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.

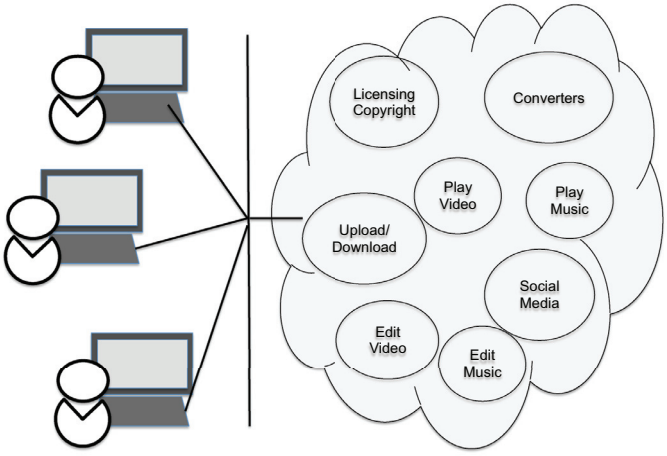

Fig. 1. Hypothetical Multi-Media SaaS. The services (on right) are all interconnected. The users (on left) choose which services they want to pay for through an interface. They can add and remove services dynamically.

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,

as well as edit and play either videos or music for their own personal use, or share them interactively with friends via social media. Copyrights can also be validated and files can be converted.

There are individual services that will perform the uploads and downloads of the media files, services to convert the media to alternative formats, services to stream the media online, a licensing and copyright service, editing services, and a social media service to allow friends to rate the quality of the video and music. Each of these services can be subscribed to (i.e. *rented* alone or in combination with others). For instance, some customers may just want to use the storage and individual online players for the media, while some may want conversion tools as well, and others will also want to edit. Finally, some may be interested in the social media, but do not care about conversion.

In this example we have different services for editing video and music, however in practice these could be joined together within a single subscription. We might also have different services for different converters. The flexibility of providing customization for users is one of the advantages of SaaS. The unique combination of services that individual users can create grows combinatorially with the number of services. For instance, if we have ten services and a user can choose to include (or not include) any combination of these, then there are 2^{10} different ways a user can combine services within this SaaS. A challenge is that different combinations of services may lead to failures. For instance, *editing a video* using an individual service may work without any problems, but a fault may be triggered while *editing a video* in collaboration mode due to an interaction fault between these services. There has been a large body of work on this type of fault in traditional configurable systems (Cohen et al., 1997; Qu et al., 2008; Garvin and Cohen, 2011; Kuhn et al., 2004) showing that failures may only appear when specific combinations of features are used together.

In traditional enterprise systems, customers purchase software and the system is customized and tested for individual customer builds and then installed on-site. Companies pay a one-time licensing fee. As each modification to the system is made, a patch is tested and then released to that customer, for their particular build. Customers have the option of accepting a new build, and different customers will have different versions of the software. In this environment, the risk of a single failure impacting many customers is reduced. However, the maintenance and installation costs for the company are high. SaaS, on the other hand, has the potential to reduce these costs. Herrick (2009) describes cost savings (measured as increased productivity and improved operational efficiency) and Hudli et al. (2009) describe the benefits of a SaaS deployment model in the health care industry. Chen and Sorenson present one way to assess quality using a Quality Assurance Party (QAP) to validate the quality of services against the customer requirements (Chen and Sorenson, 2008), and Choudhary (2007) measures SaaS utility based quality where the customer expectations are met by the rich set of services. Banerjee et al. (2010), present the first study on reliability analysis applied to SaaS applications to extract aspects that should be part of service level agreements, while Mietzner et al. (2009) and Sengupta and Roychoudhury (2011) provide a way to apply variability modeling from software product lines (one type of highly configurable software system) to the SaaS environment. Lynch et al. (2013) present a tool set for testing an IBM SaaS application. They provide a hierarchical model that includes the Build Acceptance Tests. They mention the need for frequent testing cycles such as monthly major iterations, weekly minor iterations, and expedited releases on-the-fly for any issues customers experience, however, they don't provide solutions for achieving this goal. In this paper we focus on ways to reduce the cost of regression testing SaaS applications given their large variability space and frequent maintenance updates, by ordering BATs to find faults sooner.

2.2. Testing in service oriented architectures

Bozkurt et al. (2012) survey the state of the art in testing service oriented architecture (SOA) systems. Their focus is on surveying the problem of testing services in general. Most of the literature focuses on the unique challenges such as a lack of source code, identifying when a service has changed, and the need for interoperability of services. While all of these techniques are relevant to SaaS as well, they do not directly address the SaaS business model and test process, which differs slightly from the open use of services in SOA. In SaaS, the services are all maintained and distributed in a controlled manner by the owner of the SaaS. Although implemented as services, some of the challenges, such as knowing when a service changes are not as important since the developer controls this variable. However, other issues such as interoperability, are just as important in SaaS as in SOA in general. One open issue that Bozkurt et al. (2012) point out is the lack of real-world case studies. We aim to address that issue in this work.

2.3. Regression testing and prioritization

Regression testing is the test process that is executed each time software is modified either for feature updates, bug fixes or other types of maintenance (Rothermel et al., 1996; Rothermel and Harrold, 1997; Elbaum et al., 2002; Srikanth et al., 2005). Regression testing has been shown to account for more than 50% of maintenance costs in a software project, therefore there has been a large body of work focusing on ways to improve this process. Much of the research on regression testing has focused on cost reduction such as test case selection (Rothermel et al., 1996; Rothermel and Harrold, 1997) which aims to select the appropriate (yet small) set of test cases for the modified software version. Some example strategies include selecting a random set of tests, selecting all tests, or selecting a set that executes all code possibly affected by changes (safe technique) (Rothermel and Harrold, 1997). Other work examines ways to *prioritize* test cases so the tests which are run early are those which are more likely to find faults (Elbaum et al., 2002; Srikanth et al., 2005; Qu et al., 2007; 2008; Srikanth et al., 2009).

In traditional regression testing, we have a program P_i and a test suite T_i. When P_i is modified to version P_{i+1}, we will create a new test suite T_{i+1} which contains a subset of test cases from T_i. We may also add additional test cases (test suite augmentation) to test new functionality to P_{i+1} (Xu and Rothermel, 2009). If we view prioritization from a SaaS perspective, we say that our SaaS system is P_i and our modified SaaS system is P_{i+1}. A new P_{i+1} is created anytime a change is made to *any* of the services in the system. For instance, if we modify only a small part of the copyright service of the SaaS system described, then we need to retest the entire system, since some of our customers may combine this service with others (that were not modified). Test case selection will determine which test cases in T_i are relevant (based on our selection criteria) to P_{i+1}. Test case prioritization, on the other hand, will only concern itself with the order in which tests in T_{i+1} are applied to program P_{i+1}. In this work, we do not address test case selection, but instead focus only on prioritization of the already selected BAT tests. We assume all tests which have been selected must be eventually run.

Prioritization often uses information from the test results of T_i when applied to P_{i+1}, such as the code coverage, or fault detection, since research has shown that we can use the prior version's test results as a predictor for our modified version (Qu et al., 2008; Elbaum et al., 2002; Srikanth et al., 2009). We utilize this intuition in this paper as well. We do not consider code coverage (due to the black box nature of our testing), but use history of field failures to order our tests.

