Southern Methodist University
SMU Scholar

Computer Science and Engineering Theses and Dissertations

Summer 8-7-2018

Cloud Service Reliability and Usability Measurement

Abdullah Bokhary
Southern Methodist University, abokhary@uj.edu.sa

Follow this and additional works at: https://scholar.smu.edu/engineering_compsci_etds

Part of the Software Engineering Commons

Recommended Citation
https://scholar.smu.edu/engineering_compsci_etds/6

This Dissertation is brought to you for free and open access by the Computer Science and Engineering at SMU Scholar. It has been accepted for inclusion in Computer Science and Engineering Theses and Dissertations by an authorized administrator of SMU Scholar. For more information, please visit http://digitalrepository.smu.edu.
CLOUD SERVICE
RELIABILITY AND USABILITY MEASUREMENT

A Dissertation Presented to the Graduate Faculty of the
Lyle School of Engineering
Southern Methodist University
in
Partial Fulfillment of the Requirements
for the degree of
Doctor of Philosophy
with a
Major in Computer Science
by
Abdullah Bokhary

B.A., Computer Science, King Abdulaziz University, Jeddah, Saudi Arabia
M.S., Software Engineering, Florida Institute of Technology, USA
M.S., Engineering Management, Florida Institute of Technology, USA

August 7, 2018
ACKNOWLEDGMENTS

This work could not have been accomplished without guidance of Allah, then the support of my advisor, my mam, and my wife.
Cloud computing has become a major resource for fulfilling people’s computational and storage needs. Investing in these services requires measuring and assuring its quality in general, and reliability and usability are primary concerns. However, using traditional reliability models can be challenging because of the environmental constraints and limited data availability due to the heterogeneous environment and diverse stakeholders. Also, the quality of cloud service Application Programming Interfaces (APIs) has a direct impact on the usability and reliability of the service.

We developed a framework to measure reliability with alternative available information that most cloud providers offer in three stages: 1) Defects are extracted and weighed from issue reports based on their validity, 2) Workload is measured by the number of clients as a new proxy to estimate daily clients usage, 3) Both results are linked together to examine the defect behavior over time. Software reliability growth models (SRGMs) are used to analyze this behavior, to assess current reliability, and to predict future reliability.

Google Maps APIs is used as a case study to demonstrate the applicability and effectiveness of our new framework. Then our framework is validated by extending the models to provide reasonably accurate long term reliability predictions.

Furthermore, we developed a comprehensive framework to measure and analyze cloud service APIs quality attributes in general and usability sub-attributes in particular. First, we
identify relevant quality attributes applicable to cloud service APIs. Second, we decompose cloud service APIs to measurable elements. Then we define metrics to quantify these quality attributes using decomposed elements. Lastly, we measure and analyze cloud service APIs usability using existing data sources from crowd source Q&A. We applied our framework on YouTube APIs and Stack Overflow to demonstrate its applicability and effectiveness.
TABLE OF CONTENTS

LIST OF FIGURES ............................................................................................................. ix
LIST OF TABLES .................................................................................................................. x

CHAPTER

1. INTRODUCTION ................................................................. 1

2. RELATED WORK ........................................................................ 4
   2.1. Quality Frameworks and Attributes ........................................... 4
   2.2. Quality and Defects ................................................................. 6
   2.3. Software Metrics ................................................................. 7
   2.4. Reliability ................................................................. 8
   2.5. Usability ................................................................. 11
   2.6. Cloud and APIs ................................................................. 12
   2.7. APIs and Cognitive Dimensions ........................................... 13
   2.8. APIs Metrics ................................................................. 15
   2.9. Data Source for Empirical Study ........................................... 17

3. PROBLEMS AND NEW SOLUTION ....................................... 18
   3.1. Quality in Cloud Services ................................................... 18
   3.2. Problems ................................................................. 20
   3.3. Solution Strategy ........................................................... 21

