Experience Report: An Empirical Study of API Failures in OpenStack Cloud Environments

Pooya Musavi, Bram Adams and Foutse Khomh
Polytechnique Montreal, Quebec, Canada
{pooya.musavi,bram.adams,foutse.khomh}@polymtl.ca

***Abstract*—Stories about service outages in cloud environments have been making the headlines recently. In many cases, the reliability of cloud infrastructure Application Programming Interfaces (APIs) were at fault. Hence, understanding the factors affecting the reliability of these APIs is important to improve the availability of cloud services. In this study, we mined bugs of 25 modules within the 5 most important OpenStack APIs to understand API failures and characteristics. Our results show that in OpenStack, only one third of all API-related changes are due to fixing failures, with 7% of all fixes even changing the API interface, potentially breaking clients. Through qualitative analysis of 230 sampled API failures we observed that the majority of API related failures are due to small programming faults. Fortunately, the subject, message and stack trace as well as reply lag between comments included in these failures' bug reports provide a good indication of the cause of the failure.**

I. Introduction

An Application Programming Interface (API) is a set of public methods [1] that is meant to be used by other software applications. For example, cloud APIs are provided by a cloud platform to enable applications in the cloud to interact with, deploy or manage on the platform [2].

However, too often, developers experience API call failures that threaten the reliability and availability of their cloud apps. Failures of cloud apps generally result in big economic losses as core business activities now rely on them [3]. This was the case in December 24, 2012 when a failure of Amazon web services caused an outage of Netflix cloud services for 19 hours. Hence, understanding factors affecting the reliability of cloud APIs is important to improve the availability of cloud services.

Lu et al. [4] investigated API failures of Amazon EC2 services and classified the causes in three categories of faults: *development*, *physical* and *interaction* faults. There have also been studies that have attempted to detect faults (e.g., [3] [5] [6]) responsible for these failures. To the best of our knowledge, the analysis and characterization of the causes (faults) of these failures have not been studied this far.

In this paper, we perform an empirical study on OpenStack to see how often internal and external APIs changed and how this was an API failure. We consider an API failure as any run-time problem related to the interface or implementation of an API, caused by API designers or implementers, eventually causing an outage of a service. Additionally, we qualitatively analyze the causes of API failures by mining the source code the failures are originating from as well as their fixes. Our empirical analysis focuses on OpenStack, which is a popular open source cloud infrastructure platform.

In this study, we address the following research questions:

(RQ1) How often are APIs changed to fix API failures?

A quantitative study of the 25 most important modules in OpenStack shows that a median of 23% of API changes are related to the API interface, and a median of 7% of such changes are due to the fixing of a failure. Of all the API changes that do not alter the API interface, a median of 24% are due to the fixing of a failure. Our observations demonstrate that one third of changes to the API are due to fixing failures.

(RQ2) What are the most common types of API failures and faults?

Based on our analysis of 230 randomly selected fixes, we classify the causes of the API failures into seven categories including: small programming faults (56%), major programming faults (14%), configuration faults (14%), race conditions (5%), deadlock conditions (5%), improper log message faults (4%) and data format faults (3%). On the other hand, our observation showed that db errors (16%), test errors (15%), network errors (10%), deployment errors (4%) and security errors (4%) are the most frequent symptoms of failures.

(RQ3) What are the bug fixing characteristics of the different fault types?

The small programming faults are fixed by developers with less developer activity in comparison to major programming faults. We also found that there is no significant difference in call distance between major programming faults and configuration faults. We observed that small programming faults do not take less time to fix in comparison to major programming faults. Furthermore, there is no difference in developer experience between small programming, configuration and major programming faults.

(RQ4) What are the main factors explaining the bug fixing process of small programming faults?

We developed a composite model of a Naive Bayesian and Decision Tree classifier that takes into account features such as subject, message, stack trace and reply lag from issue report. We noticed that the subject, message and stack trace information, and the number of developers working on a fix for a failure as well as reply lag are the most important characteristics of failures caused by small programming faults.

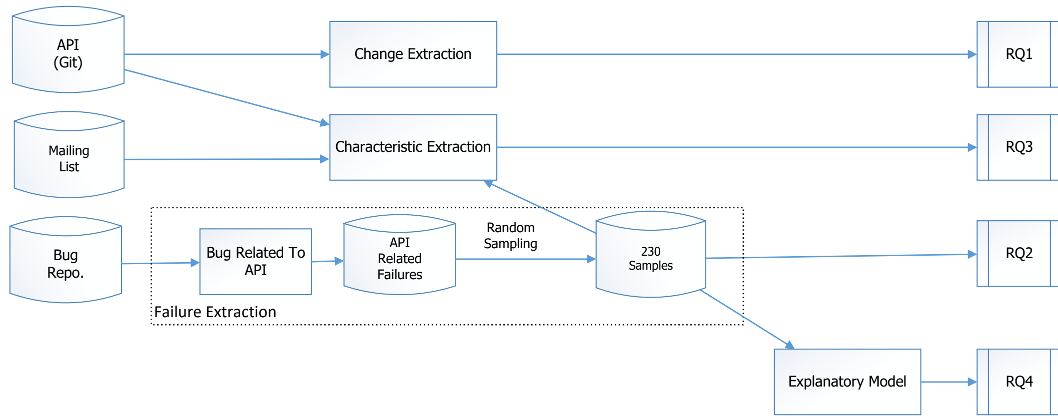

Figure 1: Overview of our approach for answering RQ1, RQ2, RQ3 and RQ4

II. CASE STUDY SETUP

In this section, we describe the studied systems, and present our data extraction and analysis approach. Figure 1 shows our approach to answer the research questions.

A. Studied Systems

OpenStack is an open source cloud infrastructure project launched by Rackspace Hosting and NASA in 2010. It is governed by a consortium of organizations who have developed multiple components that together build a cloud computing platform for ”Infrastructure As A Service”. This means that users can deploy their own OS or applications on top of OpenStack to virtualize resources like storage, networking and computing.

OpenStack hosts its bug repository on launchpad [1]. When a bug is reported related to an API failure, a user would normally put the *stack trace* of the exception into the bug report message. Then, the bug would be triaged by a developer in order to evaluate whether it is valid or not and a priority to that bug would be assigned as well. At the end, when a patch or a fix has been reviewed, a link to the corresponding git commit is added to the corresponding bug report. We selected OpenStack as the case study system based on the following criteria:

Criterion 1: **Accessibility** Since OpenStack is an open source project, its source code, bugs and stack traces are available online [2]. OpenStack is the most popular open source cloud platform, rivaling commercial competitors like Amazon and Microsoft in popularity and feature set.