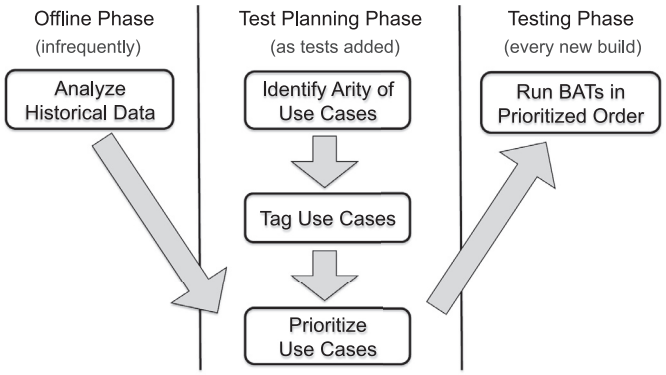

Fig. 2. Proposed prioritization process for BAT.

The most common (and most effective) algorithmic approaches to test case prioritization have been shown to be greedy (Zhang et al., 2013; Li et al., 2007). The *additional* greedy prioritization technique orders tests one by one, selecting the next test case which has the most *new* coverage at each step. The *total* greedy prioritization technique, on the other hand, simply selects the test case at each step that covers the most coverage elements (ignoring what has already been covered in prior tests). With respect to code coverage, the additional greedy technique has been shown to be the most effective technique in reducing the time to failure detection. Nardo et al. (2013) present an industrial case study on regression testing in industry that shows the additional code coverage metric is best in practice as well. In recent work Elbaum et al. (2014) discuss prioritization for fast environments such as those used in continuous integration testing at Google. As new code is added to the central build repository, which happens in a fast-paced cycle (sometimes 10s of lines are changed each minute), the tester must select and then prioritize tests. Elbaum et al. use a *history window* to select tests for prioritization, running those tests first, that found failures in the past. We also use this notion of prior failures in our work, however, we do not define a window, and our test cycles are longer.

3. Prioritization approaches

In this section we discuss a practitioner focused test prioritization approach for the BAT process. We begin by presenting a high level process. We then present several algorithms based on historical failure data. Our high level process is shown in Fig. 2. First we *Analyze Historical Data*, then we *Identify Arity of Use Cases*, and *Tag Use Cases* to provide meta-data. Next we *Prioritize Use Cases* which will then become the ordered BATs which are executed during the final testing phase. We describe each step in more detail next.

1. **Analyze historical field failures.** The first step is to obtain data about field failures seen during the history of the running SaaS application and analyze them to identify the failure-prone services as well as to determine complexity of the use cases with respect to service-to-service interactions. This step can be performed once, or aggregated over time from a reporting database. For each failure, the responsible services (or combinations of services) are determined based on the user reports and developer fixes, then this information is combined for the entire historical period. For instance if we look at five failures from the bug reporting database, and find that four of these involved the *edit music* service and two involved both *edit music* and *social media* we would record four failures for the single service *edit music*, two failures for the single service *social media*. We can use the arity data of each fault as well to guide our prioritization.
2. **Identify arity of use cases.** During the test-planning phase when use cases are identified for testing, the tester should assess if the use case can be executed within a single service or does the execution involve interaction of more than one service. Using our example from earlier, we can consider the use case - *edit video* and *edit video within social media*. The core *edit video* use case can be performed different ways. The user can simply edit the video within their own web server, or they can share the video using the social media service and edit it with others. In the first situation, the use case involves execution within a single service. However, the second case has to utilize two service end points (edit video and social media). In this step testers will identify the service to service interactions that are executed for the different use cases.
3. **Tag use cases.** Using the information from the prior step, the testers can tag each use case with its service interactions. This includes both the names of each service exercised if available, as well as the number of services involved in executing the use case. Minimally, the number of services should be included. This can be performed with minimal effort during test planning.
4. **Prioritize use cases.** This step uses historical data analysis and the tagged use cases to determine the best order for running tests. Prioritization can be performed either on the number of services involved in a use case, or by the most fault prone use cases first.

We illustrate three different approaches for prioritization next. We utilize a small example throughout for illustration purposes. Suppose we have the SaaS system presented in Section 2.1. Fig. 3(a) shows our analysis of six failures during the historical time window (F1-F6). This table is the result of step one of our process and is based on information about which services are required to reproduce and then fix each failure. The second column (labeled Services) shows which services are associated with each failure. The number in parentheses gives the arity of each. Five of our ten services appear here (the rest appear to be less critical for failures). Using this data we can see that *play video* appears in five failures, *edit music* in four failures, and *upload/download* in two failures. Both *social media* and *edit video* are each involved in one failure.

We can also see from this data that the largest number of failures (3) seem to depend on use cases that exercise two services together. The next highest occurrence (2) are in those use cases that exercise 3 services together. We only see one failure that was due to a single service use case. This leads us to our first prioritization heuristic.

3.1. Prioritization using arity of service interaction

Our first approach to prioritization makes the assumption that the arity of service interaction is an important factor in ordering test cases and that the historical data will help to indicate which arity of service interactions is the most likely to lead to failures. The regression testing literature suggests that techniques which combine features, such as combinatorial interaction testing, are effective at finding faults (Cohen et al., 1997; Qu et al., 2008). And in studies that have looked at the arity of interactions, most have a low order of interaction (2–6) (Kuhn et al., 2004). Our hypothesis is that within SaaS applications, this notion of interactions will hold as well. We have implemented an interaction based scheme and show this as Algorithm 1.

In this algorithm we first separate cases by the arity of services into sets (line #2). We then sort these sets by the desired SortOrder (line #3). Then based on the sorted order we select without replacement, tests from each of the test sets randomly. The

Failure	Services (Arity)
F1	*PV* (1)
F2	*PV*, *SM* (2)
F3	*EM*, *UD* (2)
F4	*PV*, *EM* (2)
F5	*UD*, *PV*, *EM* (3)
F6	*EM*, *EV*, *PV* (3)

(a) Historical Failures

Service	Weight
UD	0.15
PV	0.39
EM	0.31
All Others	0.15

(b) Weights

No	Test Case	Services (Arity)
1	uploads music	*UD* (1)
2	edits music	*EM* (1)
3	uploads music to social media	*UD*, *SM* (2)
4	plays video on social media	*PV*, *SM* (2)
5	downloads music from social media, then edits it	*UD*, *SM*, *EM* (3)
6	download, convert, and play video	*UD*, *C*, *PV* (3)

(c) New Build Acceptance Tests

Legend
Social Media (*SM*) Upload/Download (*UD*) Play Video (*PV*) Edit Music (*EM*) Convert (*C*) Edit Video (*EV*)

Fig. 3. Example of Field Failure Data (a), Build Acceptance Tests (b) and Resulting Weights (c).

Algorithm 1 Prioritization using service arity

```
1: Set(SortOrder)
2: Separate(TestSuite) by Arity of services into TestSets
3: Sort(TestSets,SortOrder)
4: for all Arity using SortOrder do
5:    while TestSet_arity not ∅ do
6:       Randomly select TestCase
7:    end while
8: end for
```

SortOrder should be based on our historical results. For instance, suppose we have the set of BAT tests shown in Fig. 3(c). We have six BATs. For each, we show which services are executed during the test case. Since our historical data (from Fig. 3(a)) tells us that the order of testing should be two-services first, followed by three-services, followed by single services, we begin with tests #3 and #4, selecting one of them at random for the first test. We continue to tests #5 and #6, again selecting randomly. Finally, we order the single service tests (#1 and #2). One possible order from our algorithm for this set of BATs is {4,3,5,6,2,1}.