4. CLOUD SERVICE RELIABILITY ....................................... 22
   4.1. Overall Approach ........................................................... 22
   4.2. Defects Characterization ................................................... 23
   4.3. Workload Characterization .................................................. 25
   4.4. Reliability Modeling ........................................................ 26
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Framework to measure cloud reliability</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Number of websites using Google Maps APIs</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Regression line to estimate the number of websites using Google Maps API</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Cumulative websites count</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Weighted defects over time and over cumulative websites</td>
<td>39</td>
</tr>
<tr>
<td>4.6</td>
<td>Reliability assessment using selected models</td>
<td>40</td>
</tr>
<tr>
<td>4.7</td>
<td>Reliability prediction using selected models fitted to partial data</td>
<td>40</td>
</tr>
<tr>
<td>4.8</td>
<td>Google Maps APIs usage for the second period</td>
<td>41</td>
</tr>
<tr>
<td>4.9</td>
<td>Cumulative website counts for both periods</td>
<td>41</td>
</tr>
<tr>
<td>4.10</td>
<td>Defect behavior over calendar time</td>
<td>42</td>
</tr>
<tr>
<td>4.11</td>
<td>Defect behavior over website count</td>
<td>42</td>
</tr>
<tr>
<td>4.12</td>
<td>Long term prediction</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Decompose cloud service APIs to measurable elements</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Empirical study framework for cloud service APIs</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>Data summary for all metrics</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>TBM relating cloud APIs metrics to questions score</td>
<td>56</td>
</tr>
<tr>
<td>5.5</td>
<td>TBM for cloud APIs metrics to viewers score</td>
<td>57</td>
</tr>
</tbody>
</table>
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Google Maps APIs issue status</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Issues classification &amp; defect weight</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Assessment results</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>Prediction results based on models fitted to partial data</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>Google Maps APIs JS3 issue status before and after migration with our classification</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Long term validation results</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>The quality attribute and metrics framework for cloud service APIs</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Median and mean for all metrics</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>Correlation table between variables</td>
<td>54</td>
</tr>
</tbody>
</table>
This is dedicated to my father who could not see this work completed, to my mother for her motivation and support, and to my wife for her ongoing love and support.
Chapter 1
INTRODUCTION

The internet has given birth to cloud computing, which changes the way people use computation resources. According to the National Institute of Standards and Technology (NIST), cloud computing is a model that uses communication technology to allow global customers to share computing resources such as networks, servers, storage, applications, and services [50]. In addition to providing cloud computing services to end users, organizations use it to improve performance and reduce cost by replacing server rooms with cloud computing. Developers also take advantage of cloud computing to embed cloud services in their systems and applications to satisfy their customer’s needs.

The high demand for cloud computing increases the need to assure its quality, including reliability, usability, and security as primary concerns [60]. This dissertation focuses on reliability and usability in cloud services from the client perspective. Reliability will be assessed from the client perspective, where the client is the developer who embeds or integrates the provided cloud service in his or her application or system in general. Also, cloud service APIs’ quality will be investigated and studied because it has a direct impact on reliability and usability of the service.

Reliability is an important characteristic in building a heterogeneous system with intrinsic high complexity. Software reliability is the probability of not having a failure over a specific period [56]. Therefore, we defined the cloud service reliability as the probability of not having a failure for the services, where a failure is the inability to correctly process a customer request. Researchers and practitioners have used many methods to measure and predict software reliability during development or operation phases. However, these traditional approaches are difficult to apply in cloud computing because of the environmental constraints and limited data availability. Unlike in traditional software systems, defects can be associated
with heterogeneous components distributed over wide areas and different layers of the cloud infrastructure. Workload measurement is harder to obtain due to the different stakeholders involved. Also, service providers may rely on clients to report defects who may be reluctant to share detailed circumstantial information about these defects due to legal and proprietary concerns. These limitations need to be taken into consideration when we address cloud reliability problems from the perspective of clients or developers who use these services.

Usability, on the other hand, is the degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use [38]. Therefore, The usability of cloud service is to the degree to which the cloud service can be used by client or developer to be integrated into their system with effectiveness, efficiency, and satisfaction. To assess cloud service usability we need to identify the way that client use these services first, which are Graphical User Interfaces (GUIs) and Application Programming Interfaces (APIs). Our scope in this study will be limited to the APIs’ access only since it is the most common way to embed most cloud service to any system. Actually, APIs have been used in frameworks and operating systems for decades, and there are many approaches to improve its quality. However, APIs for cloud services are working on a different platform, and it has different constraints and circumstances. Moreover, most of the studies are proposing standards and guidelines for APIs design and structure with very limited empirical studies to prove the effectiveness and efficiency of these standers and guidelines. Therefore, an empirical study for API quality with a concentration in cloud service constraints will support these standards and guidelines.

In this dissertation, we propose a reliable framework which overcomes the limitations to assess and predict cloud service reliability. The framework has three stages: 1) extract and process defect data from clients’ report system, 2) identify and extract proper workload from client usage, 3) use both results to assess and predict reliability using existing reliability models. Google Maps APIs was analyzed using the proposed framework as a case study to demonstrate its applicability and effectiveness.
We also analyze APIs in cloud service circumstances and investigate the relation between APIs’ elements and usability attributes. Then we propose a framework to conduct an empirical study to understand the influence of the APIs elements on quality attributes.

The dissertation is organized as the following. Chapter 2 includes background and related work. Chapter 3 is an overview of problems and the new solution. Chapter 4 describes our new reliability framework and uses it on Google Maps APIs to assess and predict its reliability. Chapter 5 investigates cloud service APIs’ quality and proposes a framework for an empirical study that helps to understand the influence of APIs’ elements on quality. Chapter 6 summarizes the dissertation, the accomplished, and the future work.
Chapter 2

RELATED WORK

Software quality is usually associated with satisfying user expectations as characterized by user requirements and product specifications [24] [38]. This chapter examines related work about quality, reliability, usability, cloud, and APIs.