Criterion 2: **Traceability** The bug repository is linked with the review system through a hyperlink to the Gerrit review environment, and it is also possible to link to the resulting bug fix in the version control system (git). Since we want to do a qualitative study on the files in which an API failure has been fixed in order to understand the causes of the analyzed failures, this well-established linkage is a must for our research.

[1]https://bugs.launchpad.org
[2]https://github.com/openstack

In order to select the most important APIs for our study, we queried Amazon for the most popular books related to OpenStack, resulting in 143 records. We reviewed the top 3 books [11] [12] [13] and concluded that Nova, Swift, Heat, Neutron and Keystone are the five most significant APIs. Books have been considered before in empirical studies on SE [8], especially in cases where popularity, experience or terminology of practitioners are required.

B. Data Extraction

In order to access the bug and source code change data of OpenStack, we mined the official launchpad as well as the data set provided by Gonzalez-Barahona et al. [9]. In launchpad, we manually investigate the cause of the failures (faults), while we use the data set of Gonzalez-Barahona et al. [9] to perform our quantitative study such as exploring the faults' characteristics. This data set has 221671 commits from 2010-05 to 2015-02 in the *scmlog* table. Its most important columns are *revision* (the hash id of the commit), *committer_id* (the id of the person that made the commit), *date* and the *message* (the text that the developer writes at the time of commit). It also has 55044 bug reports starting from 2010-07 to 2015-02. The most important columns of this table are *issue* (the bug number), *type* (the decision status), *summary* (subject of the bug), *description*, *status* (whether it is Fixed Release, Invalid or etc.), *priority* (High, Low or etc.) and *submitted_on* (the date that is reported). Finally, there are 88842 emails starting from 2010-11 to 2015-02. It has two most important columns: *subject* and *message_body*. Since recent data has a lower chance of being fixed than older data, we limited the data to 2015-02, in order to assure that we have more stable resolved issues.

Change Extraction. For RQ1, we focus on the 5 most important API git repositories; Nova, Swift, Heat, Neutron and Keystone. For each API, we fetched the 5 most important modules. The programming language in OpenStack is Python and a module is a file (.py) containing Python definitions and statements. We compared the differences between each pair of consecutive commits of these modules to understand whether any changes related to method signature occur or not, such as

Table I: Characteristics studied

Dimension	Metric	Unit	Description & Rationale
importance	talked_in_mailing_list	BOOLEAN	Whether the developers have talked (mentioned the bug id) about this bug in the mailing list or not. Such bugs likely are more important or complex to fix.
	number_of_times_bug _status_changed	NUMBER	Number of times that the status of a bug has been changed. When a bug status changes frequently it indicates difficulty for developers to make a decision on it.
	severity	NUMBER	Shows how critical a bug is for a project.
	number_of_people_affected	NUMBER	Number of users affected by this bug.
fixing process	developer_experience	NUMBER	The experience of the most recent developer who fixes the bug, based on the number of commits that he has made before the current FIX. More complicated defects might need more experienced developers.
	developer_activity	NUMBER	Number of commits that the developer has done across our whole data set. Simple bugs might not need more active developers.
	number_of_developer _working_on_bug	NUMBER	A bug might require several developers during its life cycle, indicating its difficulty.
	bug_activation_in_days	NUMBER	number of days to close the bugs as fixed.
symptom	subject_message_and_stack_trace	String	The subject, message left by the reporter and exception thrown by the API. This characteristic indicates the symptoms of a failure.
	call_distance	NUMBER	The number of modules existing in an exception. As this number increases, it might be more possible to have errors.
	commenter_experience	NUMBER	The more experienced in leaving comments, based on the number of comments the more helpful a discussion could be, reducing the risk of defects slipping through.
	comment_count	NUMBER	The more comments are posted for an issue, the more risk might be involved.
	comment_length	NUMBER	The number of lines of comments on an issue may indicate that the discussed commit has a high likelihood of introducing a bug.
	reply_lag	NUMBER	The average time in between comments can be related to the risk of a bug. Normally, risky ones get faster replies to comments.
bug fix	code_churn	NUMBER	Size of bug fix.
	ndev	NUMBER	Average number of developers that changed the fixed file before. Different developers modifying the same file might lead to misunderstanding.
	age	NUMBER	Average time (#days) since the last change. More recent changes are more likely to be error-prone.
	nuc	NUMBER	The number of unique changes to modified files. more files have been changed, the more opportunities for defects.

removing or adding parameter or even a deleting method. If yes, we then checked whether the change happened for fixing a bug.

To know whether a commit is fixing a bug, we looked for "*bug*", "*fix*","*defect*" and "*patch*" keywords inside the commit messages. A similar approach to determine defect-fixing changes has been used in other work [19] [20].

Failure Extraction. For this aim, we first use the data set of Gonzalez-Barahona et al. [9] to fetch all fixed bugs for the year 2014. To further understand the causes of these failures ("faults"), we then consider the subject, message and stack traces of the thrown exceptions because they contain symptoms (side effects) of the failure and help understand better the causes. We manually studied some bugs related to APIs and we understood that 90% of bug reports related to API failures contain *api* and *traceback* keywords inside. Hence, we performed a query to search for those bugs containing "*api*" and "*traceback*" within the body of the bug messages. This resulted in 923 reports related to 135 projects.

Because investigating all of these reports is a time consuming task, we performed a statistical sampling with a 95% confidence level and a confidence interval of 5.5% to see how many samples we need to study [14]. As such, we randomly selected 230 samples out of the 923 reports. Through these samples, we distinguished between bugs related to failures in OpenStack APIs or client application programming failures. We were conservative and we studied the developers' and commenters' messages to ensure that a bug is relevant to an OpenStack API failure. We removed any unrelated bug from the list and randomly replaced it by another bug. In launchpad, given the traceability between bug repository, review system and version control system, we tracked each sampled bug's review and fixes to analyze the differences between the version before and after fix. RQ2 and RQ3 concentrate on the failures related to the APIs.

Characteristic Extraction. Table 1 shows the independent metrics and the rationale why we select them to be used in RQ3 and RQ4 to build an explanatory model of small programming faults, which are the most common kind of faults found in our analysis results. The table shows 4 different dimensions of information available during the resolution of a cloud API bug. We used bug, e-mail and source code repositories to extract the characteristics. Amongst these metrics, the code churn, the number of developers working on a bug, ndev, age, nuc and whether a bug is discussed in the mailing list are not in the bug reports, but are extracted from source code and mail repository. The Call_distance represents the number of modules (files) called between the calling module until and API module raising a failure. Our definition for experience is the number of commits that the developer has done before in the control version system (git) before fixing the current bug [10], while developer activity is the total number of commits the developer has done across our whole data set.

