Evaluating the User Acceptance Testing for Multi-tenant Cloud Applications

Victor Hugo Santiago C. Pinto, Ricardo R. Oliveira, Ricardo F. Vilela and Simone R. S. Souza
Institute of Mathematical and Computer Sciences, University of São Paulo (ICMC-USP)

Keywords: Multi-tenancy, Cloud Applications, User Acceptance Testing.

Abstract: SaaS (Software as a Service) is a service delivery model in which an application can be provided on demand via the Internet. Multi-tenant architecture is essential for SaaS because it enables multiple customers, so-called tenants, to share the system's resources in a transparent way to reduce costs and customize the software layer, resulting in variant applications. Despite the popularity of this model, there have been few cases of evaluation of software testing in cloud computing. Many researchers argue that traditional software testing may not be a suitable way of validating cloud applications owing to the high degree of customization, its dynamic environment and multi-tenancy. User Acceptance Testing (UAT) evaluates the external quality of a product and complements previous testing activities. The main focus of this paper is on investigating the ability of the parallel and automated UAT to detect faults with regard to the number of tenants. Thus, our aim is to evaluate to what extent the ability to detect faults varies if a different number of variant applications is executed. A case study was designed with a multi-tenant application called iCardapio and a testing framework created through *Selenium* and *JUnit* extensions. The results showed a significant difference in terms of detected faults when test scenarios with a single-tenant and multi-tenant were included.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Cloud computing emerged from the application of several technologies such as parallel computing, distributed computing and virtualization (Ru and Keung, 2013). This model makes applications, infrastructures and platforms available as an on-demand service via the Internet. In the case of applications, SaaS (Software as a Service) is the service model that is most likely to be adopted (Buxton, 2015).

Multi-tenant Architecture (MTA) is a key mechanism for SaaS that allows a single instance of the application to be transparently shared among multiple customers, the so-called tenants, and offers exclusive configuration according to particular needs in terms of business rules and interfaces. System resources such as services, applications, database or hardware are shared to reduce costs (Kabbedijk et al., 2015). A single-tenant application requires a dedicated set of resources to properly address the needs of a particular organization, while a multi-tenant application can serve multiple companies from a single shared instance because tenants exploit the benefits of a service through a virtual isolation of the application and its data (SalesForce, 2008). In this way, by employing a multi-tenant SaaS system several variant applications can be provided for particular tenants.

Despite the growing popularity of multi-tenancy and its benefits to businesses, there is an increasing lack of data control and trust in cloud environments as a result of this architectural pattern, since resource sharing by a single customer can have an impact on all the users. It remain unclear what are the key issues in cloud computing for multi-tenant cloud applications with regard to vulnerability, requirements, security questions and potential faults (Bai et al., 2011; Gao et al., 2011; Vashistha and Ahmed, 2012; Ru et al., 2014). In view of this, security issues in cloud environments must be addressed to ensure that there is a secure deployment of cloud applications. In addition, the testing strategies should be defined so that the quality of these application can be improved (Subashini and Kavitha, 2011).

The purpose of software testing is to ensure system consistency by detecting any faults in time for them to be repaired (Ostrand and Balcer, 1988). The information obtained during the test execution can help in the debugging activity. In general, the testing activity must include the following stages: planning and designing of the test case, execution of the tests

Hugo Santiago C. Pinto, V., R. Oliveira, R., F. Vilela, R. and R. S. Souza, S.
Evaluating the User Acceptance Testing for Multi-tenant Cloud Applications.
DOI: 10.5220/0006664000470056
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 47-56
ISBN: 978-989-758-295-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

and analysis of the results (Myers and Sandler, 2004). User Acceptance Testing (UAT) is a type of black-box testing that is carried out to examine an application from user actions. As the testing procedure can be very expensive in terms of time and effort spent, and is often impracticable if undertaken manually, a solution that is widely adopted is automated testing. For instance, *Selenium*[1] is a suite of tools for automating web browsers across many platforms.

Although there have been some recent research studies on software testing in cloud computing, we found that there is a lack of empirical studies and guidelines on which techniques and automation tools to address particular objectives (Pinto et al., 2016). Since multi-tenant SaaS systems have certain similarities with other models, investigating testing techniques and tools for traditional programs can provide valuable information. However, we did not find any empirical study involving UAT and multi-tenant cloud applications in the literature.

This paper presents a case study using a multi-tenant SaaS system for creating electronic restaurant menus called “iCardapio” (Manduca et al., 2014). This system was deployed in a cloud environment called Heroku[2], a Platform as a Service (PaaS) that enables developers to build, run and operate applications entirely in cloud. On the basis of this application, we applied UAT using parallel testing to detect faults, which included test scenarios with both a single-tenant and multi-tenant. A fault is found when the output of a test case is different from the expected result, in accordance with the specification of the iCardapio. This means that the faults are not controlled, that is, they are real. Variant applications from iCardapio must be isolated so that the faults can be detected without external interference. On the other hand, simulating a real-world multi-tenant scenario with requests from tenants to single and shared instance of the application, is essential to detect other types of faults. The aim of the study is to report how far this approach is able to determine specific faults for variant applications and assess their impact when the user’s operations are carried out simultaneously with different tenants. The results of this exploratory study can help developers/testers in determining possible faults, by employing a tenant identification and persistence strategy.

