The Journal of Systems and Software 119 (2016) 122–135

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

# Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study

Hema Srikanth[a], Mikaela Cashman[b], Myra B. Cohen[b,*]

[a] IBM, Product Strategy Team, Enterprise Marketing Management, Waltham, MA, USA
[b] Dept. of Computer Science & Engineering, University of Nebraska-Lincoln, Lincoln, NE, USA

ARTICLE INFO

*Article history:*
Received 12 April 2015
Revised 2 April 2016
Accepted 8 June 2016
Available online 16 June 2016

*Keywords:*
Regression testing
Prioritization
Software as a service
Cloud computing

ABSTRACT

The use of cloud computing brings many new opportunities for companies to deliver software in a highly-customizable and dynamic way. One such paradigm, Software as a Service (SaaS), allows users to subscribe and unsubscribe to services as needed. While beneficial to both subscribers and SaaS service providers, failures escaping to the field in these systems can potentially impact an entire customer base. Build Acceptance Testing (BAT) is a black box technique performed to validate the quality of a SaaS system every time a build is generated. In BAT, the same set of test cases is executed simultaneously across many different servers, making this a time consuming test process. Since BAT contains the most critical use cases, it may not be obvious which tests to perform first, given that the time to complete all test cases across different servers in any given day may be insufficient. While all tests must be eventually run, it is critical to run those tests first which are likely to find failures. In this work, we ask if it is possible to prioritize BAT tests for improved time to fault detection and present several different approaches, each based on the services executed when running each BAT. In an empirical study on a production enterprise system, we first analyze the historical data from several months in the field, and then use that data to derive the prioritization order for the current development BATs. We then examine if the orders change significantly when we consider fault severity using a cost-based prioritization metric. We find that the prioritization order in which we run the tests does matter, and that the use of historical information is a good heuristic for this order. Prioritized tests have an increase in the rate of fault detection, with the average percent of faults detected (APFD) increasing from less than 0.30 to as high as 0.77 on a scale of zero to one. Although severity slightly changes which order performs best, we see that there are clusters of orderings, ones which improve time to early fault detection ones which don't.



## 1. Introduction

In industrial applications software testing is an essential, but expensive process, and it must be performed each time an application is modified (Beizer, 1990). Every change in the build of an application leads to the re-execution of the existing black box tests to validate those changes. With limited resources and time to market pressures, companies are often unable to complete their testing efforts within the allocated time, which can result in faults escaping into the field, and customer dissatisfaction. Shorter testing cycles (Elbaum et al., 2014; Lynch et al., 2013) can magnify this problem. With the advent of the cloud, many companies are moving into new computing and software licensing paradigms, some of which are highly dynamic, and these paradigms have the potential to reduce the time between testing even more, creating implications for software quality.

One such model that is being increasingly used by industry to provide software is called Software as a Service (or SaaS) (Mietzner et al., 2009; Goth, 2008). In SaaS, companies provide services (applications) online so that customers can access these services from anywhere via the web. They can subscribe and unsubscribe to the services based on their business needs. This type of business model, is *tenant-based*; users come in and *rent* (or pay) for the services as they are needed, and then leave when these no longer fulfill a business purpose. While SaaS has many benefits for both the customer and the provider, quality management has emerged as a major challenge. The systems must remain available twenty four hours, seven days a week, and since all customers use a single version of the application, the impact of faults which do escape into the field may be amplified. In a traditional software distribution model, customers use a range of versions and configurations of an application, and changes to these occur infrequently. This means

* Corresponding author. Tel.: +14024722305.
*E-mail addresses:* hema1900@gmail.com (H. Srikanth), mcashman@cse.unl.edu (M. Cashman), myra@cse.unl.edu (M.B. Cohen).

http://dx.doi.org/10.1016/j.jss.2016.06.017
0164-1212/

that failures which are observed at one customer's site may not be observed at another site, and each build is relatively stable. In SaaS, however, changes are made often, and the entire customer base has the potential to exercise any faulty code that slips into the release.

There has been a large body of work on regression testing during software maintenance (Rothermel et al., 1996; Rothermel and Harrold, 1997; Elbaum et al., 2002), much of which focuses either on test case selection or test case prioritization, and recently there has been research on continuous integration testing (Elbaum et al., 2014) which aims to improve the fast-paced continuous integration testing of code via prioritization. There have also been some proposals to model service commonality and variability for SaaS (Sengupta and Roychoudhury, 2011) which can impact testing. Yet there is little research that targets the SaaS test process directly, and little work aimed at its black box testing process in an industrial setting, which includes both integration and system testing. Lynch et al. (2013) present an overall testing process and tools for SaaS in an industrial setting, but do not address the specific challenges of how to select and prioritize test cases. In earlier work, we examined issues related to reliability in SaaS (Banerjee et al., 2010), and we analyzed SaaS field failures for the front end of an enterprise system that is used by many other SasS applications (Srikanth and Cohen, 2011). We applied our analysis to generate sequences of use cases that achieve broad coverage of the sequences of the application leading to prior field failures. We then prioritized these sequences. However, we looked at only a small part of a SaaS system.