C. Explanatory Model

In this section, we describe our approach for constructing our explanatory model in RQ4 from the sample of 230 bugs.

Composite Data Mining Approach. While RQ2 analyzes and classifies failures, RQ3 builds an explanatory model to understand the important characteristics of the bug fixing process of API failures caused by small programming faults as opposed to other faults. Since a Decision Tree classifier

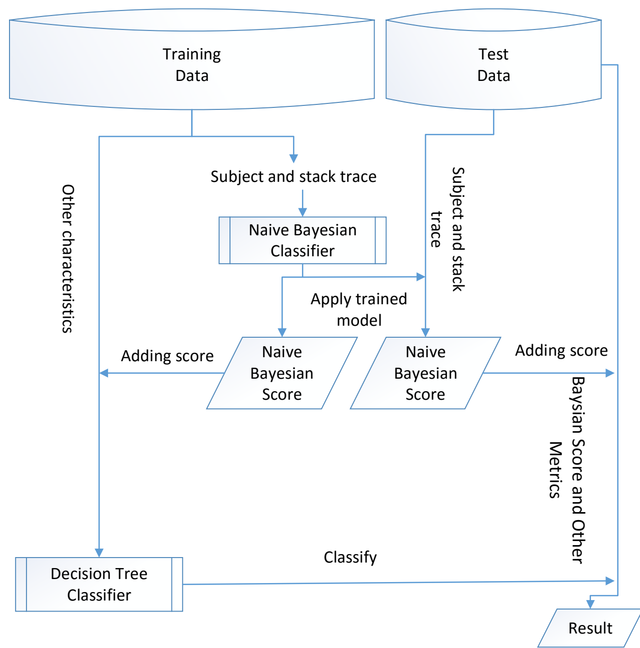

Figure 2: Overview of our approach for answering RQ4

does not have good support for the "String" data type and we want to include textual subject, message and stack trace content into our model, we use a Naive Bayesian classifier to deal with these fields of a bug report.

As shown in Figure 2, we use a composite model, similar to Ibrahim et al. [16], which involves two data mining approaches. First, we apply a Naive Bayesian classifier (as used by spam filters) [29] on the bug subject, message and stack trace content to determine how much this information is relevant to small programming faults. Second, we add the calculated Bayesian score (probability) to the other characteristics of Table 1 as the input to a Decision Tree classifier.

The **Naive Bayesian classifier**. Similar to a spam filter, this classifier takes the subject, message and stack trace from the training corpus. In fact, the Naive Bayesian classifier divides the content into tokens and counts the occurrences of each token. These counts are used to determine the probability of each token to be an indicator of the fault type. Finally, it gives a score indicating whether a whole string is relevant to small programming faults. The closer the score is to 1, the higher the probability that the content will be relevant.

The **Decision Tree classifier**. Our Decision Tree classifier takes the bug subject, message and stack trace score from the Naive Bayesian classifier algorithm as input instead of the original string data, together with the other characteristics discussed before. We use a Decision Tree classifier as a machine learning algorithm, since this classifier offers an explainable model explicitly showing the main factors that affect a fault type, while many of the other machine learning techniques produce black box models that do not describe their classification decisions. We have used the C4.5 algorithm [18] to create our Decision Tree.

Table II: Confusion Matrix

	classified as	
Actual category	Small fault	Not small fault
Small fault	a	b
Not small fault	c	d

Evaluation of the Model. To validate our model, similar to the strategy used by Christian et al. [31], we use an 80-20 split. To this aim, we divide the studied bugs into two parts: the training corpus-containing 80% of the data (randomly selected) and the testing corpus-containing the remaining 20%. The training corpus is used to build the classification model, while the testing corpus is used to test the accuracy of the model. This process is repeated 100 times to get more robust measurements. To this aim, we build a confusion matrix at each iteration to measure the performance of our model. The confusion matrix looks like Table II:

Based on the confusion matrix, we evaluate our explanatory model by the metrics below:

- **Precision (P)**: Proportion of failures correctly classified as small programming faults (a) over all failures classified as small programming faults (a+c), i.e., $p = \frac{a}{a+c}$
- **Recall (R)**: Proportion of failures correctly classified as small programming faults (a) over all failures that are caused by small programming faults (a+b), i.e., $R = \frac{a}{a+b}$
- **F-Measure**: The harmonic mean of precision and recall, i.e., $F = 2.\frac{precision-recall}{precision+recall}$
- **Area Under Curve (AUC)**: The range of AUC is [0,1], with a large value indicating better model performance than random guessing and a value of 0.5 indicates that the classifier is no better than random guessing.

III. CASE STUDY RESULTS

In this section, we describe the results for each research question.

(RQ1) How often are APIs changed to fix API failures?

Motivation. Due to different activities such as re-engineering, refactoring [1] and bug fixing, libraries and frameworks often need to change. In the simplest case, only the implementation of API methods needs to be fixed. Such changes are safe for API clients. However, changes to an API's signature would be problematic for the applications that are consuming them. For example, consider a case in which a client application is calling a public method from an API to accomplish a transaction on a user's account. If this method in the API changes, a failure would be raised by the API noting that this method does not exist within that API, which leads the end-user of that service to suffer from the malfunctioning. In the worst case, the end-user might even make a decision to change his service provider.

Hence, this question aims to understand the rate of API change for fixing failures in a popular cloud platform like OpenStack. In this RQ, we investigate changes of both API method signature and implementation.

Table III: Analysis of the 25 most important modules in OpenStack. TC:Total Commits, MSC: Method Signature Changes, NMSC: Non Method Signature Changes, FB: Fixing Bugs. All percentages are relative to TC.