A testing framework was created with *JUnit*[3] extensions and *Selenium* to automate parallel tests. In the context of web applications, parallel testing is the process of running multiple test cases in multiple combinations, with OS and browsers at the same time. Its main advantage is that it allows the testers and developers to devote their meaningful resources to serious problems in the interests of cross-platform compatibility. However, within the scope of our case study, the reason for carrying out parallel testing is to simulate simultaneous requests from different tenants to a single and shared instance of the iCardapio. With regard to generating test cases, functional testing criteria such as Equivalence Partitioning Criteria (EPC) (Roper, 1994; Copeland, 2004) and Boundary Value Analysis (BVA) (Myers and Sandler, 2004) were applied.

The remainder of the paper is structured as follows: Section 2 provides an overview of multi-tenancy, testing activities and other concepts. Section 3 examines the case study. Section 4 addresses the problem of threats to the validity of the study and Section 5 summarizes the conclusions and makes suggestions for future work.

[1]http://www.seleniumhq.org/
[2]https://www.heroku.com/
[3]http://junit.org/junit4/

2 BACKGROUND

This section outlines the main concepts of multi tenancy for cloud applications and software testing.

2.1 Multi-tenancy

SaaS is a cloud delivery model that allows applications to be provided by a cloud service provider and in which the customer does not need to monitor or control the underlying infrastructure, such as the network, servers, OS and storage or installation of applications and services (Mell and Grance, 2011). SaaS enables service providers to share their infrastructure by meeting the needs of a much larger number of customers simultaneously (Chong and Carraro, 2006). An architectural pattern known as multi-tenancy must be employed to maximize the use of resources by the tenants and determine which data belong to which customer.

MTA enables software vendors to serve multiple customers, the so-called tenants, and share the system resources in a transparent way, such as services, applications, databases, or hardware, to reduce costs. At the same time, it is still able to configure the system exclusively to the needs of the tenants with variant applications from a single online product (Kabbedijk et al., 2015). Thus, both virtualization and resource sharing must be carried out by the service provider since these are key aspects of multi-tenancy.

With regard to the application layer, implementations of multi-tenancy differ significantly and, for this

reason, appropriate decisions such as (i) identifying tenants and (ii) adopting a strategy for isolating the tenant's data, must take into account the target markets and requirements of the application.

Finding an approach for identifying the tenants is important for the implementation of multi-tenancy (Krebs et al., 2012). In making a multi-tenant aware application, it is necessary to isolate the tenants in nearly every part of an application. A "tenant ID" or the so-called tenant context, is generally used to identify each tenant and its resources. The following three approaches for implementing this identification are well-known to the developers: (i) custom http headers or OAuth[4] with bearer tokens; (ii) tenant ID in the URI (http://app.com/tenantid/...) and (iii) tenant ID in the hostname (http://tenantid.app.com). In the first alternative, the URL may be completely different for each variant application, whereas in the second, the tenant ID is included in the main domain. Finally, in the third choice, the tenant ID is shown at the beginning of URI.

With regard to the persistence of data, a suitable implementation model must be chosen for isolating the tenants' data. Three models can be highlighted (Chong and Carraro, 2006): (i) isolated databases - each tenant has its own database; (ii) isolated schema - the tenants share a common database where each tenant has its own schema and (iii) shared schema - data for all the tenants are stored in the same tables and identified through a tenant discriminator column. The first and second choices are probably the least intrusive, provide the highest level of isolation and do not vary significantly in the way they implement the application. The third can be more vulnerable several threats with regard to data privacy, since these same tables are used for all the tenants, so that if there is an attack on a variant application, the data from the other tenants may become visible.

2.2 Software Testing

Software testing is a critical and dynamic activity in the development of software, since faults may appear in the software during the process. The application must be executed with appropriate test data so that these faults can be detected and the quality assurance of the software can be improved (Balci, 1994). Although this activity is not able to guarantee that a software system is free of defects, it can help to make it more reliable. Generally, software testing activities involve running an application with test cases provided by a tester, observing their results, comparing them with some expected values and reporting their consequences (Hoffman, 2001; Myers and Sandler, 2004).

As the testing usually cannot take into account all the possible inputs because the input space is too large or even infinite, testing criteria must be adopted to decide which test inputs to use and when to stop testing (Ostrand and Balcer, 1988). Functional testing criteria as equivalence partitioning criteria and boundary value analysis can be applied to define suitable test cases that cover equivalence classes with valid and invalid values, including upper and lower limits (Myers and Sandler, 2004).