3.2. Prioritization using historically failing services

Using only the arity of service interactions, does not consider the specific services that contribute to faults and may waste effort testing services early that have rarely been seen to fail in prior testing. For instance, in our example the service *convert* was not associated with any of the prior failures, yet it would be tested before some test cases (Test case #6) involving failing services in the historical data, if we used the first algorithm. Our second algorithm weights services based on their importance with respect to failures. We define importance as how often a service contributes to a failure in historical data. If we return to our example, we see the services which contribute to more than 80% of the failures are *upload/download, play video*, and *edit music* (85%) we use these as individual services and group all others into an *Other* category. This category will include all other services including those that may not appear in any current failures (e.g. *convert*).

We next derive a weighting based on the total number of services involved in these failures using the following formula:

$$service_i weight = \frac{\text{count of service}_i}{\text{total service occurrences}}$$

where count of service is the number of times a service has been involved in a failure and the total service occurrences is the count of services in our failure analysis. For instance, the count for *upload/download* is two and there are a total of 13 occurrences for all services, so it has a weight of 0.15. The weights are shown in Fig. 3(b). We can see that the service which is involved in the largest number of failures (*play video*) has the highest weight (0.39). The *Others* category has a weight of 0.15, but when using this in our algorithms, we will normalize it by the number of services in this category (in this case two). Each service in the *Others* category, therefore, will have a weight of 0.075 individually. Note, the BATs may include services that were not seen in prior failures (e.g. *convert*) and may also not include all services that were seen in prior failures (e.g. *edit video*). Anything that does not have its own weight (such as *social media* and *convert*) fall into the *Others* category at this stage.

Algorithm 2 shows the pseudocode for the weighted approach. On line #1 we assign weights to each service based on the historical failure data. Once we compute weights we then mark the services within each test case by its weight (line #2) and then sort in descending order by either the Maximum weight or the Average weight (line #3). We allow for two variants of this algorithm (and will evaluate them in our study).

We treat the services in the other group as a special case when computing the Average weights to avoid biasing towards tests with lower arity. Any service in that group has a weight of zero and is not included in denominator for the average. In this algorithm, if the ordering is not unique then we break ties in the following manner. If we are using the maximum weight, then we continuously select the next highest weight in the test case until either the tie is broken or there are no more weights. If this is still not unique then we use the arity of the test case (biasing towards test cases that exercise more services), and finally if this is still not unique then we break the ties randomly. In the case of the average weight (line #13) we only use the arity of services followed by

Table 1
Detailed weights for BATs from example in Fig. 3.

No.	Use Cases	Arity	$Weight_1$	$Weight_2$	$Weight_3$	Max_1	Avg
1	uploads music	1	0.15	–	–	0.15	0.15
2	edits music	1	0.31	–	–	0.31	0.31
3	uploads music to social media	2	0.075	0.15	–	0.15	0.15
4	plays video on social media	2	0.075	0.39	–	0.39	0.39
5	downloads music from social media then edits it	3	0.075	0.15	0.31	0.31	0.23
6	download, convert, and play video	3	0.15	0.39	0.075	0.39	0.27

Algorithm 2 Prioritization using service weight

```
1: GetServiceWeights(Historical Failure Data)
2: AssignWeights(TestSuite, ServiceWeights)
3: Sort(TestSuite, Maximum (or Average) weight of individual services in test case)
4: if Sort order not unique then
5:   if Maximum then
6:     i ← 2
7:     while Not Unique Sort Order and i ≤ NumServices in largest arity test case do
8:       Break ties by ith Maximum
9:       i++
10:    end while
11:  else
12:    Break ties by Selecting Highest Arity Test Case
13:  end if
14:  if Sort order not unique then
15:    Break ties randomly
16:  end if
17: end if
```

random to break the ties (the average already considers additional weighted services).

Let's return to our example one more time. Table 1 shows the weights for each service in the BATs and also shows the maximum weight as well as the average weight. If we use the maximum weighting, we begin by finding the test case with the highest maximum weight. In this case it is either test case #4 or #6. We next break the tie, by looking at the second maximum weight, and choose test case #6 (it has a service with a weight of 0.15). So our first two tests are #6 and #4. We then choose between test case #2 and #5. Test case #5 is chosen since it has a second maximum weight of 0.15. Next we choose tests #1 and #3, both with a weight of 0.15. Test #3 has a second service from the other category with a weight of 0.02 so we select that first. Our final order would be: {6,4,5,2,3,1}.

For the average weighting, the services in the *other* category are not averaged (i.e. they have a working weight of 0 in this algorithm and do not become part of the denominator). For the average, Test #4 is chosen first, followed by test #2, and then #6 and #5. We then choose between Test #1 and #3 and choose #3 since it has the higher arity. This gives us the order {4,2,6,5,3,1}.

3.3. Prioritization using additional greedy

For this last approach we utilize an additional greedy algorithm, since this has been shown to be the most effective with respect to code coverage in prioritization of traditional systems. The idea is to cover as many services as possible (greedily), selecting the next test as one that covers the largest number of previously untested services. The pseudocode is shown in Algorithm 3. We initialize the set of services seen in S to be empty (line #1). In lines #2 through #4 we sort by arity as we did in Algorithm 1. We begin with a test

Algorithm 3 Prioritization using additional greedy

```
1: S ← ∅ {set of services seen}
2: Set(SortOrder)
3: Separate (TestSuite) by Arity of services into TestSets
4: Sort (TestSets,SortOrder)
5: Randomly select TestCase i ∈ TestSet_highestArity
6: S ← S ⋃ services in i
7: while |S| < # of services do
8:   AssignTags(TestSuite,NumberNewServices) {tag is number of services not in S}
9:   Select Max(TestCase,Tags)
10:  if tie then
11:    Select Highest Arity Test Case
12:  end if
13:  S ← S ⋃ services in i
14: end while
```

case of highest arity since we have seen no services yet (line #5) then add those services to S (line #6). Then we tag the remaining test cases with the number of services that are new (line #8) and choose the highest (line #9). If there is a tie we choose the highest arity since it will not significantly elongate the process to cover more services (line #10). Then we add these new services to S (line #13) and continue until S contains all of the services possible. Using the additional greedy approach may lead to test cases which do not contribute anymore to an improvement in coverage. These remaining tests can be sorted randomly or by continually applying the Greedy Algorithm without replacement (e.g. resetting the covered set to be empty each time we cover all services) until we have run out of test cases. We evaluate both approaches in our study.

Let us revisit our running example. Suppose we choose test case #5 first (this is one of our highest arity test cases). Now $S = (social\ media, upload/download, edit\ music)$. In the next round test case #6 has a tag of 2 because we have not seen services *play video* or *convert*, test case #3 has a tag of 1 with the new service *play video*, and the rest have a tag of 0. We choose test case #6 since it has the highest tag. Now S contains all of the services and the remaining test cases do not contribute new coverage. To use the continual Greedy algorithm, we begin again by randomly choosing a test case of highest arity, perhaps test case #4. Now all three remaining test cases have a tag of 1. We break the tie by choosing the test case of the highest arity which is test case #3. Now $S = (social\ media, play\ video, upload/download)$. In the next round we choose test case #2 since we have not seen service *edit music* yet. Now there are no more services left to be found, so we move on again. There is only one remaining test case. It goes last. Our final ordering for this algorithm is {5,6,4,3,2,1}.