API	Module Name	TC	MSC		NMSC
			All	FB	FB
Heat (2012-2015)	engine/service.py	469	129 (27%)	60 (12%)	123 (26%)
	engine/resource.py	433	139 (32%)	49 (11%)	79 (18%)
	db/sqlalchemy/api.py	214	15 (7%)	9 (4%)	56 (26%)
	api/openstack/v1/stacks	129	22 (17%)	5 (3%)	21 (16%)
	common/wsgi.py	116	10 (8%)	5 (4%)	37 (31%)
Keystone (2012-2015)	identity/core.py	212	71 (33%)	27 (12%)	55 (25%)
	service.py	143	15 (10%)	12 (8%)	55 (38%)
	assignment/backends/ldap.py	118	24 (20%)	7 (5%)	33 (27%)
	common/utils.py	105	15 (14%)	3 (2%)	38 (36%)
	common/controller.py	114	31 (27%)	15 (13%)	28 (24%)
Nova (2010-2015)	compute/manager.py	2775	843 (30%)	195 (7%)	468 (16%)
	db/sqlalchemy/api.py	1972	194 (9%)	23 (1%)	319 (16%)
	compute/api.py	1867	664 (35%)	146 (7%)	268 (14%)
	api/ec2/cloud.py	776	197 (25%)	23 (2%)	120 (15%)
	virt/libvirt/driver.py	1803	472 (26%)	194 (10%)	343 (19%)
Neutron (2012-2015)	db/db_base_plugin_v2.py	204	58 (28%)	20 (9%)	76 (37%)
	plugins/openvswitch/agent/ ovs_neutron_agent.py	197	47 (23%)	29 (14%)	74 (37%)
	agent/l3/agent.py	129	34 (26%)	7 (5%)	34 (26%)
	db/l3_db.py	120	32 (32%)	15 (12%)	47 (39%)
	plugins/ml2/drivers/openvswi/ agent/ovs neutron agent.py	95	25 (26%)	11 (11%)	27 (28%)
Swift (2010-2015)	common/utils.py	345	53 (15%)	13 (15%)	46 (13%)
	obj/server.py	212	25 (11%)	9 (4%)	28 (13%)
	proxy/controllers/obj.py	165	22 (13%)	12 (7%)	33 (20%)
	container/server.py	140	11 (7%)	5 (3%)	19 (13%)
	common/db.py	111	22 (19%)	8 (7%)	9 (8%)
Median			23%	7%	24%
Total Median				31%	

Approach. We developed a Groovy [3] script and used the JGit [4] library to calculate the following metrics for the 25 most important modules of the 5 most popular OpenStack APIs: Total Number Commits, Number of Method Signature Changes and Number of Method Signature Changes For Bug Fixing. We then distinguished between changes affecting the API signature and others.

Findings. **In total, 31% (one third) of all commits fixes API faliures**. **We found that a median value of 23% of the sampled API changes is devoted to signature changes**. Table 3 shows that how this percentage fluctuates from 2% (keystone/common/utils.py) to 15% (swift/common/utils.py). A median value of 7% of all commits changes the method signature during the resolution of an API failure. On the other hand, the remaining 77% commits not changing method signature have a median value of 24% for fixing failures as well.

Our finding that 7% of API commits changes an API's signature to fix a failure confirms the result of Wu et al. [33], who analyzed and classified API changes and usages from 22 framework releases in Apache and Eclipse ecosystems. Wu et al. [33] found a median value of 11% for the changes of API method signature, which they considered such changes as rare. Our finding that one third of API changes are related to (the fixing of) API failures prompts us to the next research question.

[3]http://www.groovy-lang.org/

[4]http://www.eclipse.org/jgit/

(RQ2) What are the most common types of API failures and faults?

Motivation. This RQ analyzes what API failures are the most common, as well as what the most popular causes ("faults") of these failures are. This information is useful for developers and clients alike to better understand the failures that they are experiencing as well as to have an indication of the possible fault responsible for the failures.

Approach. We conduct a qualitative study to manually evaluate the bug reports as well as bug fixes of API failures during the year 2014 in OpenStack projects. We adopted a "Card Sorting" technique to classify the symptoms and causes of the failures in 230 randomly selected reports (see section 2.2). The "Card Sorting" technique [22] [23] is an approach that systematically derives structured information from qualitative data. This technique is commonly used in empirical software engineering when qualitative analysis and taxonomies are needed. For example, Bacchelli et al. [24] used this technique to analyze code review comments, while Hemmati et al. [25] used it to study survey discussions [27]. We used Google Keep [5] as a tool for this purpose, since it allows to search through cards and can export them into a text file.

To that end, we first read each bug report's stack trace to analyze the reported symptoms, i.e., the exception or main error (e.g., "DbError Exception"). Second, the first author analyzed the corresponding bug fix changes. For example, when a developer added try-catch, he added this kind of changes as a new card "*adding try-catch*". We also added the

[5]http://keep.google.com

symptom of each bug in the same card as we classified its fault (cause). After analyzing all sampled defect reports, we started clustering the cards into related topics. We did one clustering for the symptoms, and one for the faults.

As initial inspiration for the fault clusters, we used the IEEE standard classification for software anomalies [6] and Orthogonal Defect Classification (ODC) [7]. However, we soon realized that these classifications are too coarse-grained. For instance, we found a race condition as a main cause of a failure, which is a much more detailed category than the IEEE Standard's "logic fault" and "Timing/Serialization" category in ODC. Hence, we started to classify the faults in as much detail as possible.

Findings. **We obtained almost 30 categories of API faults, which we could group into 7 categories**. However, we noticed that in many cases, a bug fix only touches a couple of lines in one file, making simple logic changes like inverting logical conditions, fixing typos in variable names or adding a new catch exception. Since such changes only touched one file, and the changes were minor, we created one category for this and called it "*small programming faults*". To clarify more, Figure 3 shows a sample of this fault type, where the developer changes the default value of a variable to another value.

Contrary to small programming faults, we observed that many fixes involved several files and/or multiple parts of files are touched by the developers. We created a category for this and we called it "*major programming faults*". We include method signature changes (interface faults) into this group as well. Figure 4 shows a sample of this kind of faults where a developer changes the method signature by adding more parameters. While enumerating the samples for major programming faults, we separately counted the statistics for method signature changes to see whether there exists any aligned statistics with our previous result for RQ1 on the five most important APIs.

"*Configuration faults*" is another category of causes of API failures, where a wrong value is set in a configuration file. Figure 5 shows a bug that is fixed by a correction to the value in a configuration file.

As mentioned earlier, we faced "*race condition faults*" where a variable is accessed concurrently by multiple threads. Also, similar to this fault, we faced "*deadlock condition faults*" where a process or thread locks an object and other processes or threads are not able to access this object. Since these kinds of faults are difficult to identify, we were conservative and we read the commit messages to make sure what the cause of the failure is exactly about. Figure 6 and 8 show these categories.

"*Data format faults*" cover situations in which an incorrect data type was given to a method or the data was not in a correct format. Figure 8 shows how a developer fixes defects related to a data encoding issue.