User Acceptance Testing is a type of functional testing technique that involves validating software in real-world conditions. The goal is not just to check the requirements but to ensure that the software satisfies the customer's needs. User actions can be simulated for this to check if the requirements were met (Pusuluri, 2006). Automating these tests reduces the effort needed to run a large test suite, as well as assisting regression testing and iterative development. For instance, *Selenium*[5] is a suite of tools designed to automate web browsers across many platforms and offers an interface for test classes written with *JUnit*[6].

Parallel programming has become an increasingly common practice for cloud applications, insofar as it ensures the tests can benefit by dividing test activities into separate and parallel tasks to reduce costs. However, as the cloud environment changes its state continuously at runtime in proportion to the number of virtual machine instances, the effectiveness of hypervisor scheduling and load balancing, the testing may become a complex task because these factors cannot be fully predicted and controlled even for a single application being tested (Bai et al., 2011). Generally, in the case of conventional web applications, the parallel execution of test cases takes place to confirm multiple combinations with OS and browsers, support the regression testing (Garg and Datta, 2013) and solve problems concerning cross-platform compatibility.

3 THE CASE STUDY

This section describes the case study which was carried out to compare the number of detected faults including single-tenant and multi-tenant test scenarios. The multi-tenant SaaS system chosen was iCardapio because it is open-source and can address multiple tenants from a single instance. Thus, variant applications can be defined and configured for a simultaneous execution of the tests. According to the authors

[4]https://oauth.net/2/

[5]http://www.seleniumhq.org/projects/webdriver/

[6]http://junit.org/junit5/

(Manduca et al., 2014), iCardapio was developed with the aid of the following technologies: Spring[7], Spring Tool Suite[8], MySQL, Maven, EclipseLink[9] and Java. It should be noted that the source code is available to download from github[10] and was designed to allow any SaaS provider to download and start using it directly. The application was deployed in Heroku and is available[11].

Figure 1 shows an entity relationship diagram of this application. One tenant refers to a particular restaurant which has a single identifier, i.e., a tenant ID called "subdomain". Products have a single category, such as: pizza, pasta, drinks or desserts. In addition, there is only one user who has authenticated access and can carry out the registration operations for all the possible tenants.

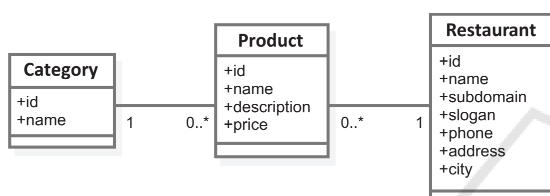

Figure 1: Entity relationship diagram of iCardapio.

General information about the project structure and complexity[12] was assessed by the code quality management tool SonarQube[13] and the data are displayed in Table 1.

Table 1: Information about iCardapio collected by SonarQube.

Code lines	1.552
Java	843
CSS	97
JavaScript	9
JavaServer Pages (JSP)	239
XML	166
Complexity	4.900
Complexity/classes	7.0
Complexity/method	2.3
Implementation units	37
Number of classes	20
Number of methods	2.159

Whith regard to the tenant identification, the application has the tenant ID in the hostname (http://tenantid.myapplication.com). Owing to the technical constraints of the cloud provider, the tenant identification was changed for the tenant ID in the URI (http://myapplication.com/tenantid/...). The isolation of the tenants' data is a key requirement when providing a multi-tenant application. As implementation model, iCardapio was developed by means of a shared schema, in which data for all the tenants are stored in the same tables and can be identified through a tenant discriminator column.

The automated tests were executed locally to ensure the test environment was controlled more effectively and the different types of faults could be clearly isolated. Thus, the user actions are simulated from different local instances of the browser. The test scripts can be executed automatically in the cloud through a TaaS (Testing as a Service), however, when using the cloud environment there is a lack of knowledge of how the tests will run. Moreover, the fact that the cloud environment changes its state continuously and the test execution depend on its runtime environment (which cannot fully predicted and controlled), means that the results may be distorted (Bai et al., 2011; Subashini and Kavitha, 2011).

3.1 Planning

The main reason for planning the case study was to answer the following research questions:

- *RQ*1: *Is there a different number of detected faults when either a single-tenant or multi-tenant test scenarios are used?* To answer this question, we gathered and assessed the number of detected faults when only one variant application from iCardapio was subjected to the tests and two variant applications were checked with the same tests in a parallel way. The purpose of this comparison is to provide evidence that in a multi-tenant scenario some external factors (such as simultaneous requests from users for variant applications) can affect their execution, and result in a larger number of faults than in a single-tenant scenario.
- *RQ*2: *Is the number of detected faults different when a multi-tenant test scenario is used and there is an increasing number of variant applications?* Similarly, when answering this second question, we analyzed the number of faults by taking account of the number of variant applications using parallel tests. In other words, we wanted to find out if an increasing number of variant applications can affect the number of detected faults.

The planning phase was divided into three parts that are described in the next subsections.