In this paper we extend our industrial study to an end-to-end SaaS application, including the front-end with more than ten services. These services are bundled into many combinations of subscriptions, tailored to support the collaboration needs of premium enterprise customers. For many enterprise companies the test process comprises first of unit testing by development teams, and then black box testing which includes build acceptance tests (the focus of this work), functional tests, system tests, and reliability/stress tests. These are the very basic set of black box testing steps, all of which must also be executed in parallel in web and mobile environments. The complexity of testing at the black box level arises from the combinations of subscriptions and use cases within each of the subscriptions that need to be validated for each release.

Every time a new software build is released, the build goes through a build verification process (called a Build Acceptance Test, or BAT). The BATs consist of the most critical use cases, those that exercise mainstream scenarios for all services, and for all subscription-bundle combinations. Since BATs are black box, they have no access to code information. Only after the BATs successfully complete, do testers run additional feature or system tests. Lynch et al. (2013) discuss the fact that BATs represent a *go no-go* decision for the team. If the BATs pass, then the build testing and release can move to the next stage. If not, testing will be stopped. Ideally, BATs need to be validated on many unique servers such as in a development environment, a staging environment, and finally within the live environment where the build gets hosted for all customers to use. In addition, the number of use cases to validate the mainstream functionality grows as the services and features grow. The same test teams' efforts are divided to complete different tests in different environments, and the successful release depends on the successful completion of all these test efforts, not just one. In any given day, a single test team is thus spread thinly, using their efforts in a distributed manner. This means that BAT can become a bottleneck, as the time to new versions decreases.

In this paper, we ask if it is possible to prioritize BATs to decrease the time to early fault detection. Since we do not have access to code coverage, we use prioritization heuristics gathered from historical field failures. Our conjecture is that we can utilize what we have seen in past SaaS failure reporting systems, to guide prioritization for the current BATs. We utilize two metrics. First we examine if the number of services involved in a use case can impact the ability of a test case(s) to find faults. Second, we look at specific services that were associated with multiple failures in the past. Since the BAT already has chosen a critical set of use cases (which are likely based on the same prior information), it seems to follow that we can utilize this historical information to drive the new testing. However, until this point in time we do not have empirical evidence to draw this conclusion. Therefore, we perform an empirical study on a set of real development BATs for a large enterprise SaaS application and use a historical window to drive prioritization to validate this idea. We present a process that testers can follow and conclude with some lessons learned based on our results.

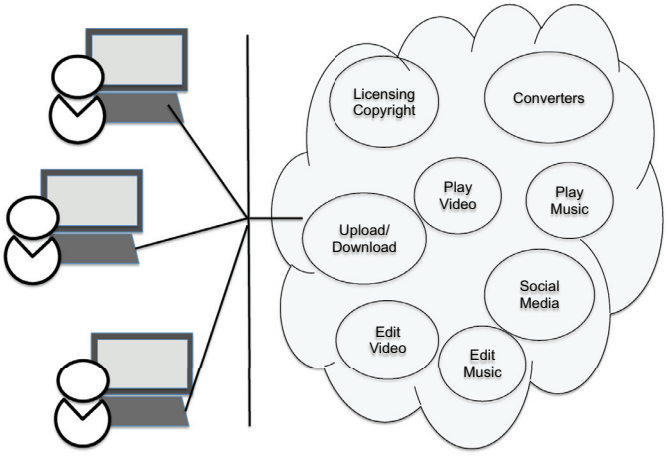


**Fig. 1.** Hypothetical Multi-Media SaaS. The services (on right) are all interconnected. The users (on left) choose which services they want to pay for through an interface. They can add and remove services dynamically.

The contributions of this work are:

1. A process for prioritizing Build Acceptance Tests based on historical field failures;
2. A set of black box prioritization heuristics for SaaS Build Acceptance Testing; and
3. An industrial case study on a large enterprise cloud production application showing the order of tests can significantly impact the order of finding problems prior to release.

The rest of this paper is structured as follows. In the next section we present some background and discuss related work. We follow this with a technical discussion of the prioritization schemes proposed in Section 3. We then present our empirical study (Sections 4–5). We end with conclusions and future work in Section 6.

## 2. Background and related work

In this section we provide background and related work on the business model for software as a service. We also describe some related work on regression testing and prioritization.

### 2.1. Software as a service model

We present a hypothetical SaaS system to illustrate how customers use SaaS. Suppose our company provides a multi-media application implemented and delivered as a SaaS (shown in Fig. 1). In this application, users can upload and share videos and music,