[6]http://standards.ieee.org/findstds/standard/1044-2009.html

[7]http://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2.pdf

Table IV: Prevalence of API fault types

Fault Type	Percentage
Small programming fault	56%
Configuration fault	14%
Major programming fault	14%
Race condition	5%
Deadlock condition	4%
Improper log message	4%
Data format fault	3%

Table V: Prevalence of API failure types

Symptom	Percentage
db error	16%
test error	15%
network error	10%
deployment error	4%
security error	4%
Other(vm error, volume error, task error, etc.)	56%

"*Improper log message*" corresponds to cases where a wrong message or inappropriate log is sent to the users. This makes problem diagnostics and resolution difficult for users. Figure 9 shows that the developer tries to give a more appropriate message by modifying data in the output text string. Table 4 summarizes the different categories obtained.

As Table 4 shows, **there are 7 major categories of fault type in our findings: Small programming faults, configuration faults, major programming faults, race condition, deadlock condition, improper log message and data format fault. Small programming faults are the most common type of API faults, followed by major programming and configuration faults**.

In Table 4, we can see the proportion of each category. It is clear that almost half of the causes are related to small programming faults. In other words, the majority of API failures were caused by a trivial programming mistake. The next most common type of fault are major programming faults, which are 4 times less common, but are caused by more serious programming issues. Configuration faults typically are easier to fix, depending on the understanding of the cloud configuration.

Surprisingly, the number of method signature changes in our sample data (part of major programming faults) is about 6%, which is aligned with the median number of method signature changes found in RQ1 for the 25 important modules, i.e., 7%.

The most common API failures are database and test failures. Table 5 shows the different types of failures and their percentages.

(RQ3) What are the bug fixing characteristics of the different fault types?

Motivation. Understanding how different API fault types are being fixed helps developers and managers be well prepared for future API failures. In this RQ, we analyze possible differences in characteristics of different fault types.

Approach. Using the tool that we developed to answer RQ1, we find the commits that fix the failures studied in RQ2. The characteristics that we investigate are described in Table 1 and are obtained from bug reports, bug fix commits

```
class Rbd(Image):

    SUPPORTS_CLONE = True

    def __init__(self, instance=None, disk_name=None, path=None, **kwargs):
        super(Rbd, self).__init__("block", "rbd", is_block_dev=True)
        if path:
            try:
                self.rbd_name = path.split('/')[1]
            except IndexError:
                raise exception.InvalidDevicePath(path=path)
class Rbd(Image):

    SUPPORTS_CLONE = True

    def __init__(self, instance=None, disk_name=None, path=None, **kwargs):
        super(Rbd, self).__init__("block", "rbd", is_block_dev=False)
        if path:
```

Figure 3: Bug No.1362221-Small programming fault.

```
    def dvr_vmarp_table_update(self, context, port_id, action):

        """Notify the L3 agent of VM ARP table changes.

        Provide the details of the VM ARP to the L3 agent when
        a Nova instance gets created or deleted.
        """
        port_dict = self._core_plugin._get_port(context, port_id)
    def dvr_vmarp_table_update(self, context, mac_address, ip_address,
                               subnet_id, router_id, action):
        """Notify the L3 agent of VM ARP table changes.

        Provide the details of the VM ARP to the L3 agent when
        a Nova instance gets created or deleted.
        """
```

Figure 4: Bug No.1362985-Major programming fault.

```
                                                  Patch Set 1 2
+10 ... skipped 41 common lines ... +10
pyOpenSSL>=0.11

# Required by openstack.common libraries
six>=1.7.0

oslo.db>=0.2.0  # Apache-2.0
oslo.i18n>=0.1.0  # Apache-2.0
oslo.messaging>=1.4.0.0a3

retrying>=1.2.2 # Apache-2.0
osprofiler>=0.3.0
```

Figure 5: Bug No.1354500-Configuration fault.

```
stop_instance and the _sync_power_states periodic task to try and fix a
race between stopping the instance via the API where the task_state is
set to powering-off, and the periodic task seeing the instance
power_state as shutdown in _sync_instance_power_state and calling the
stop API again, at which point the task_state is already None from the
```

Figure 6: Bug No.1339235-Race condition.

and developer emails. Overall, we are interested in all characteristics related to the resolution of faults, i.e., symptoms, the importance of the failure, the fixing process and the eventual fix. Because small programming, major programming and configuration faults have more occurrences than the other faults, we focus only on the differences of these three fault types.

Findings. **There is a significant difference in the activity of developers fixing small and major programming faults**. It seems that small programming faults require less active developers than major programming faults. A Mann-Whitney u test [26], i.e., a non-parametric statistical test to validate the null-hypothesis "There is no significant difference in the activity of developers fixing small and major programming faults", we obtain a p-value of 0.001 (alpha value of 0.01) which rejects the null-hypothesis, accepting the alternative hypothesis that there is a significant difference between both

```
@_retry_on_deadlock
def service_update(context, service_id, values):
    session = get_session()
    with session.begin():
        service_ref = _service_get(context, service_id,
                                   with_compute_node=False, session=session)
        service_ref.update(values)

    return service_ref
```

Figure 7: Bug No.1370191-Deadlock condition

```
    def __init__(self, **kwargs):
        meter_id = '%s+%s' % (kwargs['resource_id'], kwargs['name'])
        # meter_id is of type Unicode but base64.encodestring() only accepts
        # strings. See bug #1333177
        meter_id = base64.encodestring(meter_id.encode('utf-8'))
        kwargs['meter_id'] = meter_id
        super(Meter, self).__init__(**kwargs)
```

Figure 8: Bug No.1333177-Data format fault

```
    except ValueError as vex:
        LOG.error(_('Failed to parse %(dir)s/%(name)s') % {
                  'dir': env_dir, 'name': env_name})
        LOG.exception(vex)
    except IOError as ioex:
        LOG.error(_('Failed to read %(dir)s/%(name)s') % {
                  'dir': env_dir, 'name': env_name})
        LOG.exception(ioex)

    except ValueError as vex:
        LOG.error(_('Failed to parse %(file_path)s') % {
                  'file_path': file_path})
        LOG.exception(vex)
    except IOError as ioex:
        LOG.error(_('Failed to read %(file_path)s') % {
                  'file_path': file_path})
        LOG.exception(ioex)
```

Figure 9: Bug No.1272114-Improper log message

distributions.

A related null hypothesis is about the amount of developer experience of the developer attempting to fix an API failure. In particular, we believed that developers with low experience fix small programming faults. Therefore, we created a null-hypothesis "There is no significant differences between developer experience in small programming and major programming faults". A Mann-Whitney U test with the p-value = 0.22 was not able to reject. This implies that **we found no proof of significant difference in terms of experience of developers who fix small faults and developers who fix major faults**. Figure 10 and Figure 11 show the boxplot of experience and developer activity metrics.