4. Empirical study

We conducted an empirical study to evaluate the prioritization techniques described in Section 3 for the Build Acceptance Test

process. We have identified four research questions. The first research question asks about the distribution of the history of reported field failures to see if they are associated with individual services or with combinations of services, and to understand if there is any pattern that can be utilized. The second research question asks if the history of field failures provides heuristics that predict what will happen in a future build test process. The third question asks about the importance of the actual order of testing during build verification. Our last questions evaluates the impact of the cost of failures (measured as fault severity) on our results. Our research questions are:

RQ1: What is the arity and distribution of service composition interactions in field data failures?

RQ2: How similar is the distribution of service composition interactions in BAT, with that seen in the field data?

RQ3: How does the prioritization order impact the time to find build verification failures?

RQ4: How does the consideration of severity impact the results of the different prioritization orders?

4.1. Software under test

We applied our scheme to an enterprise level IBM cloud application that provides communication and collaboration systems for knowledge workers in enterprise organizations[1]. The application is deployed as a Software as a Service for thousands of enterprise customers around the world. The users of this system have mostly been customers for over a year. The application has over a million lines of code, divided among the various services and needs to be reliable and functional twenty four hours, seven days per week, and has to be validated in all working environments including the mobile domain. The software team is distributed around the world which further adds to the testing challenges.

Since the application is hosted as a service the teams have to make system updates frequently (at least once a month) to remain competitive. Each release comprises both new features and failure updates. With the frequent releases, it requires software teams to perform black box testing on the new features along with regression testing with more than 1000 test cases per cycle. A test cycle may last more than 24 hours given that there is manual setup work performed by a limited size team, and each test suite has to run on multiple servers (development, different build, production, etc.). The services in this system are not independent. They can interact, therefore any change to the system requires re-testing across services.

Based on the experience of the first author in an industrial team and as a consultant, it has been a common practice for her test teams to prioritize test cases randomly for execution. We will use this as our base order. In this particular product, however, the teams order tests by considering execution of the simple use cases first. The simple use cases are the ones that usually involve executing the test within a single service, followed by complex use cases that involve integration of two or more services

4.2. Study process

We began by manually analyzing the data history of failures from the field that were reported and subsequently fixed from the production application for more than six months, and used this data as our historical baseline. First, for each failure we recorded the specific services that were involved in reproducing the failure and implementing the fix, as well as the count of services involved (arity) and the fault severity. The failure and count data was used to create our weightings and to determine which arity ordering should be used.

We then collected an instance of approximately 300 Build Acceptance Tests from a current release, that represent a new regression cycle run on one of the development servers. These BATs represent the most important use cases (successful completion of these tests are essential for a build to pass an acceptance criteria (Lynch et al., 2013)), therefore they will implicitly contain tests that are expected to find failures. All of these tests must eventually be run during the testing cycle at least once. Although the number of services in the test cases differs, we cannot directly measure the difference in cost since there are other factors that also impact the running time of an individual BAT. This includes non-quantifiable attributes such as the skill and familiarity with the system under test of each of the testers on the team. However, we do see a difference in the severity of the failures, which impacts overall testing cost as well. Prior industrial studies have used an exponential increase in the cost of finding and fixing a failure with respect to its severity (Srikanth et al., 2014). There are four levels of severity that are normally used (defined by the test team). The two most critical (level 3 and 4) mean that the system cannot be used at all (level 4), or that there are workarounds, and the product can be used with the workaround, however this is a serious failure that impacts core functionality (level 3). Levels 1 and 2 represent failures that either have a workaround which has little impact on overall system functionality, or whose impact is relatively minor.

The BATs consist of a mixture of single service, two service and three service tests (51%, 36% and 13%, respectively). Each test is a use case that utilizes all services in combination (i.e. a test for a two-service test, explicitly utilizes both services in that test). Each was marked as either pass or fail based on the results of testing and the severity was recorded. We used this to build a fault matrix for further analysis.

We then re-ordered these BATs using our various heuristics, and then used the fault matrix to compare how each of the orders performs. For each prioritization order, we repeated this process 30 times due to the stochastic tie-breaking aspects of the algorithms.

4.3. Independent Variable

The independent variable is the prioritization order. We use the three algorithms (and their variants) presented in Section 3. The orders that we use (in the same order that they are presented in our graphs) are listed below. The first set of orders utilizes the arity of services in a BAT.

- **Random**. This is a completely random order. We randomly select (without replacement) a test case and run that. This order represents the current state-of-the-art practice for this particular enterprise system. The next five orders, utilize Algorithm 1.
- **123**. For this order, we run all of the single service BATs first, and then run all of the two-service BATs, and end with the three service BATs. This order represents the practice in some organizations where the simple tests are run first.
- **231**. This order selects all of those use cases that involve two services first. It then selects those with three services and finally ends with single-service use cases. This is the order suggested by the results of RQ1 (using our historical data).
- **321**. In this order we use descending order of the complexity of the use cases, running those with three services first.
- **3plus21**. This is a variant of the last two orders. We were curious to see what would happen if we put the two-service and three-service use cases into a single bucket for selection and ran all of those first. We then run the single-service use cases.

[1] Due to proprietary reasons we are unable to provide the exact name or full details of this system or its acceptance tests.

Table 2
Build acceptance test failures.

No.	Services	Fault Detected	Severity
1	UD	0	0
2	EM	0	0
3	UD, SM	1	3
4	PV, SM	0	0
5	UD, SM, EM	1	1
6	UD, C, PV	1	1

- **132**. This version is a variant of the single-service first order. We begin with all single-service use cases, followed by those which involve three services, and end with those involving two services.
- **213**. This order runs all of the two-service uses cases first and follows that with the single-service use cases, ending with the three-service use cases.
- **312**. This order runs the three-service use cases first, followed by the single-service use cases and ends with the two-service use cases.

The next two prioritization orders, utilize the weights based on which services are associated with more failures (Algorithm 2). The differences between these are seen in lines #3 and #5-12.

- **MaxWeight**. For this order, we use the maximum weight of the use case to drive the prioritization. If this is not unique then we iteratively break ties using the next highest weight and if this is still not unique, then use the highest arity use case first and finally use random selection if the tests are still not unique.
- **AvgWeight**. In this order, we use the average weight of the test case (where zero weights are removed). If there is a tie, we use the arity (highest to lowest) to break the tie, followed by a random selection.

The last two prioritization orders, utilize an additional greedy algorithm.

- **GreedyAdd**. This order is a pure additional greedy approach. The additional greedy algorithm is applied, then the remaining test cases are randomized.
- **GreeyCont**. In the last order, we apply additional greedy continually to the test cases without replacement. Once one pass is complete and we have covered all services, we apply the same algorithm to the remaining test cases.