2.1. Quality Frameworks and Attributes

Assessment of software quality required first to understand how the community defined it [35] [77]. Organizations tried to identify the software quality characteristics to improve their systems and applications. There are several software quality standards that scientists and practitioners adapted since 1970’s. Most of these standards share many characteristics and attributes. However, with the passage of time some attributes have been renamed, merged, or substituted with new attributes. These changes can be referred to the evaluation of software, infrastructure, and communicating technologies. In fact, evaluations have a direct effect on customers’ usage and needs which reflect on the requirements. As a result, changing of requirements will lead to a change of quality attributes which makes the differences between these standards.

One of the early software quality standards is a technical report by McCall et al. for U.S. Air Force Electronic System Division’s [15] [49]. The document was made to provide standard and technical guidance to software acquisition managers. They proposed Factor-Criteria-Metrics(FCM) approach to decompose quality to eleven main quality factors and these factors are decomposed to 25 criteria. They address these criteria with 41 metrics. We will consider only the relevant characteristics from this reports. First, reliability is the extent to which a program can be expected to perform its intended function with required precision. Another characteristic is usability defined as the effort that required to learn, prepare input,
and interpret output of a program. Reusability is also a relevant quality characteristic which has been used in some of the literature for APIs in a specific environment. It is defined as the extent to which a program can be used in other applications related to the packaging and scope of the functions that programs perform. Digging deep into each characteristic we find reliability is decomposed to error tolerance, consistency, accuracy, simplicity. Where usability is decomposed to training, communication, and operability. Finally, reusability is decomposed to generality, modularity, software system independence, machine independence, self-descriptiveness. The document also shows some relationships between software quality characteristics. Such subcharacteristics should be taken in consideration to cover different perspectives of quality.

Later, the Software Engineering Institute (SEI) proposed a framework to measure the capability of organizations to produce quality software for US governments which called Capability Maturity Model (CMM) [68] [47] [64]. The model has five scaled levels, starts from ad hoc through repeatable, defined, managed, and optimizing. The organization that has better understanding and control on their resources and development process gets of higher level.

Beside FCM, several other approaches have also been used to investigate quality. Basili proposed Goal-Question-Metric (GQM) approach, which is a top down method to investigate factors that influence software process [9] [76]. The approach starts with identifying the goals of measurement. Then, ask questions to determine the way to achieve the goals. Finally, define the metrics that will provide quantitative answers for each question.

CUPRIMDSO is an IBM standard for software which stands for capability or functionality, usability, performance, reliability, installability, maintainability, documentation or information, service, and overall [42]. The most relevant to cloud service APIs from these standards are reliability, usability, and documentation.

The most widely used quality standards for software engineering are several versions from International Organization for Standardization (ISO). [36] [37] [38]. The latest Standard IOS/IEC 25010:2011 divides software quality to a quality in use model and a product
quality model each model consists of characteristics which subdivided to subcharacteristics [38]. Quality in use model related to the interaction with the product and composed of five characteristics that include: effectiveness, efficiency, satisfaction, freedom from risk, and context coverage. On the other hand, product quality model is related to static properties of software and dynamic properties of the computer system. It is composed of eight characteristics including functional suitability, performance efficiency, compatibility, reliability, usability, security, maintainability, and portability.

Web-based software changes the perspective of measuring software quality. According to Offutt, the high competition on web-based software and its natural increase customer awareness of quality and allowed them to move their business from provider to another once they discover it [60]. His survey shows that the important quality process divers for software was changed from time to market in traditional software to reliability, usability, and security in web-based software.

Denning said software quality should be measured by customer’s satisfaction [20]. He divides satisfaction to three levels. First level fulfills the basic requirement. Second level prevents negative consequences that may occur. Highest level where system excites the expectations of customers by doing much more work than they ask for.

### 2.2. Quality and Defects

The concept of defects, error, fault, and failure have a direct relationship to software quality. In fact, defects considered to be the major metric to measure quality from the development point of view. Most software enterprises ensure quality by examining intermediate and final products to detect defects. [14] [25] [43]. According to ISO/IEC/IEEE 24765-2017, a defect is an imperfection or deficiency in a work product where the work product does not meet its requirements or specifications and needs to be either repaired or replaced [40]. The life cycle of defects starts with a human error which transfers to faults, then to failures. Error is defined as a human action that leads to inject the system with faults or bugs. Therefore, a fault is the human error in the form of requirement, design, or code. For instance, when a
developer misunderstands user requirement, he or she will transfer this misunderstanding to the design and code. Errors can lead to one or more faults as much as this misunderstanding is used in design and code. Failure is the inability of a system or component to perform its required functions within specified performance requirements or time. Therefore, failure is related to system behaviors according to requirements. In other words, faults are what developers see and failures are what customers see. However, if the faults never executed, then failure will never occur. Therefore, both developers and customers concern about failures because they are what effect system operation. Failures can be measured directly by count, distribution, and density, etc. Also failure can be classified by its severity to capture its impact to the system or environment [22] [71].