As Figure 12 shows, we understand that there would be a significant difference in the call distance between configuration and major programming faults. However, a Mann-Whitney U test with alpha value of 0.01 between major and configuration faults is not able to reject the null-hypothesis (p-value=0.05), hence, **there exists no significant differences in terms of call distance**. This indicates that major programming faults have no longer call distance in comparison to configuration faults. The Mann-Whitney U statistical test did not show any significant difference between small programming faults and configuration faults either. This implies that any fault type can occur in an API with any number of modules inside and there is no correlation between this number and the occurrence of a specific fault type.

As Figure 13 shows, **the code churn of major pro-**

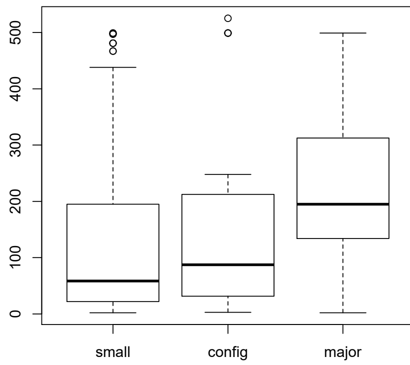

Figure 10: Developer activity (number of commits in the whole data set).

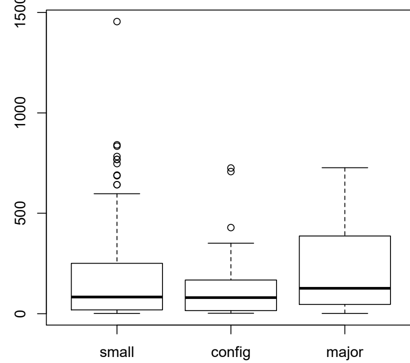

Figure 11: Developer experience (number of commits before fixing current fault).

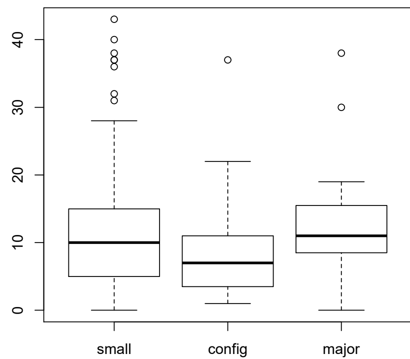

Figure 12: Call distance (number of modules in the stack trace).

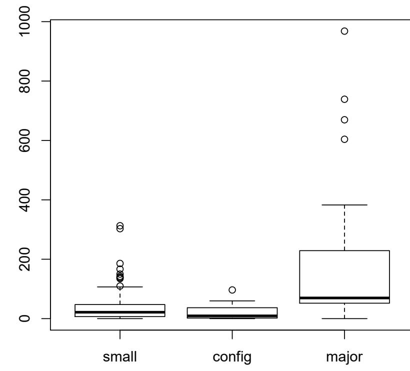

Figure 13: Code churn (size of fix).

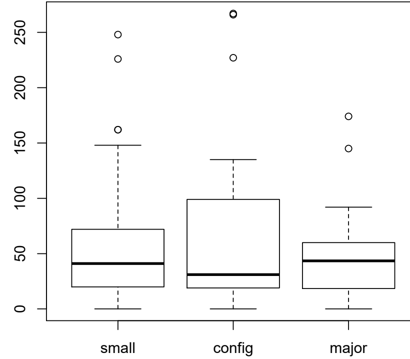

Figure 14: Bug activation in number of days.

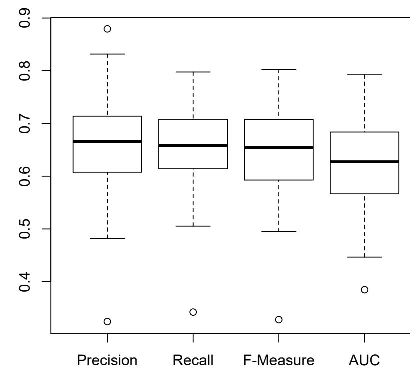

Figure 15: Performance measurements for 100 iterations.

gramming faults is significantly higher than the other two categories. This is expected, since in our fault type classifications, we considered bug fixes involving larger code changes as well as method signature changes as major faults.

Surprisingly, small programming faults do not take significantly less time to be fixed than major programming faults. According to Figure 14, we see no significant differences between different kinds of faults, specifically the small and major programming faults. One conjecture might be that, despite the small sizes of bug fixes, small programming faults can still be difficult to be detected and diagnosed.

(RQ4) What are the main factors explaining the bug fixing process of small programming faults?

Motivation. Until now, we have found that almost half of the causes of the failures are related to small programming faults. We have gathered various characteristics of these faults showing that despite requiring a simple fix, they might actually take as long to be resolved as major programming faults. Now we are interested to know the major factors in the bug fixing process of such failures. If one would be able to predict for a given reported bug, either right after the bug is reported or during the bug fixing process (when more data becomes available about the bug fixing process), that a bug likely is

Table VI: Decision tree top-node analysis score of metrics after 100 iterations.

Dimension	Metric	Score
importance	talked_in_mailing_list	55
	number_of_times_bug _status_changed	0
	severity	34
	number_of_people_affected	61
fixing process	developer_experience	0
	developer_activity	42
	number_of_developer _working_on_bug	193
	bug_activation_in_days	35
symptom	subject_message_and_stack_trace	99
	call_distance	35
	commenter_experience	62
	comment_count	30
	comment_length	60
	reply_lag	124
bug fix	code_churn	53
	ndev	40
	age	64
	nuc	46

due to a small programming fault, bug fixing could be planned differently than in case a deadlock or major programming fault is to be expected.

Unfortunately, we do not have sufficient manually classified failures (see RQ2) to build and evaluate a prediction model. Hence, we focus on an explanatory model.

Approach. First, we randomly select 80% of all 230 samples and we train the Naive Bayesian classifier based on it. Since an imbalanced training set creates suboptimal results, we examined two approaches to balance the data. We did **re-weighting** and **re-sampling** (under-sampling and over-sampling) by using Weka [17]. For re-weighting, we used AdaBoostM1 algorithm. We compared the output of both balancing techniques. The best results were obtained when we did re-sampling using under-sampling. In that case, our classifier was trained well and outperformed the other cases.