4.4. Dependent Variables

For RQ1 and RQ2 we study the percent of failures based on the overall failures observed. For RQ3 we use the Average Percentage of Faults Detected (APFD) (Elbaum et al., 2002). APFD is a common metric used in prioritization. It indicates how rapidly a prioritized test suite detects faults during the execution of the full test suite. In essence it is the area under the curve for fault detection if you consider the `y-axis` to contain the failures and the `x-axis` to contain the test cases. APFD is calculated by Eq. (1).

$$APFD = 1 - \frac{\sum_1^m F}{mn} + \frac{1}{2n} \tag{1}$$

where m is the number of faults, F is the position of the first test case in the prioritization order that detects fault m, and n is the total number of test cases. The APFD values range from 0 to 1. A higher APFD score indicates a higher rate of fault detection. Let us consider our running example with the results of the BATs as shown in Table 2. If we run the test cases in the order shown then the first fault occurs at position 3, the second at position 5, and third at position 6 which leads to a sum of 14. Plugging that into equation where $m = 3$ because we have 3 faults and $n = 6$ because we have 6 test cases (1) we get $1 - \frac{14}{3*6} + \frac{1}{2*6} = 0.31$. This is a relatively low APFD score. Now if we prioritize by the **321** order, one possible order would be: {5, 6, 3, 4, 2, 1}. The first fault occurs at position 1, the second at position 2, and third at position 3 which leads to a sum of 6. Plugging that into Eq. (1) we get $1 - \frac{6}{3*6} + \frac{1}{2*6} = 0.75$. This APFD score is higher which shows prioritizing may provide faster fault detection compared to running the test in a random order.

For RQ4 we utilize the cost-cognizant APFD metric (APFDc) given by Elbaum et al. (2001). This variant of the APFD incorporates two types of costs to the APFD. First it includes the cost of running each test case. Second it adds a severity cost to each failure. It rewards higher cost/severity test cases more if detected early in the testing process. In this study, since we only measure fault severity, we call the metric APFDsv. Elbaum et al. suggest two different ways to incorporate severity (linear and exponential), but do not provide empirical evidence for which one should be used in practice. We use an exponential increase as in Srikanth et al. (2014), (each increment of severity increases the weight by a power of 2). The reasoning is that the real time and cost required to fix higher severity faults in industry grows in a non-linear fashion.

We define this alternative APFD metric as follows:

$$APFDsv = \frac{\sum_1^m (\mathbf{2^{s-1}}(n - F + \frac{1}{2}))}{n \sum_1^m \mathbf{2^{s-1}}} \tag{2}$$

where s is the severity measure of test case m and the weight is 2^{s-1}. A higher severity equates to more serious fault. Again consider the running example. In the case of a **321** prioritization {5, 6, 3, 4, 2, 1} the first fault results in $1(6 - 1 + \frac{1}{2})$ in the numerator since fault 1 occurs at position 1 with a severity weight of $2^{1-1} = 1$. Continuing for the rest of the faults we obtain a sum of 24. So the APFDsv value is $\frac{24}{6(4+1+1)} = 0.67$. Now let us consider another prioritization. One possible ordering for 231 is {3, 4, 5, 6, 2, 1}. The APFDsv value for this prioritization ends up being $\frac{28}{6(4+1+1)} = 0.78$ which is a higher. Recall that the original APFD for **321** was 0.75, and the APFD for **231** is 0.64. So the APFD value is higher for **321**, but the APFDsv value is higher for **231**. This means that we can find the faults faster using **321**, but if we want to find the higher severity faults first we should use **231**.

4.5. Threats to validity

We have selected a single software system and used only a window of historical data, and prioritized a single BAT round. However, this is a large, highly subscribed SaaS enterprise system, and we believe it is representative of many SaaS systems that exist in practice. We also think that the window of history was sufficient to indicate what we are likely to see in alternative windows. With respect to the selection of the BATs, we acknowledge that a different BAT might produce slightly different results, but we did not bias this by selecting a specific one (rather we randomly chose the BATs to target). In the BAT data, we do not have the failure fixes, so we assume that each failure is unique. Based on the fact that the tests themselves are carefully created to cover a broad range of behaviors, we believe that this is a legitimate approximation. Finally, we used a post-hoc analysis for prioritization and did not physically re-run the tests in each prioritization order, but rather used the results of a single concrete run (in the form of a fault matrix) and then re-ordered these and assumed determinism in outcomes, because 30 runs for 12 different orders would have been impossible in a production environment. We believe that the results are still valid. This type of post-analysis is common in the regression testing literature. We also note that the author who ran the BATs

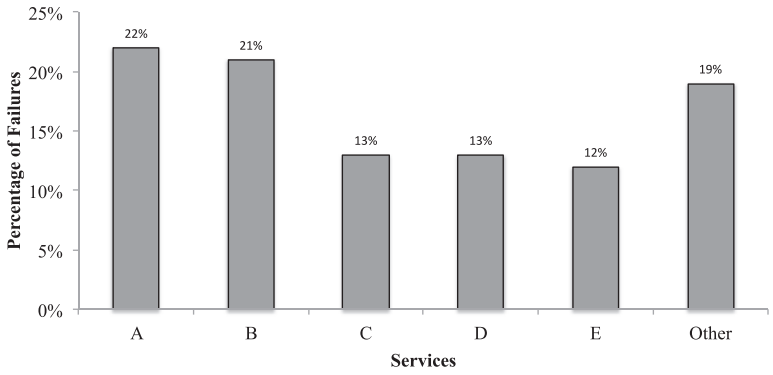

Fig. 4. Field failures: percent of failures by service.

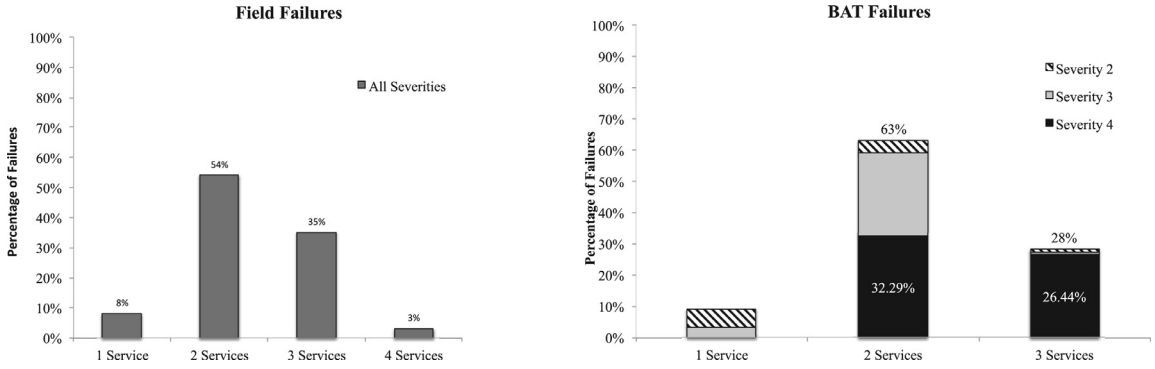

Fig. 5. Field failures (left) and BAT failures (right): percent by number of interacting services.

was not involved in the prioritization; that was done by another author who was not familiar with the system under test.

5. Results

In this section we present the results of our study organized by research question.

5.1. RQ1: distribution of field failures

To answer RQ1 we begin by looking at Fig. 4. This shows by percentage, the distribution of the services that were involved with each of the field failures. As we can see, Service *A* (leftmost) was the service that was included in the largest number of failures (22%). The next highest service *B* was in 21%. Three other services, *C*, *D* and *E* each were involved in 12 or 13 percent of the failures. The rest of the services were each involved in less than 5% of the failures and therefore grouped into the *other* category.

We next examine the numbers (or arity) of services that are involved in each failure. The data for the field failures is shown on the left side of Fig. 5. We will examine the right side in RQ2. More than one half of the failures (54%) involved 2 interacting services. 35% involved 3 interacting services, and 3% involved 4, meaning that 90% of the failures require a use case that exercises more than a single service to be exposed. Less than 10% of our failures (only 8%) are manifested by a single service. To obtain the arity, we looked at both the failure report and the description of the fix. This leads us to conclude that use cases which exercise combinations of services are more likely to lead to a failure than those which exercise a single service alone. The most common failure scenario for our historical data is when two services are used together, followed by three services, four services and last for single services. We also examined the severity of the field failures. The lowest severity (level 1) accounts for almost 40% of the failures, while levels 2 and 3 together make up about 56% of the failures (23% for level 2 and 33% for level 3). The most severe failures account for around 4% of what is seen in the field.