[7] https://spring.io/
[8] https://spring.io/tools/sts
[9] http://www.eclipse.org/eclipselink/
[10] https://github.com/michetti/icardapio/tree/multitenant
[11] https://icardapio.herokuapp.com/
[12] https://docs.sonarqube.org/display/SONAR/Metrics+-+Complexity
[13] https://www.sonarqube.org/

3.1.1 Experimental Setup

Conducting the case study included the following phases:

- Generation of test cases in accordance with EPC and BVA testing criteria for iCardapio based on its requirements;
- The *Selenium* and *JUnit* tools are used to automate the execution of these test cases;
- Deployment of iCardapio in Heroku;
- Development of a testing infrastructure that allows the parallel execution of the tests for multiple variant applications;
- Execution of the test classes for a unique variant application from single and shared instance of the iCardapio - single-tenant (ST) test scenario;
- Execution of the test classes in a parallel way for several variant applications from single and shared instance of the iCardapio - multi-tenant (MT) test scenario;
- Collecting, evaluating and reporting the test results.

There were 10 test executions in both the ST and MT scenarios, because this value was assumed to be a sufficient measure for a reliable analysis. With regard to the number of variant application, there was only one for ST and for parallel executions in the MT scenario - the numbers were defined as 2, 5 and 10. The execution order between the variant applications is not pre-defined, which makes a real and non-deterministic behavior likely.

In the case of the MT test scenario, the average must be calculated on the basis of the number of variant applications (n) under test (2, 5 and 10), as expressed in Equation 1. For instance, from 10 executions of 2 variant applications ($n = 2$) carried out in a parallel way and the detection of their corresponding faults, the number of identified faults are added and divided by n for each execution. Finally, the averages by execution are added and divided by 10. The same calculation must be made when $n = 5$ and $n = 10$.

$$\frac{1}{10}\sum_{z=1}^{10}\left[\frac{1}{n}\sum_{i=1}^{n}x_i\right] \quad (1)$$

Regarding the defined host for test execution, the settings are as follows: Ubuntu 16.04 64 bits, CPU Intel Core i5-3210M 2.5 GHz, 16 GB of RAM, 250 GB SSD and AMD Radeon HD 7670M - 1 GB. Firefox was the browser chosen for running the tests. It should be pointed out that in case of parallel executions, *Selenium* was used for running tests in multiple runtime environments, specifically when there were different and isolated instances of the browser at the same time.

3.1.2 Hypothesis Formulation

The $RQ1$ was formalized as follows: **Null hypothesis, H_0:** There is no difference between the ST and MT test scenarios in terms of detected faults, that is, the test scenarios are equivalent.

Alternative hypothesis, H_1: There is a difference between the ST and MT test scenarios in terms of detected faults; this means that the test scenarios are not equivalent. Hypotheses for the $RQ1$ can be formalized by Equations 2 and 3:

$$H_0 : \mu_{ST} = \mu_{MT} \quad (2)$$

$$H_1 : \mu_{ST} \neq \mu_{MT} \quad (3)$$

Similarly, $RQ2$ was also formalized in two hypothesis, as follows: **Null hypothesis, H_0:** There is no significant difference in terms of faults found during the test execution, since there are a different number of variant applications (n and p) running at the same time.

Alternative hypothesis, **H_1:** There is a difference in the number of faults found during the tests, since there is a different number of variant applications running simultaneously. Similarly, the hypotheses for *RQ2* can be formalized by Equations 4 and 5:

$$H_0 : \mu_{MT^n} = \mu_{MT^p} \mid n \neq p \quad (4)$$

$$H_1 : \mu_{MT^n} \neq \mu_{MT^p} \mid n \neq p \quad (5)$$

3.1.3 Variable Selection

The dependent variables are (i) the "**number of detected faults from the test executions**" and (ii) "the **number of variant applications being used simultaneously in testing**". The independent variables are: (i) "**iCardapio**", (ii) **test scenarios** with a single-tenant and multi-tenant, (iii) **testing framework for parallel execution** established with the support of *Selenium* and *JUnit* and (iv) a **test environment** that includes test cases and scripts for simulating user actions.

3.2 Operation

Once the case study had been defined and planned, it was carried out in the following stages: preparation, execution and data validation.

3.2.1 Preparation

At this stage, we were concerned with preparing the material needed, such as data collection forms and the testing environment variables. iCardapio was deployed in Heroku and a testing infrastructure was defined for a parallel execution of the test classes and variant applications.

In the ST test scenario, a single thread for running the tests might be enough, but there are many threads for MT test scenario that must be executed in parallel because several variant applications will be testes at the same time. However, we cannot be sure that all the submitted tasks will be executed in the same amount of time or before some event, because this depends on thread scheduling and it cannot be controlled (on account of its non-deterministic behavior). Even if we submit all the tasks at the same time, this does not imply that they will be executed simultaneously. Despite these limitations, a framework was developed from *JUnit* extensions, (such as the Parameterized[14] runner) so that the tests could be conducted in a parallel way. Test classes were formed with the aid of *Selenium* and *JUnit* to automate and simulate possible user actions in iCardapio. Additionally, other objects were taken into account such as the following:

- *Documentation of the application:* technical guidance documents to assist in the development of iCardapio and which are useful to define test cases;
- *Test Case forms:* forms to be filled in with the test cases in accordance with EPC and BVA criteria;
- *Data Collection form:* a document with blank spaces to be filled in with the number of detected faults obtained from the test execution and variant application.

