017.

[25] E. Dolstra, R. Vliegendhart and J. Pouwelse, "Crowdsourcing GUI tests," in *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 332-341, 2013.

[26] B. Busjaeger and T. Xie, “Learning for test prioritization: an industrial case study,” in *Proceedings of the 24th International Symposium on Foundations of Software Engineering*, pp. 975–980, 2016.

[27] M. Eyl, C. Reichmann, and K. Müller-Glaser, “Fast feedback from automated tests executed with the product build,” in *Proceedings of the International Conference on Software Quality*, pp. 199–210, 2016.

[28] S. Elbaum, G. Rothermel, and J. Penix. “Techniques for improving regression testing in continuous integration development environments,” in *Proceedings of the International Symposium on Foundations of Software Engineering*, pp. 235–245, 2014.

[29] E. Laukkanen, J. Itkonen, and C. Lassenius. “Problems, Causes and Solutions When Adopting Continuous Delivery - A Systematic Literature Review,” *Information and Software Technology*, Vol. 8, pp. 55-79, 2016.

[30] J. Penix, “Large-scale test automation in the cloud”, in *Proceedings of the 34th IEEE International Conference on Software Engineering (ICSE)*, page 1122, 2012.

[31] M. Gousset, A. Krishnamoorthy, B. Keller and S. Timm, *Professional Application Lifecycle Management with Visual Studio 2010: with Team Foundation Server 2010*, New Jersey, USA, Wiley Publishing, 2010.

[32] P. Duvall, S. Matyas and A. Glover, *Continuous integration: improving software quality and reducing risk*, Addison-Wesley, 2007.

[33] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*, Boston, Addison-Wesley, 2011.

[34] M. Erder and P. Pureur, *Continuous architecture: Sustainable architecture in an agile and cloud-centric world*, Morgan Kaufmann, 2015.

[35] E. Givoni, N. Albert, Z. Ravitz, T.Q. Nguyen and T. Nguyen, "Automated software testing and validation system", 2006, U.S. Patent No US 7,093,238 B2.

[36] S. Khurshid, C.S. Păsăreanu and W. Visser, “Test input generation with Java PathFinder: then and now”, *in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1-2, 2018.

[37] J. Wolf and S. Yoon, “Automated Testing for Continuous Delivery Pipelines” (industrial talk), in *Pacific NW Software Quality Conference*, 2016.