2.3. Software Metrics

Software metrics can be divided into external product attributes and internal product attributes [22]. External product attributes include quality attributes such as reliability, usability, maintainability, etc. Some of these attributes will be covered in other sections. Internal attributes are divided into size metrics and structure metrics.

Size can be described by length, functionality, and complexity. Source Line of Code (SLOC) is the common metric for length that not includes blank lines and comment lines [19] [29]. In some cases, comment density can be used to represent self-documentation [22]. In addition, Software Engineering Institute (SEI) proposed a framework to measure software size that can address the influence of need [63]. However, SLOC is more influenced by programming language and technology rather than solving the problem. Therefore, functionality is used to reflect a better picture of product size from user perspective. The common method to measure functionality is function points by measure the amount of functionality in the system, which can be extracted from specification or high level design. Size complexity has two dimensions: time and space needed to complete a job. It can be measured by efficiency of the solution which can be deafened by big-O notation [42] [45].
On the other hand, structure attributes can be measured in different ways depending on the component of structure which includes control flow, data or information flow, and data structure. McCabe proposed cyclomatic complexity to measure the control flow structure by measuring the number of linearly independent paths [48]. Henry and Kafura proposed information flow measurement approach by considering the number of data that passed and received by the module to calculate its complexity [34]. There are several approaches to measure data structure. Boehm used the ratio between database size and the number of delivered source instructions as metric to measure the amount of data in a system which is part of COCOMO model [12].

Chidamber and Kemerer proposed metrics suite to measure the structure of Object Oriented Design(OOD) [16]. This suite contains six metrics including weighted methods per class, depth of inheritance tree, number of children, coupling between object, response for class, and lack of cohesion on methods. Some of these metrics can be adapted to measure the quality of APIs.

2.4. Reliability

Software reliability is defined as the probability of a software system to perform its specified functions correctly during a specified exposure period under the customers’ usage environment or similar environments [56]. The period can be a single run, number of runs, calendar time, or execution time unit. There are several approaches that been used to measure software reliability, but the most popular are Input Domain Reliability Models (IDRM)s and Software Reliability Growth Models(SRGMs) [21] [71].

IDRM)s use input states such as the low level architecture of the system and its defects to measure the reliability in the current state only. An example of such models is Nelson model where reliability is the ration between the successful executions and the total number of executions. [59]. This model can be applied by ruing a few samples of different inputs and
classes. Therefore, the estimated reliability $R$ will be expressed by

$$R = \frac{n - f}{n} = 1 - \frac{f}{n}$$  \hspace{1cm} (2.1)$$

where $f$ is the number of failures and $n$ is the total of runs. The advantage of this model that is ease of applying and random testing is passable. It is suitable to measure software reliability for devices that have limited functionality. On the other hand, the model becomes inaccurate for large samples and does not predict future reliability.

SRGMs are time-based models commonly used to assess and predict reliability by analyzing defects data over time [21]. The reliability growth is due to the defect detection and fixing that lowers the number of system faults and leads to improved reliability over time. Choosing the suitable models should consider the model required data and the assumptions of such models [21] [26] [56].

One of the widely used SRGMs is Goel-Okumoto model (GO) from the Non-Homogenous Poisson Process (NHPP) model class [27]. In this model, the number of failures in the system is finite, and the mean value function for the number of failures is:

$$\mu(t) = N(1 - e^{-bt})$$  \hspace{1cm} (2.2)$$

which predicts the cumulative defects in each given time ($t$) with constants $b > 0$ and $N > 0$, where $b$ and $N$ can be estimated from observation data.

Another NHPP model is the S-shaped model [62] [79], which considers the learning curve in the beginning and uses the following formula:

$$\mu(t) = N(1 - (1 + bt)e^{-bt})$$  \hspace{1cm} (2.3)$$

where $t$, $b$, and $N$ are similar to the GO model.

Another widely used NHPP model is Musa-Okumoto Model(MO) [52] [54]. The failures are infinite in this model and it is required the time that failures occur without concern
about completion time. It uses the following formula:

\[ \mu(t) = \beta_0 \ln(\beta_1 t + 1) \]  

(2.4)

where \( \beta_0 \) and \( \beta_1 \) are constants that can be estimated from observation data. The logarithmic model has been extended to address the earlier discovered failures reduces failure intensity [57].

The period can be measured by time units that reflect actual usage by the customer and users, because software failures are triggered by actual usage that exposes some internal defects. Software reliability modeling used calendar time as the time measurement until Musa introduced execution time to better characterize actual system usage or workload [56]. Alternative usage related time measurements have also been used for reliability modeling, including test runs and transitions from test tracking reports and number of usage instances and web traffic extracted from web logs [5] [41] [59] [69] [70] [73].

Although cloud computing provides the advantage of sharing computing resources, it has some limitations. One such limitation can be a result of encapsulation of cloud service which hides the system specification including low level architecture. Also the traditional data sources of defect such as testing reports or log files are generally unavailable for clients due to their legal or proprietary concerns.