The platform used to conduct the experiment employed Java as its implementation language and the Eclipse IDE as the development environment.

3.3 Data Analysis

This section sets out our findings. The analysis is divided into two areas: (i) descriptive statistics and (ii) hypothesis testing.

3.3.1 Descriptive Statistics

Descriptive statistics of the gathered data from the case study are shown in Table 2. The first column contains the execution order and the remaining columns, the number of faults detected in each variant application. A total number of 95 test cases were defined that followed the requirements of iCardapio and its possible execution paths, from the users' perspective. On the basis of this principle, the number of faults reported in this study refers to failed test cases. It should be noted that the data are grouped by the number of variant applications. This led to four groups being categorized which are as follows: the first with only "t1" represents a single variant application, the second contains data from two variant applications ("t1' and "t2", $n = 2$) executed simultaneously and so on.

Table 2: Gathered data from the results of the automated tests.

Exec.	t1
1st	34
2nd	34
3rd	34
4th	34
5th	34
6th	34
7th	34
8th	34
9th	34
10th	34

t1	t2
37	39
38	40
37	39
40	42
38	40
37	39
38	40
37	42
39	42
38	39

t1	t2	t3	t4	t5
48	47	47	51	47
50	49	49	49	48
47	47	47	48	50
48	48	48	51	49
47	48	49	50	47
47	47	48	50	49
51	49	49	49	48
47	47	47	49	50
48	48	47	50	49
47	47	51	48	51

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
1st	46	42	46	45	44	43	51	47	42	45
2nd	48	46	49	50	42	42	43	49	47	49
3rd	46	47	48	42	43	50	51	48	49	48
4th	47	43	47	45	44	44	49	48	49	46
5th	47	43	45	49	51	50	49	48	44	45
6th	47	42	47	46	45	44	51	50	43	46
7th	47	43	47	43	45	47	51	50	45	46
8th	46	42	47	46	45	50	51	48	43	46
9th	47	46	45	44	44	43	52	48	43	46
10th	47	48	48	45	46	50	49	48	49	51

Before applying the statistical methods, we determined the quality of the input data. Incorrect data sets may be formed because of systematic errors or the presence of outliers, which are data values that are much higher or much lower than expected when compared with the remaining data. For this reason, we used box plot as a means of identifying the outliers. Figure 2 shows the box plot based on the faults detected in the ST and MT test scenario, where $n = 1$, $n = 2$ (t1...t2), $n = 5$ (t1...t5) and $n = 10$ (t1...t10). Note that in the ST test scenario, the number of faults found was the same for all the executions. In contrast, in case of the MT test scenario, the results from Equation 1 for $n = 2$, $n = 5$ and $n = 10$ were as follows:

[14] https://goo.gl/DS6Gkg

39.05, 48.44 and 46.49 identified faults (on average and including all the executions), respectively.

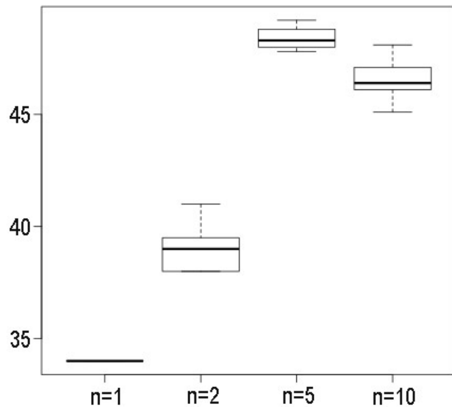

Figure 2: Box plot for the detected faults with an increasing number of variant applications.

3.3.2 Hypothesis Testing

In this section, there is a discussion of hypothesis testing for research questions.

RQ1 **- Hypothesis Testing:** Since some statistical tests are only valid if there is a normal population distribution, before choosing a statistical test we examined whether our gathered data departs from linearity. This led us to use the Shapiro-Wilk test for the collection of detected faults. With the data concerning the t1 and $n = 2$ (t1..t2), Shapiro-Wilk statistic W was calculated as 0.874, the critical value of W (5% significance level) was 0.930 and the *p-value* was 0.002061, assuming $\alpha = 0.05$. Therefore, we reject the null hypothesis that the data are from a normally distributed population, as calculated W is less than the critical value of W and $p < 0.05$. In addition, the test *Q-Q* plot was applied, which is plotted in Figure 3.

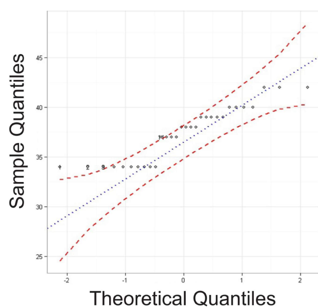

Figure 3: A normality test for gathered data when a single application and two variant applications were subjected to automated testing.