Summary of RQ1: There appears to be a small number of services that are most commonly associated with failures, so it may be possible to use the failure prone services in field data for future predictions and testing. We also observe that most failures are associated with more than a single service, and that the largest number are associated with two services.

5.2. RQ2: Distribution of BAT Failures

In this question we look at the distribution of the build acceptance test failures to see if the history of field failures predict a best practice for ordering those tests. For this data we turn to the right side of Fig. 5. We note that the test cases run involved only up to three services, because BAT includes only critical use cases. We again see that a large percent (91%) of failures are observed with tests that exercise two or more services together, and again the two service failures account for the largest proportion (63%). Those which exercise three services are responsible for finding 28% of the failures. Although we don't have the fixes for these failures, each is unique with respect to the BAT that found it, which means that the failure correlates to the arity of the test case which discovered it. We don't show the data for the service distribution, since the BATs are already biased toward the most likely to fail services (whereas the original field data allows for any possible use case). However, we do note that again the top two services were *A* and *B*.

We also examine the severity of the BAT failures. We do not see any level 1 failures, but this is not surprising since these tests are meant to find the most severe faults. The percentages of severity are seen in Fig. 5. We see many more of level 4 severity failures in the BATs and all of these are found within the 2 and 3-service

Table 3
APFD Over 30 runs by prioritization order.

	Rand	123	231	321	3plus21	132	213	312	MxWgt	AvWgt	GrAdd	GrCont
Avg.	0.50	0.27	**0.70**	**0.72**	**0.71**	0.30	0.57	0.43	**0.71**	0.67	0.50	0.67
StDev.	0.02	0.00	0.01	0.01	0.01	0.00	0.01	0.00	0.00	0.00	0.02	0.00
Min.	0.43	0.26	0.69	0.71	0.69	0.29	0.56	0.42	0.71	0.67	0.48	0.66
Max.	0.57	0.28	0.71	0.73	0.72	0.31	0.58	0.44	0.72	0.68	0.55	0.67

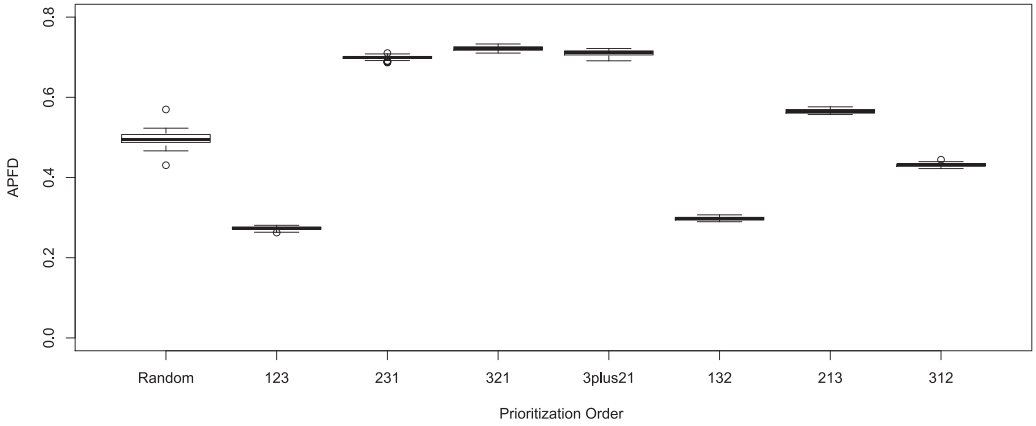

Fig. 6. APFD of prioritization orders based on arity for 30 Runs.

test cases. Together the level 4 failures account for more than 50% of those seen.

Summary of RQ2: We conclude that the BAT data follows the field failure data with respect to the distribution of failures. As in the field failure data, we see more failures in the use cases which exercise more than one service, with the most common BAT data failure category that of the 2-service use cases. We see more higher level severity failures in the BATs but that is expected since this phase of testing aims to remove the critical failures from the system.

5.3. RQ3: prioritization results

We now examine the various prioritization orders. Table 3 shows the APFD values for the different prioritization orders. The first column is the Random order, with an average APFD of 0.50. We then show the other orders, highlighting all that have an average APFD higher than or equal to 0.70. We see that the APFD for testing single services followed by 2 and then 3 services (123) is the worst ordering (average APFD of 0.30) suggesting that a naive approach to testing (e.g. testing the simplest use cases first) could result in running out of time before all faults are found. The highest APFD is seen in the 321 order (0.72), which tests all three-service tests first, followed by all 2-service tests, followed by single service tests. The average of 3plus21 and the MaxWeight are close with 0.71 and 0.72 respectively. The AvgWeight does not perform as well as MaxWeight which is not surprising. Given that the BATs are biased using a subset of already important tests, the average makes less sense for weighting. The order that is suggested by historical data, 231 improves significantly over either 123 or Random, however it is not quite as high as the 321 order. It is possible this is due to the fact that some of the 3-service failure use cases are actually finding 2-service failures (we do not know the fixes for these). It is also interesting to see that the order 312 does very poorly (0.43) – even worse than Random. We looked at this more closely and see that although the 3-service test cases have a very high initial slope and find many faults, there are few of these compared with the one and two-service failures. Since the one-service tests have the highest number (51%) but the lowest fault detection, this causes the area to drop significantly. The order 213 is higher because there are more 2-service tests. But it is only slightly better than Random (0.53). The GreedyAdd order is very close to random. This is logical since only the first few test cases are ordered, and the rest (which account for the majority) are randomized. GreedyCont however performs better (0.67). By the nature of GreedyCont the 3-service test will be favored in general. So it is reasonable that GreedyCont performed significantly better then GreedyAdd. Figs. 6 and Fig. 7 show boxplots for each of the orders for all 30 runs of our prioritization. In Fig. 6 we show only those prioritization orders related to arity. In Fig. 7 we examine how the MaxWeight, AvgWeight, GreedyAdd, and GreedyCont compare with the best results (and Random) from Fig. 6. We can see that there is little variation (other than Random) in the APFD except for GreedyAdd. GreedyAdd has more variance since only the first few test cases are selected and the reminder is randomized. Even so, within GreedyAdd and Random there is a relatively tight APFD value.

Summary of RQ3: The order that the Build Acceptance Tests are run is significant and should be considered. If a naive approach is used to run tests that exercise fewer services first, this may mean that all tests cannot be run in the time given and failures will be missed. A reasonable approach is to randomize the tests, however it appears that history information is useful in improving the time to fault detection and should be used when available. A possible alternative is to use the in-house defects when field failure data is missing. We plan to explore this as future work.

5.4. RQ4: severity results

In our last research question we consider the impact severity has on our prioritization orders. Table 4 shows the APFDsv values for the various prioritization techniques. Fig. 8 shows a boxplot of the APFD*sv* for the prioritization orders based on service interaction, and Fig. 9 shows the best interaction orders in addition to the weighted and greedy orderings.