Past failure data of other similar users were used to predict the web service reliability for the current users [81] [82]. An enhancement to this work was done by considering provider and client location, service load, and computational requirements [66]. However, this approach required historical data from other similar users which might not be available for cloud services. Also the approach does not predict future reliability.

Available methods for measuring software reliability would be challenging to apply in cloud service because of the limited data availability. Calendar time is not an accurate workload representation in reliability models for cloud services due to the fast growth or change of service usage. Also execution time, or the number of service invocations are not available. Therefore, we need an alternative data source that represents defect behavior over
appropriately defined usage time to measure cloud service reliability.

2.5. Usability

According to IOS/IEC 25010:2011, Usability is defined as the degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, and satisfaction in a specified context of use. Usability can either be specified or measured as a product quality characteristic in term of its subcharacteristics, or specified or measured directly by measures that are a subset of quality in use [38]. In our case, we will use the subcharacteristics of product quality since we interested to improve the product which is APIs itself. However, these subcharacteristics have been updated for the previous standard ISO/IEC 9126-1:2001 [37]. The following is the update of subcharacteristics of usability:

- Appropriateness recognizability: used to be called understandability and it defined by degree to which users can recognize whether a product or system is appropriate for their needs.
- Learnability: degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.
- Operability: degree to which a product or system has attributes that make it easy to operate and control.
- User error protection: new subcharacteristics defined by the degree to which a system protects users against making errors.
- User interface aesthetics: used to be called attractiveness and defined by degree to which a user interface enables pleasing and satisfying interaction for the user.
- Accessibility: new subcharacteristics defined by the degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.
There are several works on web application usability. Geng and Tian proposed a method to identify and improve web interface usability [23]. The method uses web server logs to compare between usage patterns against cognitive user models. The new method helps to discover usability issues and suggest corrective actions to improve web interface usability.

2.6. Cloud and APIs

Cloud services are accessed through Graphical user interfaces(GUIs) or Application programming interfaces (APIs). In this study, our concern is APIs and its quality. Apart of this study dedicated to investigating the cloud service reliability, the other community concern of cloud service that we focus on is the usability of the services. Usability in general is defined as the degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use [38].

Before defining cloud service usability, we need to define some keywords from the definition in cloud services which they are: product, user, goals, and context. First, the product or the system we are targeting is the cloud service. The user in our case is the developer, who wants to integrate some functionality that cloud service offers without concern about the internal code. In other words, API of service is similar to the functionality of word processor or spreadsheet. These applications have tens of functionality that by itself alone achieve a small contraption such as table or sum of numbers, but together will achieve the goal such as a letter, technical report, or financial statement. Finally, the context of use in this situation is where developer sees and access APIs which is the programming environment. Therefore, Usability of APIs cloud service is to which degree the APIs can be used by client or developer to integrate the service in their system with effectiveness, efficiency, and satisfaction in the programming environment.

Zheng and other proposed cloud quality dimensions and metrics which include usability of cloud. They consider the availability of three features of cloud service as the metrics of its usability. These three features are graphical user interface(GUI), application programming interface(API), and web user interface(WUI). So, if the service provides one of these fea-
turers it has 0.33 usability, where service that offers all three features has 1.00 usability [80]. However, this measurement is very abstract and not accurate since it is not measuring the usability rather that it is counting features in the system which does not have a direct impact on usability.

2.7. APIs and Cognitive Dimensions

Clarke et al. from Visual Studio Usability Group at Microsoft proposed cognitive dimensions framework to evaluate the quality of APIs [18]. Their work was inspired by Green and Blackwell’s framework for cognitive dimensions of information artifacts which contains thirteen dimensions [30]. Clark et al. tried to map a previous API usability study results to the original cognitive dimensions framework. The study result could not be described by the framework because it might fall under one or more dimensions, and dimensions could not translate the industrial setting so well. Therefore, only ten of the original framework dimensions have been adapted. Also, they renamed some of the dimensions and add two new dimensions to be more applicable to their work environment. These two dimensions are Working Framework and Work-step Unit. The following is the list of the adapted dimensions with the new definitions:

- Abstraction Level: what are the minimum and maximum level of abstraction exposed by the API, and what are the minimum and maximum levels usable by a targeted developer?
- Learning Style: what are the learning requirements posed by the API, and what are the learning styles available to a targeted developer?
- Working Framework: what is the size of the conceptual chunk needed to work effectively?
- Work-Step Unit: how much of a programming task must/can be completed in a single step?
• Progressive Evaluation: to what extent can partially completed code be executed to obtain feedback on code behavior?

• Premature Commitment: to what extent does a developer have to make decisions before all the needed information is available?

• Penetrability: how does the API facilitate exploration, analysis, and understanding of its components, and how does a targeted developer go about retrieving what is needed?

• API Elaboration: to what extent must the API be adapted to meet the needs of a targeted developer?

• API Viscosity: what are the barriers to change inherent in the API, and how much effort does a targeted developer need to expend to make a change?