Second, we give the remaining 20%, i.e., test set, to the Naive Bayesian trained model in order to generate the Naive Bayesian score. Then we give the training set with the Naive Bayesian score and the other characteristics to a Decision Tree learner. Finally, we apply this model to the testing set. We repeat these iterations 100 times. This type of validation has been used in several studies, such as Pinzger et al [32].

In order to identify the most impactful variables in the model, we use top-node analysis [28]. For each of our 100 iterations, we parse the Decision Tree and we create a hash table of metrics for the nodes on levels 0, 1 and 2 of the tree. For each level, we assign a weight starting from 3 to 1 for levels 0 to 2 respectively, since a metric appearing at level 0 has the highest discriminatory power of all metrics. At the end, we multiply the frequency of each metric in a level by the weight of that level and we sum all the multiplications to obtain the score for that metric. For example, out of 100 iterations, the metric "ndev" has appeared 9 times in level 0, 5 times in level 1 and 3 times in level 2. We calculate the score for "ndev" like: 9*3 + 5*2 + 3*1 = 40. The higher the resulting score, the more important the metric would be for the explanatory models.

Findings. Our explanatory composite model shows that **(1) the number of developers working on a bug, (2) subject, message and stack trace information, and (3) reply lag are the main factors explaining small programming faults**.

Table VI shows the most important metrics after running 100 times our model. It clearly shows that, the metric number_of_developers_working_on_bug is the most important metric in the top-node analysis of the decision tree with a score of 193. Our analysis on the tree showed that if the number of developers working on a bug increases, the more likely the fault type would be a major programming fault. Also, we can see that the text variables (subject, message and stack trace) are amongst the most important factors. It demonstrates that the text content of a bug can be a good indicator to distinguish between small faults and other faults. Since the subject, message and stack trace information is available from the moment a bug is reported, this can open the door for actual prediction of whether an API failure would be easy to fix. Furthermore, the reply_lag metric indicates that the average time between bug report comments is another most important factor determined by our model for small programming faults. We also observe that the number_of_times_bug_status_changed never appeared in any iteration. The same goes for developer_experience, which confirms our findings in RQ3.

IV. THREATS TO VALIDITY

Construct validity threats concern the relation between theory and observation. Our metrics might not reflect all characteristics related to failures and we could include more metrics specifically related to source code or even the review process of bug fixes.

Internal validity threats concern other possible explanations for some of our observations. Since this study contains a qualitative study, there may be some human factors and subjectivity in the categorization analysis. Even though the first author performed the qualitative study, he frequently checked cases with other authors to double-check when in doubt. Moreover, in the fault categorization, the unknown and unresolved issues were not taken into account, since such issues are either not reproducible, not popular or important for the OpenStack project as a whole, or just too simple (typos in documentation, etc.).

In our quantitative study, our criteria to select the most important modules is based on the most frequently changed module, i.e, the most committed module file in *git* repository is considered the most important one in each API. This raises concerns that the most committed ones not necessarily reflect the most important ones.

On the other hand, we used heuristics on bug report messages and keywords to identify stack traces related to the APIs. One concern is that is not 100% reflecting all related API stack traces. Another concern is related to the bug activation in days. There are bugs that generally would be treated very soon but are closed at a specific time of each month with other bugs. Therefore, these days of the bug activation are not reflecting a 100% correct time for fixing the failures.

Threats to external validity concern the possibility to generalize our results. Since we have only studied one large open source infrastructure in the cloud, we cannot generalize our findings to other open and closed source (e.g., Google, Amazon, and Microsoft) projects. Moreover, since Git is a pliable version control system that can be used in various installments, we also need to take care when extending our results to other projects using Git. Therefore, more case studies on other projects are needed.

V. RELATED WORK

API changes. Wu et al. [33] studied the Apache and Eclipse open source projects to understand API changes and usages. They found that missing classes and methods are the most prevalent issues that affect the client programs in case of API usage. They also found that interface faults occur rarely in APIs and 11% of API changes have effects on the client programs. In our study we did not investigate the effects of API changes on the whole ecosystem of the OpenStack. Our statistics for API interface faults is 7% and this is close to the statistics presented by [33].

In other work, Dietrich et al. [34] studied the differences between Java compile-time and link-time compatibility in the Qualitas corpus and reported that such incompatibilities exist but it merely affects other client applications.

Dig et al. [1] studied five well known open source systems (Eclipse, Log4J, Struts, Mortgage and JHotDraw) in order to understand the API changes. They found that changes that break existing applications are not random and 80% of these changes are due to refactorings.

API failure in cloud. The closest work that categorizes the failures in the cloud environment is related to Lu et al. [4]. They studied nearly 900 issues related to API failures in the Amazon EC2 forums. They classified the causes of failure into three categories: development, physical and interaction. They found that 60% of the failures are related to either stuck API calls or unresponsive API calls. However, in their study they did not go into more details into the actual root causes of those failures.

Farshchian et al. [3] tried to detect anomalies in the cloud, focusing on Amazon EC2, by injecting faults and detecting them by using a regression based statistical modeling. Their work addresses DevOps operations such as backup, redeployment, upgrade, customized scaling, and migration. However, our work is different since we focused on the development faults and its classification by using a qualitative study. In fact, we extracted the actual bug fixes and analyzed them by using composite statistical modeling.

Gunawi et al. [7] did a large bug study on seven popular and important cloud systems (Hadoop, MapReduce, HDFS, HBase, Cassandra, ZooKeeper and Flume). They reviewed 21,399 submitted issues in the issue repository and concluded a set of detailed classifications from a variety of aspects such as reliability, scalability, etc. and the percentage of related issues in each category. In their work they also investigated the software fault types. However, their systems under study are not cloud infrastructure. Our study differs with them, in a way that they just studied issue repositories, but we covered mail and source code repositories as well. Their fault categories include: Error handling, Configuration, Race condition, Hang, Space and Load. The configuration, data race and even error handling fault types overlap with our small programming faults.

VI. CONCLUSION

In this paper, we conducted an empirical study to investigate the API failures in a cloud environment such as OpenStack-an open source cloud-infrastructure. First, we studied the source code repository to find out the percentage of changes in the APIs that were aimed for fixing bugs. We also included method signature changes in our study since this type of modifications to an API make it fragile and creates serious problems for other projects. We conclude that a median value of 31% (one third) of all changes fix API failures, where 24% includes non-method signature changes and 7% include method signature changes.