The prioritization orders with the highest APFDsv values are MaxWeight and 321 with values above 0.76. Also scoring high are

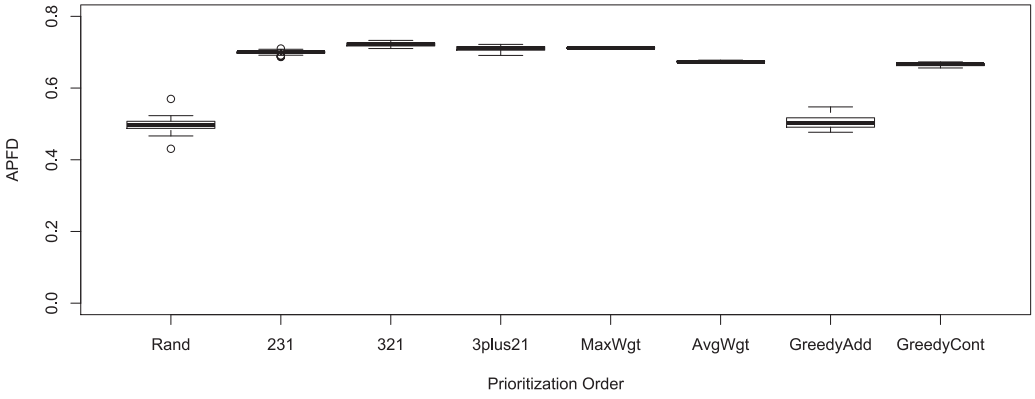

Fig. 7. APFD of best arity based prioritization orders vs. weight-based orders and greedy orders for 30 Runs.

Table 4
APFD*sv* Over 30 runs by prioritization order.

	Rand	123	231	321	3plus21	132	213	312	MxWgt	AvWgt	GrAdd	GrCont
Avg.	0.50	0.24	**0.71**	**0.76**	**0.74**	0.29	0.53	0.46	**0.77**	**0.75**	0.51	**0.70**
StDev.	0.02	0.01	0.00	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.21	0.00
Min.	0.43	0.23	0.70	0.75	0.72	0.28	0.52	0.46	0.77	0.74	0.47	0.69
Max.	0.56	0.24	0.72	0.77	0.75	0.30	0.54	0.47	0.77	0.75	0.56	0.71

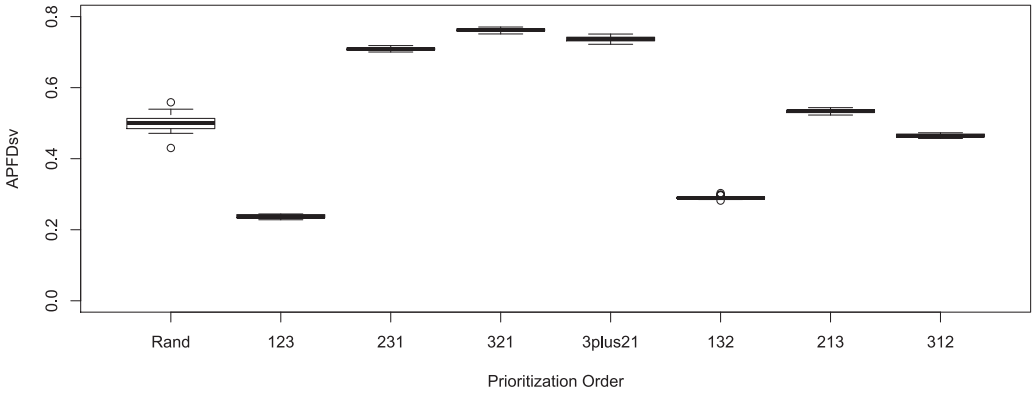

Fig. 8. APFD*sv* of prioritization orders based on arity for 30 Runs.

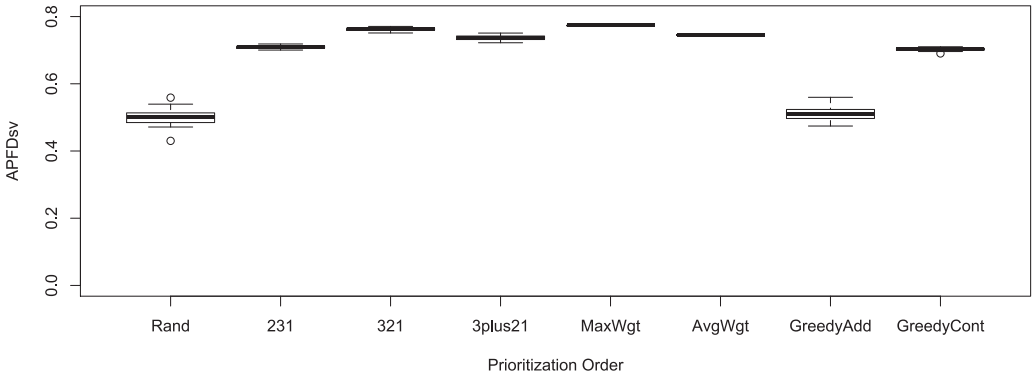

Fig. 9. APFD*sv* of best arity based prioritization orders vs. weight-based orders and greedy orders for 30 Runs.

AvgWeight and 3plus21 (above 0.74). Then there is a drop-off to the next order (231) which has an APFD*sv* score of 0.71.

The results are comparable to APFD with some slight differences. In Table 5 we rank the prioritizations by average APFD and APFD*sv* value as well as group the orders into *tiers*. Since the APFD and APFD*sv* values are not directly comparable using absolute values, we use the tiers to compare which orders perform best. We can see that the orders 321 and MaxWeight are in the first tier of both metrics. These are the two orders that consistently score the best and should be used in this practice. 3plus21 also scores well in the first tier of APFD and the second of APFD*sv*. We can also see that the order 231 performs well in the APFD, but only mediocre in APFD*sv*. Similarly, AvWeight performs well in APFD*sv*, but mediocre in APFD. Greedycont ranks in the middle in both metrics. We also see that the six orders that rank worst, all below 0.60, are in the same order for both metrics.

Table 5
Ranking of prioritizations by Avg.

	APFD	Rank	APFDsv	
	321	1	MaxWeight	
	MaxWeight	2	321	
	3plus21	3	AvWeight	
≥ 0.70	231	4	3plus21	≥ 0.74
	AvWeight	5	231	
≥ 0.67	Greedycont	6	Greedycont	≥ 0.70
< .60	213	7	213	< .60
	Greedyrand	8	Greedyrand	
	rand	9	rand	
	312	10	312	
	132	11	132	
	123	12	123	

Summary of RQ4: Severity has an impact on prioritization orders with the MaxWeight performing the best in this scenario. However the two best prioritization orders are the same (in inverted order) in each of the metrics. These are 321 and MaxWeight. 3plus21 also scores well in both. 231 scores better in APFD, and AvWeight scores better in APFD*sv*.

5.5. Discussion and lessons learned

We have learned some valuable lessons from this empirical study. First, we have seen that the order of Build Acceptance Testing is important, and the simplest approach of testing single services tests followed by two service tests, etc., and which is most likely to be used by test teams, may provide the longest time to fault detection. We have also seen that it is possible to get some improvement over the worst case, without using the history by ordering the tests randomly, but our rate of fault detection is still relatively slow. While we expected the prioritization that uses the most information from historical data (MaxWeight) to always provide the best result, it did not. For APFD the simple ordering (321) that uses only arity (and is easier to perform) gave us the best results. When we consider severity, however, MaxWeight performed best. We also see that combining the higher arity tests (3plus21) performed well in both metrics, suggesting again that the history is only a guideline and perhaps is less important than the complexity of the use case. We cannot say if the higher arity use case results will generalize, or if this result is due to historical data without further studies, but we think that this indicates an interesting possibility which merits some investigation.