• Consistency: once part of an API is learned, how much of the rest of it can be inferred?

• Role Expressiveness: how apparent is the relationship between each component and the program as a whole?

• Domain Correspondence: how clearly do the API components map to the domain?

Are there any special tricks?

Later, they categorized developers to groups according to their behavior or profiles which are: Opportunistic, Pragmatic, and Systematic [17]. In conclusion, their work created common terminology between APIs developers and usability groups. Also their framework can measure the degree of convenience between APIs and developers group in each dimension.

The study redefines the dimensions to fit with APIs environment, but renamed dimensions are not referred to which original dimensions it came from. Also the new dimension called Working Framework could be under one of the original dimension that removed called Visibility. However, Green stated about this new framework “the original conception of cognitive dimensions has been diluted; but on the other hand, the loss of purity is evidently accompanied by greater real-world applicability” [31].
2.8. APIs Metrics

Bloch provides some guidelines steps to design good APIs [11]. He lists 38 points that developers should consider to design APIs. These points focus on four aspects: development performance and process, classes and methods design, and documentation. However, these guidelines missing an empirical study that shows the degree of the impact on the user.

Robillard conducted a survey in Microsoft to gather information about developer experiences and obstacles they faced with learning APIs [65]. The survey contains thirteen questions includes six open end questions about obstacles which answered by 80 developers. The responses are classified by five major categories. Most responses emphasize resources to learn API which is documentation, and what documentation should contain? One of the most important elements in documentation is the example. Robillard divides it to three types. First, the Snippets which is a small portion of code. Some responses complain that this type of example does not satisfy developers in a complex situation such as involving multiple function calls. Therefore, they must use another type of example. Robillard list tutorials as the second type which is typically longer than snippets , and consist of multiple segments of code. Tutorials intend to teach developers specifies aspects of API. The third type of examples is a complete application which is more detailed and can be offered by open source projects. The survey shows that old documenting can be a critical obstacle for the developer, and source of documentation play important role in credibility. Therefore, the developer will use code from API documentation without any hesitation while internet code will be his or her last option. Another obstacle from the survey shows that high-level design documentation would be not enough in some cases, and developer needs more internal information. Robillard suggests documentation should be supported by low-level design documentation to adders the structures that impact the APIs behavior. The paper succeeded to gather many obstacles to learn APIs from practitioners, and to categorize these obstacles. However, an empirical study is needed to show the degree of the impact on the developers.

Zibra did a comprehensive literature review to collect most factors that make APIs difficult to use [83] [84]. He lists twenty two factors that appeared in the literature. Most of
these factors related to design issue such as complexity of exposed features, data type, parameters, return type, and other. It also includes some process factories such as API version and maintainability.

One of the related environment to cloud services that has similar circumstance is Component-based software development (CBD). Wahizaki and other proposed a metrics suite to measure CBD reusability [78]. They used McCall’s Factor-Criteria-Metrics (FCM), which is mentioned in Section 2.1, to map between metrics and subcharacteristics of reusability which include understandability, adaptability, and portability. Some of these metrics in this article are not applicable for APIs. For instance, the existence of meta-information (EMI) is a criteria that represented by the existence of BeanInfo class for each component which is not the case in API. However, this can be substitute by documentation for each API, but the existence of documentation would be a very abstract measurement for understandability. Also the rate of component observability (RCO) and the rate of component customizability (RCC) are criteria to measure the ratio between the number of variable in the component and the number of the method that read these variables (get), or the ratio between the number of variable and number of the methods that change these variables (set) respectively. However, the relation between the criteria and such metrics is not clear. Another criteria is Self-Component’s Return Value (SCCr) which represent by the ratio between the methods that return value to the methods that don’t return value to represent the dependency of the method which is not accurate.

Mosqueira-Ray et al. [51] adapted usability framework that proposed by Aloso-Rios et al. [6]. The framework was based on literature including old ISO standard. However, the structure of attributes and subattributes are not well justified. For instance, understandability and learnability are merged into knowability. One of the subattributes of knowability is remembering how to use which should be more related to operability. Also safety has been added to the attributes and related to system breach and confidentiality which are more likely security subattributes.
2.9. Data Source for Empirical Study

Measuring software quality required a data source to extract metrics that linked to quality attributes. We mentioned some of the data sources used for quality in different environments which are summarized in the following.

Testing reports, during software development cycle, have been used as a direct resource for defects data to analyze and measure reliability [21] [56]. Besides testing reports, log files play an important role in quality measurement. They have been used to extract several metrics for different environments. In traditional software, log files used to extract metric including defects to measure and analyze reliability [44] [74]. In web environments, server logs has been used to classified errors [4] [41] [46] [72] [73]. Others used client log files for the web as a source to extract quality metrics [81] [82].

Web analytic sites offer a large amount of data for organizations that help them to study the customer behavior and market to improve their service [1]. These data source can be used to analyze software quality or associated with other metrics to improve accuracy.