Second, during a qualitative study, on a random sample of fixed bugs for a variety of OpenStack projects, we explored the root causes of the failures by analyzing bug reports and fixes. Based on our observation, we found seven categories of causes (faults) for API failure: small programming faults, major programming faults, configuration faults, race conditions, deadlock conditions, data format faults and improper log message faults. Our finding indicates that the majority of the causes is due to small programming faults (56%), whereas a) the developers who have fixed the bugs are less active in the project in comparison to major programming faults, b) the time to fix these kinds of faults does not differ to other types of faults and c) the developer experience of these types of faults does not differ to each other.

The major programming faults category comprises two subcategories: method signature changes as well as changes in which multiple files have been touched. Limited to our samples, our statistics showed that major programming faults are 14% of all existing fault types, whereas only 6% of these major faults include method signature changes, confirming the result of our quantitative study.

Third, our explanatory model showed that metrics such as subject, message and stack trace information, number of developers working on a fix for a failure and reply lag within the comments are the main factors in small programming faults. These results open the door for prediction of whether a newly submitted API failure will be easy to fix.

We have to mention that we use Decision Trees to build our models, but other techniques such as Support Vector Machines (SVM) and Logistics Regression should be studied and compared. Also, additional characteristics should be explored in our model, because they might improve the performance of our model. Finally, we would like to collect more samples and more metrics in order to predict API failures in the future particularly for cloud environments.

REFERENCES

[1] Danny Dig, and Ralph Johnson. “How do APIs evolve? A story of refactoring.” Journal of software maintenance and evolution: Research and Practice 18.2 (2006): 83-107.

[2] Dana Petcu, Ciprian Craciun, and Massimiliano Rak. “Towards a cross platform cloud API.” 1st International Conference on Cloud Computing and Services Science. 2011.

[3] Mostafa Farshchi, Jean-Guy Schneider, Ingo Weber and John Grundy, “Experience Report: Anomaly Detection of Cloud Application Operations Using Log and Cloud Metric Correlation Analysis,” Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE ’15), IEEE, Gaithersburg, Maryland, November 2015

[4] Qinghua Lu, Liming Zhu, Len Bass, Xiwei Xu, Zhanwen Li, and Hiroshi Wada. “Cloud API issues: an empirical study and impact.” In Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures, pp. 23-32. ACM, 2013.

[5] LZ Xiwei Xu et al. “Error diagnosis of cloud application operation using Bayesianian networks and online optimisation.” 11th European Dependable Computing Conference (EDCC). 2015.

[6] Min Fu et al. “Process-oriented recovery for operations on cloud applications.” Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013.

[7] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T.Do, et al., “What bugs live in the cloud?: A study of 3000+ issues in cloud systems,” presented at the Proc. of the ACM Symposium on Cloud Computing, Seattle, WA, USA, 2014.

[8] Karan Aggarwal, Tanner Rutgers, Finbarr Timbers, Abram Hindle, Russ Greiner, and Eleni Stroulia. “Detecting duplicate bug reports with software engineering domain knowledge.” In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 211-220. IEEE, 2015.

[9] Gonzalez-Barahona Jesus, Gregorio Robles, Daniel Izquierdo-Cortazar, “The MetricsGrimoire Database Collection”, 12th Working Conference on Mining Software Repositories, Florence, Italy, 2015

[10] Audris Mockus, and David M. Weiss. ”Predicting risk of software changes.” Bell Labs Technical Journal 5.2 (2000): 169-180.

[11] Dan Radez, OPenStack Essentials, Demystify the cloud by building your own private OpenStack cloud. Copyright©2015 Packt Publishing.

[12] Kevin Jackson, Cody Bunch, Egle Sigler,OpenStack Cloud Computing Cookbook Third Edition. Over 110 effective recipes to help you build and operate OpenStack cloud computing, storage, networking, and automation

[13] http://www.openstack.org

[14] G.Kalton, Introduction to survey sampling. Sage Publications, Inc, September 1983.

[15] Tony A. Meyer, and Brendon Whateley. “SpamBayes: Effective open-source, Bayesian based, email classification system.” CEAS. 2004.

[16] Walid M. Ibrahim et al. “Should I contribute to this discussion?.” Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on. IEEE, 2010.

[17] Ian H. Witten, and Eibe Frank. Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann, 2005.

[18] Quinlan, J. Ross. C4. 5: programs for machine learning. Elsevier, 2014.

[19] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. IEEE Trans. Softw. Eng., 39(6):757-773, 2013

[20] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? IEEE Trans. Softw. Eng., 34(2):181-196, 2008

[21] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In Proc. Int’l Conf. on Softw. Eng. (ICSE’13), pages 382-391, 2013

[22] M. B. Miles and A. M. Huberman, Qualitative data analysis : an expanded sourcebook, 2nd ed. Thousand Oaks, Calif. : Sage Publications, 1994, includes indexes

[23] L. Barker, “Android and the linux kernel community,” http://www.steptwo.com.au/papers/kmc whatisinfoarch/, May 2005

[24] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in Proceedings of the 2013 International Conference on Software Engineering, ser. ICSE’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 712-721.

[25] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey, “The msr cookbook: Mining a decade of research,” in Proceedings of the 10th Working Conference on Mining Software Repositories, ser. MSR ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp.343-352.

[26] M. Hollander, D. A. Wolfe, and E. Chicken. Nonparametric statistical methods. John Wiley Sons, 3rd edition, 2013

[27] M. Shridhar, B. Adams, and F. Khomh, “A qualitative analysis of software build system changes and build ownership styles,” in Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM), Torino, Italy, September 2014

[28] A. E. Hassan and K. Zhang, “Using decision trees to predict the certification result of a build,” proc. of the 21st Int. Conf. on Automated Software Eng. (ASE), pp. 189-198, 2006

[29] T. A. Meyer and B. Whateley, “Spambayes: Effective open-source, bayesian based, email classification system,” in Proc. of the First Conf. on Email and Anti-Spam (CEAS), 2004

[30] R. Kohavi, “A study of cross-validation and bootstrap for accuracy estimation and model selection,” in Int. Joint Conf. on Artificial Intelligence (IJCAI), 1995, pp. 1137-1145

[31] Christian Macho, Shane McIntosh, and Martin Pinzger. ”Predicting Build Co-Changes with Source Code Change and Commit Categories.” 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Vol. 1. IEEE, 2016.

[32] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. “Can developer-module networks predict failures?.” Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, 2008.

[33] Wei Wu et al. “An exploratory study of API changes and usages based on apache and eclipse ecosystems.” Empirical Software Engineering (2015): 1-47.

[34] Jens Dietrich, Kamil Jezek, and Premek Brada. “Broken promises: An empirical study into evolution problems in java programs caused by library upgrades.” Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on. IEEE, 2014.