Subsequently, we applied the Wilcoxon-Mann-Whitney Test, which is a suitable non-parametric test for two unpaired samples. This statistical test evaluates the existence of significant differences in continuous and ordinal-level dependent variables. The alternative hypothesis H_1 is that the number of detected faults when running a single variant application (t1) and running two variant applications simultaneously, i.e., a MT test scenario with $n = 2$, are non-identical populations. As a result, the p-value is $7.186e^{-06}$; therefore, at a significance level of 0.05, there is a difference between the samples.

RQ2 **- Hypothesis Testing:** in a similar way, the Shapiro-Wilk normality test was used, for the data concerning $n = 2$, $n = 5$ and $n = 10$. As a result, W is 0.922, critical value of W (5% significance level) was 0.984 and *p-value* was $6.797e^{-08}$, assuming $\alpha = 0.05$. Therefore, we also reject the null hypothesis for RQ2 that the data are from a normally distributed population. Figure 4 shows a *Q-Q* plot for gathered data that only includes the MT test scenario.

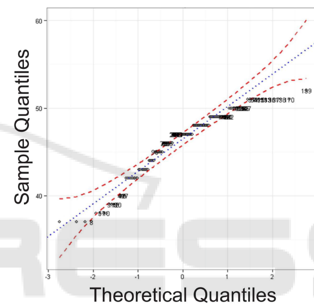

Figure 4: A normality test for gathered data when 2, 5 and 10 variant applications were subjected to automated testing.

Afterwards, we applied the Kruskal-Wallis Test, which is a non-parametric test that evaluates significant differences in continuous and ordinal-level dependent variables by a categorical independent variable with two or more unpaired samples. As a result, the Kruskal-Wallis chi-squared was 68.764, $df = 2$, H was 67.94, critical chi-square was 5.991 and the p-value was $1,221e^{-15}$, when $\alpha = 0.05$. If it is assumed that $p < 0.05$ and the critical chi-square is less than the H statistic, the null hypothesis that the medians are equal can be rejected. Therefore, there is a significant difference between the three samples ($n = 2$, $n = 5$ and $n = 10$).

As the alternative hypothesis H_1 assumes that at least one of the samples comes from a different population than the others, we applied the Bonferoni method for multiple comparisons, with FWER = 0.05. This allowed us to conclude that each sample is statistically different. The difference between $n = 10$ and $n = 2$ was 73.025, with upper limit of 50.58 and lower limit of 95.48. For $n = 10$ and $n = 5$, the difference was -34.14, with an upper limit of $-50,02$ and lower limit of $-18,27$. Finally, between $n = 2$ and $n = 5$, it

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

was $-107,165$, with an upper limit of -131.42 and lower limit of -82.92. With regard to the groupings, the rankings for $n = 5$, $n = 10$ and $n = 2$ are as follows: 118.19, 84.05 and 11.025, respectively.

Additionally, an effects plot was defined from the lower limit, effect and upper limit, by analyzing these three samples separately, as shown in Figure 5. For $n = 10$, the values were as follows: 46.049, 46.49 and 46.931; with $n = 2$: 38.063, 39.05 and 40.036; and finally, for $n = 5$: 47.815, 48.44 and 49.064.

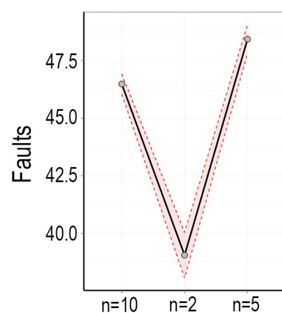

Figure 5: Effects plot for the MT test scenario.

3.3.3 Discussion on the Nature of the Detected Faults

A) *Single-tenant faults:* after analyzing the detected faults when only a single variant application was determined from the automated tests, we concluded that all of them are functional and have typical web faults. 61 out of 95 test cases passed, i.e, the behavior of the application was consistent with its specifications in 64% of the tests. The detectability of faults in the single-tenant test scenario did not require a considerable effort, since the tests are conducted at the unit level and this enables the faults to be discovered clearly in the source code from the user actions in the web pages. It should be noted that there was no competition with other tenants for the same resources and it is possible that there are other kinds of faults with regard to performance, security and data privacy that were not considered to be within the scope of this study.

B) *Multi-tenant faults:* an important question is how to detect faults, which can only be discovered during runtime. For this reason, automated and parallel UAT can be regarded as a valid approach to examine these kinds of faults. Within the multi-tenant context, it is clear that there are problems related to concurrent behavior in cloud applications that must be further investigated in the testing activity.

The challenge of detecting a fault in this type of program, mainly arises from the synchronization process (i.e. the threads). The tenants compete for shared resources that are made available on demand; that is, a scheduling policy must share and distribute the resources in an equitable manner. This means that we cannot control the execution order of the processes, i.e., the data dependency and inter-process synchronization of a program might be affected during the scheduling. With this in mind, a program can carry out different execution sequences, even with the same test input. Hence, the testing activity must check whether the possible synchronization sequences were executed and if the outputs obtained are correct. This difficulty is well-known among practitioners in the concurrent programming field, as being a non-deterministic kind of behavior. There are several problems in testing concurrent applications due to this characteristic, such as high computational costs and many false positives (Arora et al., 2016).