One of the internet advantages is the interacting forums. These forums can be restricted between client and service provider such as help disk system or issues tracking system. In such forums, client starts a post and provider follow up with feedback. Another type of forums is open to the public such as crowd Q&A where anyone can start a post or participate in the same post. Most of these forums is a public access which can be used for an empirical study. Stack overflow is one of the crowd Q&A forum that used intensively for empirical study in general [8], or evaluating crowd post for different purpose [3] [28] [32] [75].
Chapter 3
PROBLEMS AND NEW SOLUTION

The high demand for cloud services increases the need to improve its quality. The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value [38]. The primary concern is to improve quality in general. The environment of a cloud service increases the complexity of the system and introduces new constraints. This chapter will describe problems of assuring quality under these environmental constraints and propose a solution.

3.1. Quality in Cloud Services

We addressed some of the quality standards in Section 2.1. Most of these standards share many characteristics and attributes. We notice that with the passage of time some attributes have been renamed, merged, or substituted with new attributes. These changes can be referred to the evolution of software, infrastructure, and communicating technologies. In fact, evolution has a direct effect on customers usage and needs which reflect on the requirements. As a result, the changing of requirements will lead to a change of the quality attributes, which makes the differences between these standards. Offutt’s survey shows that in traditional software the most important factor is the time to deliver the product and maintenance can be performed later where as the primary concerns in web-based software are reliability, usability, and security [60]. The survey supports that change of technology and environment influence of the quality requirement. We will elaborate on each of these quality attributes and its influence on cloud services.

Reliability is one of the important attributes that emphasize functional requirements. It is defined as the probability of a software to perform its specified functions correctly during a specified exposure period under the customer’s usage environment or similar environments.
Cloud service reliability can be defined as the probability of not having a failure for the service, where a failure is the inability to correctly process a customer request [13]. One of the common metrics to measure reliability is failures over time. In traditional software, failures can be collected during the testing phase or operation from testing team reports or system log files respectively. Time can be represented by calendar time, execution time, and web transactions [41] [52]. However, cloud service requires a different source of metrics to measure reliability in the new environment. Therefore, defects can be extracted from server traffic logs with different methods considering the related defects only [4]. Also, calendar time and execution time become inappropriate metrics because they don’t represent the workload on the service. Therefore, service invocation would be a better representation for workload which can be extracted from provider log files. Otherwise, a proxy for workload can be used as an alternative metric [13].

Usability in traditional software plays an important role to improve customer satisfaction. It is defined as the degree to which a product or system can be used by specified goals with effectiveness, and satisfaction in a specified contest of use [38]. Where usability of cloud service can be defined as to which degree the cloud service can be used by client or developer to be embedded in their system with effectiveness, efficiency, and satisfaction. In fact, cloud service and traditional software have common metrics in usability when they both use Graphical User Interfaces (GUIs) to access the system. However, cloud service offers Application Programming Interfaces (APIs) as an additional or main method to access their service. This new method brings new metrics to this quality attribute. Therefore, a new framework needs to be developed to consider the APIs usability metrics.

Security is defined as the degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization [38]. According to ISO, cloud service security is a sub-role of cloud service provider which has the responsibility of ensuring that the information security policies of the cloud service customer and the cloud service provider are aligned and meet the security requirements stated in the SLA [39]. Software security
impact directly with accessibility which is very high in cloud service and can be very limited in traditional software. The distribution of cloud service component and layers change the concept of security measurement. Data are exposed through HTTP and the firewall is no longer the main protection for data. Therefore, new quality metrics for cloud service that address the accessibility, system distribution, and data exposed are changing the way that security is measured.

We focus on reliability and usability as the primary quality attributes since they are the primary concern in cloud service. Although security is one of the primary concern for cloud service, it is out of our scope since it required an independent investigation with a security specialist.

3.2. Problems

One of the traditional approaches that measure software reliability uses software reliability growth models (SRGM) to assess the current reliability and predict the future reliability. This approach required an accurate defects behavior over time with appropriate representation for the workload. The defects can be extracted from provider log file, and workload can be extracted from service invocations. Both resources are not available for clients or developers, who integrate the service in their system. The limited access to these resources is related to security and privacy issues. Also, the size of the log file can be extremely large to extract defects behavior over a long period, which motivates the provider to search for alternative source. Alternative resources need to be used to represent the defects behavior and workload.

Clients or developer want a convenient way to embed or integrate these services to their system. APIs with high quality will make the job easier and decrease effort and time besides the impact to their product reliability. There are many metrics that are used in guidelines and standards to improve APIs quality. However, they don’t address the cloud service APIs constraints and environment. Also, there is no clear relationship between these metrics and quality subcharacteristics of APIs. Moreover, there are limited empirical studies to prove
the effectiveness and efficiency of applying these metrics. An empirical study is needed to support the impact of APIs elements on the cloud service APIs quality.

### 3.3. Solution Strategy

The limitation of access to quality metrics in cloud service increases the need for alternative resources. Also, the lack of empirical studies that shows the influence of quality on user increases the need for evidence. Our solution strategy is using an alternative source of information to extract the quality metrics or proxy for them.