In this study, we did not focus on the time difference of running individual test cases, since in this particular BAT environment, adding additional services to the use case did not have a significant impact on time since there were other (external factors) impacting test time that we could not control such as the tester performing the tests. We acknowledge that in some systems, we may be able to measure this accurately and it can be considered. We leave the study of a time-based prioritization in SaaS as future work.

We now summarize some lessons learned that BAT practitioners can take from this work.

1. **Order matters**. When running BATs the order that the tests are run can significantly change the time to failure detection and may impact our ability to complete important tests within the allotted testing time.
2. **Study the historical field failure data** and use this as a predictor for ordering BATs.
3. **In absence of information** it may be possible to use in-house defects.
4. **When a deterministic order** is needed, then use the higher arity use cases first.

6. Conclusions and future work

In this paper we have presented an empirical study on prioritization for the build acceptance test process of a large industrial SaaS application. We base our prioritization on historical data from field failures. The results of our study show that two or three interacting services tend to be involved in field failures and that this is mirrored in the Build Acceptance Testing process. We also see that (1) the order of running BATs can significantly impact efficiency of testing (2) we can use historical field failure data to drive BAT prioritization (3) a simple random heuristic works better than a fixed order in case of no historical information, however, we suggest using in-house defects as a heuristic in the absence of field failures and plan to study this in future work. In future work, we plan to apply our technique to additional SaaS systems, and to examine a longer time period of testing. We also plan to evaluate systems where running times of individual test cases are different.

Acknowledgments

We thank the reviewers for their detailed and helpful comments on this work. This research was supported in part by IBM and the National Science Foundation awards CCF-1161767 and CNS-1205472.

References

Banerjee, S., Srikanth, H., Cukic, B., 2010. Log-based reliability analysis of software as a service (saas). In: Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on, pp. 239–248.
Beizer, B., 1990. Software Testing Techniques. International Thomson Computer Press.
Bozkurt, M., Harman, M., Hassoun, Y., 2012. Testing and verification in service-oriented architecture: a survey. Software Testing, Verification Reliability 23 (4), 261–313.
Chen, X., Sorenson, P., 2008. A QoS-based service acquisition model for is services. In: International Workshop on Software Quality, pp. 41–46.
Choudhary, V., 2007. Software as a service: Implications for investment in software development. In: System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on. 209a–209a
Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C., 1997. The AETG system: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23 (7), 437–444.
Elbaum, S., Malishevsky, A., Rothermel, G., 2001. Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of the 23rd International Conference on Software Engineering. ICSE, pp. 329–338.
Elbaum, S., Malishevsky, A.G., Rothermel, G., 2002. Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering 28 (2), 159–182.
Elbaum, S., Rothermel, G., Penix, J., 2014. Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 235–245.
Garvin, B.J., Cohen, M.B., 2011. Feature interaction faults revisited: An exploratory study. In: International Symposium on Software Reliability Engineering, ISSRE, pp. 90–99.
Goth, G., 2008. Software-as-a-service: The spark that will change software engineering? IEEE Distributed Systems Online 9 (7). 3–3
Herrick, D., 2009. Google this! using google apps for collaboration and productivity. In: ACM SIGUCCS Fall Conference on User Services.
Hudli, A., Shivaradhya, B., Hudli, R., 2009. Level-4 SaaS applications for heathcare industry. In: 2nd Bangalore Annual Compute Conference.
Kuhn, D., Wallace, D.R., Gallo, A.M., 2004. Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering 30 (6), 418–421.
Li, Z., Harman, M., Hierons, R., 2007. Search algorithms for regression test case prioritization. IEEE Transactions on Software Engineering 33 (4), 225–237.
Lynch, M., Cerqueus, T., Thorpe, C., 2013. Testing a cloud application: IBM smartcloud inotes: Methodologies and tools. In: Proceedings of the 2013 International Workshop on Testing the Cloud, TTC 2013, pp. 13–17.
Mietzner, R., Metzger, A., Leymann, F., Pohl, K., 2009. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, PESOS, pp. 18–25.
Nardo, D.D., Alshahwan, N., Briand, L., Labiche, Y., 2013. Coverage-based test case prioritisation: an industrial case study. In: Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 302–311.

Qu, X., Cohen, M.B., Rothermel, G., 2008. Configuration-aware regression testing: an empirical study of sampling and prioritization. In: International Symposium on Software Testing and Analysis, pp. 75–85.
Qu, X., Cohen, M.B., Woolf, K.M., 2007. Combinatorial interaction regression testing: a study of test case generation and prioritization. In: International Conference on Software Maintenance, pp. 255–264.
Rothermel, G., Harrold, M.J., 1997. A safe, efficient regression test selection technique. ACM Transactions on Software Engineering Methodology 6 (2), 173–210.
Rothermel, G., Harrold, M.J., Dedhia, J., 1996. Analyzing regression test selection techniques. IEEE Transactions on Software Engineering 22 (8), 529–551.
Sengupta, B., Roychoudhury, A., 2011. Engineering multi-tenant software-as-a-service systems. In: Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems, PESOS, pp. 15–21.
Srikanth, H., Banerjee, S., Williams, L., Osborne, J., 2014. Towards the prioritization of system test cases. Software Testing, Verification Reliability 24 (4), 320–337.
Srikanth, H., Cohen, M., 2011. Regression testing in software as a service: an industrial case study. In: International Conference on Software Maintenance (ICSM), pp. 372–381.
Srikanth, H., Cohen, M.B., QU, X., 2009. Reducing field failures in system configurable software: cost-based prioritization. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 61–70.
Srikanth, H., Williams, L., Osborne, J., 2005. System test case prioritization of new and regression test cases. In: Empirical Software Engineering. 2005 International Symposium on, pp. 64–73.
Xu, Z., Rothermel, G., 2009. Directed test suite augmentation. In: Software Engineering Conference, 2009. APSEC'09. Asia-Pacific, IEEE, pp. 406–413.
Zhang, L., Hao, D., Zhang, L., Rothermel, G., Mei, H., 2013. Bridging the gap between the total and additional test-case prioritization strategies. In: Software Engineering (ICSE), 2013 35th International Conference on, pp. 192–201.

Hema Srikanth is part of the product strategy team for IBM's analytics portfolio. She has held many leadership roles within IBM for the past 8 years. She has over 10 patent grants and over 30 patent files in the areas of cloud computing and collaborative solutions. Her research focuses on improving development and test methodologies for applications on cloud; addressing technological challenges associated with big data, and improving end user quality for cloud applications.

Mikaela Cashman is a Ph.D. student in the Laboratory for Empirically Based Software Quality Research and Development, ESQuaReD at the University of Nebraska-Lincoln. She completed her B.S. in Mathematics and Computer Science at Coe College, Iowa. Her research interests are in test case prioritization, applying AI techniques to software engineering, and in using software testing techniques to understand and model biological systems.

Myra B. Cohen is a Susan J. Rosowski Associate Professor at the University of Nebraska-Lincoln where she has been a member of the Laboratory for Empirically Based Software Quality Research and Development, ESQuaReD since 2004. She is the recipient of a US National Science Foundation Early Career award and an Air Force Office of Scientific Research Young Investigator award. Her research interests are in software testing of highly configurable software, software product lines, graphical user interfaces, and in search-based software engineering.