Another concept applicable to the multi-thread context is thread safety. A piece of code is thread-safe when it can manipulate shared data structures in such a way that it can ensure a secure execution across multiple threads at the same time (Lewis and Berg, 1995). Similarly, variant applications can be executed from a single instance of a multi-tenant SaaS system that must manipulate shared data structures. Regardless of the persistence strategy, each variant application must be connected to a specific database/schema. To achieve this, the multi-tenant system must identify the tenant and be able to provide users with the expected data, interface and features.

The results of the case study demonstrated that there are differences when running the same test cases when they include both a single and several variant applications at the same time. Update Synchronization Failure is the most likely reason for the number of faults in an MT test scenario, because many test executions have led to unexpected ways. For instance, when users carried out the same actions in different variant applications, specific execution paths were affected. This happened when the user logged on and logged out in a certain variant application. As these deviations adversely affected the expected execution, several faults were reported in the test environment.

With regard to faults in tasks carried out concurrently by tenants, the parallel execution of the variant applications has enabled us to identify some faults that could not be detected because they only included one isolated application. Figure 6 shows the results of the parallel execution for a test class that verifies the field “name” of the products. In this execution, two variant applications were verified with simulated requests. It should be pointed out that the number of detected faults is different in each execution, even with the same input and features provided by variant applications.

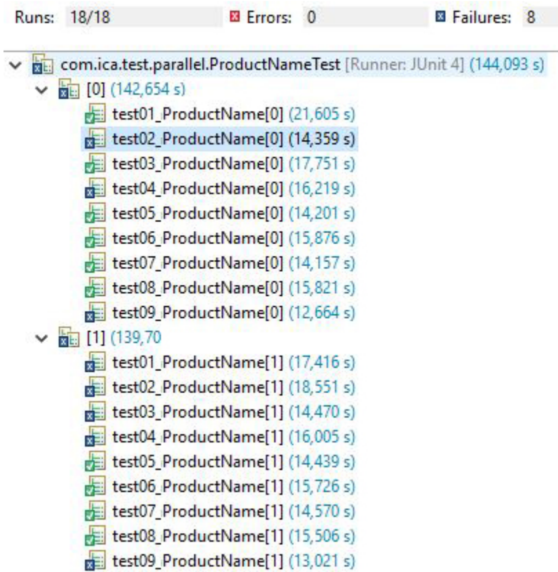

Figure 6: Results of automated and parallel tests of two applications that verify the product names.

4 THREATS TO VALIDITY

Test executions: the processing capacity of the host where the tests were conducted and the robustness of the cloud provider can influence the test results (Bai et al., 2011). However, this influence was not measured and compared with other settings. To mitigate this threat, all the tests were conducted in the same hardware configuration and operating system.

Although we have defined ten executions, there may be unidentified faults in the application or false positives. As the faults examined in this case study were not seeded (as in a controlled environment with a predefined set of possible faults), the test cases were manually defined to explore the execution flows from the user's standpoint and in accordance with the requirements of iCardapio, and for this reason, the faults were reported from this set of test cases.

Test cases: designing test cases is an important matter that can affect the collected data and make the case study less impartial. A test professional who only had access to the application's documentation and GUI, was invited to ensure impartiality, and avoid bias by not knowing the source code of the application.

Interaction between configuration and handling: it is possible that the exercises carried out in the case study are not accurate enough for any real-world applications. However, the main value of this empirical study is that it has provided evidence that if the testing activity only includes a variant application, it is unable to detect the main faults owing the external factors such as the resource sharing and the competition among variant applications, which can result in non-deterministic effects.

Assessing the degree of reliability: this refers to the metrics used to measure the number of faults. To mitigate this threat we carried out ten executions which can be regarded as a sufficient measure, since the number of detected faults was recorded in forms filled in after each execution of the variant applications.

Low statistical power: the ability of a statistical test to reveal reliable data. We applied two tests, the Wilcoxon-Mann Whitney test to statistically analyze the differences between two unpaired samples for *RQ1* and the Kruskal-Wallis test to analyze the number of faults found when a different number of variant applications were verified simultaneously, in answer to *RQ2*.

5 CONCLUSION

The focal point of this study was to investigate the parallel and automated UAT for multi-tenant SaaS systems. A case study was designed using a multi-tenant application called iCardapio. Test cases were defined by applying functional criteria and conducting automated tests with *JUnit* and *Selenium*. The aim of this study was to compare the number of faults by taking into account test scenarios with single-tenant and multi-tenants. The main findings of our experiment showed that, in terms of detected faults, there is a statistical difference between the samples. In addition to faults of functional and web nature, there was evidence of update synchronization failure and faults arising from a non-deterministic behavior. Although the results are provisional, given the limitations of the study, they can demonstrate that these questions must be included in the testing of multi-tenant cloud applications.

A package containing the tools, materials and more details about the experiment steps is available at https://goo.gl/ESNCGD. In future studies, we intend to explore the following factors: (i) the feasibility of low-cost fault injection in multi-tenant SaaS systems to examine the mechanisms of fault detection; (ii) defining a tenant-oriented testing process that takes into account faults and test levels when supporting the development of high-quality cloud applications, and (iii) conducting controlled experiments to investigate the use of a TaaS (e.g., *Sauce Labs*[15]) that takes ac-

[15]https://saucelabs.com/

count an increasing number of variant applications.

Despite the growing popularity of the MTA, there are still very few empirical studies on the techniques and testing criteria employed in the cloud domain. We believe that an essential requirement for more targeted testing techniques in this field, is to evaluate the testing strategies applied in other contexts.

ACKNOWLEDGEMENTS

The authors are grateful to CAPES, ICMC/USP and Federal Institute of Education, Science and Tecnology of South of Minas Gerais - IFSULDEMINAS, Poços de Caldas campus, for supporting this work.

REFERENCES

Arora, V., Bhatia, R., and Singh, M. (2016). A systematic review of approaches for testing concurrent programs. *Concurr. Comput. : Pract. Exper.*, 28(5):1572–1611.

Bai, X., Li, M., Chen, B., Tsai, W.-T., and Gao, J. (2011). Cloud testing tools. In *Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on*, pages 1–12. IEEE.

Balci, O. (1994). Validation, verification, and testing techniques throughout the life cycle of a simulation study. *Annals of operations research*, 53(1):121–173.

Buxton, A. (2015). The reasons why SaaS will remain the dominant cloud model - Techradar. https://tinyurl.com/yc7ktewl. [Online; accessed 10-April-2017].

Chong, F. and Carraro, G. (2006). Architecture strategies for catching the long tail. In *Microsoft Corporation*, page 910. MSDN Library.

Copeland, L. (2004). *A practitioner's guide to software test design*. Artech House.

Gao, J., Bai, X., and Tsai, W.-T. (2011). Cloud testing-issues, challenges, needs and practice. *Software Engineering: An International Journal*, 1(1):9–23.

Garg, D. and Datta, A. (2013). Parallel execution of prioritized test cases for regression testing of web applications. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference-Volume 135*, pages 61–68. Australian Computer Society, Inc.

Hoffman, D. (2001). Using oracles in test automation. In *Proceedings of Pacific Northwest Software Quality Conference*, pages 90–117.

Kabbedijk, J., Bezemer, C.-P., Jansen, S., and Zaidman, A. (2015). Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software*, 100(0):139 – 148.

Krebs, R., Momm, C., and Kounev, S. (2012). Architectural concerns in multi-tenant saas applications. *CLOSER*, 12:426–431.

Lewis, B. and Berg, D. J. (1995). *Threads Primer: A Guide to Multithreaded Programming*. Prentice Hall Press, Upper Saddle River, NJ, USA.

Manduca, A. M., Munson, E. V., Fortes, R. P., and Pimentel, M. G. C. (2014). A nonintrusive approach for implementing single database, multitenant services from web applications. In *Proc. of the 29th Annual ACM Symposium on Applied Computing*, pages 751–756.

Mell, P. and Grance, T. (2011). The NIST definition of cloud computing - recommendations of the national institute of standards and technology. Technical report, National Institute of Standartization.

Myers, G. J. and Sandler, C. (2004). *The Art of Software Testing*. John Wiley & Sons.

Ostrand, T. J. and Balcer, M. J. (1988). The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686.

Pinto, V. H. S. C., Luz, H. J. F., Oliveira, R. R., Souza, P. S. L., and Souza, S. R. S. (2016). A Systematic Mapping Study on the Multi-tenant Architecture of SaaS Systems. In *28th Int. Conf. on Software Engineering & Knowledge Engineering (SEKE)*, pages 396–401.

Pusuluri, N. R. (2006). *Software Testing Concepts And Tools*. Dreamtech Press.

Roper, M. (1994). *Software Testing*. McGraw-Hill Book Company Europe.

Ru, J., Grundy, J., and Keung, J. (2014). Software engineering for multi-tenancy computing challenges and implications. In *Proc. of the Int. Workshop on Innovative Software Development Methodologies and Practices*, pages 1–10. ACM.

Ru, J. and Keung, J. (2013). An empirical investigation on the simulation of priority and shortest-job-first scheduling for cloud-based software systems. In *Software Engineering Conference (ASWEC), 2013 22nd Australian*, pages 78–87. IEEE.

SalesForce (2008). The Force.com Multitenant Architecture - Understanding the Design of Salesforce.com's Internet Application Development Platform. https://goo.gl/zxYywi. [Online; accessed 3-September-2017].

Subashini, S. and Kavitha, V. (2011). A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 34(1):1–11.

Vashistha, A. and Ahmed, P. (2012). Saas multi-tenancy isolation testing challenges and issues. *International Journal of Soft Computing and Engineering*.