To measure cloud service reliability, we propose a framework to use the issues report system and client numbers as an alternative resource for log file to study defects behavior in three stages: 1) Defects are extracted and weighed from issue report based on their validity. 2) Workload is measured by the number of clients as a new proxy to estimate daily clients usage. 3) Both results are linked together to examine the defect behavior over time. Software reliability growth models (SRGMs) are used to analyze this behavior, to assess current reliability, and to predict future reliability.

On the other hand, cloud service APIs can be mapped to quality subcharacteristics after decomposing APIs components, such as design and documentation, to quantitative elements. Then we derive the metrics for each quality subcharacteristics. Also, we propose a framework to analyze the impact of APIs metrics on the overall usability metrics. This study starts by matching between each cloud service API method and its relevant discussions in the crowd Q&A. The result will be a list of methods with all relevant discussions to each method or API. By using multivariate statistical analysis between API metrics and overall usability metrics, which extracted from crowd source Q&A, we can evaluate the impact of API metrics on quality attributes.
Cloud service reliability is a critical factor that every provider and client are concerned about. The heterogeneous environments and frameworks at provider and client impact its reliability. Ensuring the reliability of the service is one of the main requirements for both parties. The limited information that is available prevents clients from using traditional measurements for service reliability. Also the large log file that needs to be processed motivates the provider to look for an alternative resource to measure the service reliability. We propose a new framework that uses weighted defects from issue reports over a new client usage proxy to characterize defect behavior over time and to assess and predict reliability.

4.1. Overall Approach

The limited information of cloud computing, such as lack of access to source code or execution logs, prevents clients from directly applying the traditional reliability models. Also, the difficulty of analyzing the extremely large log file motivates providers to search for alternative metrics to measure and predicate the reliability of their services. In order to measure reliability, an alternative measurement of defects over an accurate representation of workload need to be used.

Most cloud providers offer defect reports or a feature requests system. This system can be a source for defect data, since it has all the details of each defect, including discovery time and how it has been treated. However, these defect data are in calendar time, which is not an accurate usage metrics for cloud service due to large usage variations. Also, invocation count for the service by customers is not available. Therefore, we propose to use the number of clients instead of calendar time or service invocations. In other words, the number of clients that accessed the service when defects were reported can be a proxy for usage. These
types of information are offered by some providers or available in external resources such as web analytics sites.

This paper proposes a new framework to measure cloud service reliability in three stages: defects characterization, workload characterization, and reliability modeling, as shown in Figure 4.1 and briefly described below:

- **Defects characterization**: Extract and process defects from the issue tracking system which contains all issues that the clients have reported or suggested to improve the service. Then classify each issue according to its validity. The result of this stage is weighted defects depending on their validity.

- **Workload characterization**: Extract service workload from a published number of clients statistics. Then estimate the number of clients for each day to calculate the cumulative usage or workload.

- **Reliability modeling**: Use the weighted defects over clients usage to plot the defect profile. Then use SRGMs to assess the current reliability and predict future reliability.

Details of the framework and its three stages, including individual steps in each stage, are described below.

**4.2. Defects Characterization**

An issues report system is offered by most cloud service providers for clients or developers who use the services. Since each record in the system is created by a client, it needs to be confirmed by a provider before it is considered as a defect. Therefore, we propose the defects characterization stage in Figure 4.1 consisting of:

1. Screening: Collect all records from issues report system, and exclude all records unrelated to defect observations or the target environment.

2. Classification: Categorize remaining issues into three types: valid, invalid, and uncertain.

The screening step is to extract issues from the issue system that the provider offers to record all client issues, including system failures or any request to improve or add new functionality. The issue system covers all types of environments or programming languages that the provider uses to deliver the service. The screening step should exclude unrelated data such as enhancement requests or defects related to languages or environments outside the study scope. The result should be all defect issues in the specific services or the environment for the specific clients of a given study.

The classification step is to classify the output of the previous step into three categories:
valid, invalid, and uncertain. This classification is not part of the data rather than a label we provide according to the provider team response to each issue. Valid issues are all records that are agreed as defects by the service provider. Invalid issues are all records that the provider disagreed as defects due to several reasons, e.g. it can be labeled as a duplicate, it works as intended, or it is obsolete because of a new release. The uncertain issues are all pending records that the provider did not categorize yet because it is a new record or awaiting clarification. The result of this step is a classification for all issues under the three classes we suggested.

The consolidation step counts all the valid defects but excludes all the invalid ones. Uncertain issues will be considered according to the history of the clients’ issues. Actually, every issue starts as a new status which is uncertain, then it ends eventually as a valid or an invalid issue. Therefore, we can use the historical data to estimate the likelihood of its validity. In other words, we use the valid to the invalid ratio as a weight for uncertain issues using this formula \( w = \left( \frac{V}{V+I} \right) \), where \( V \) and \( I \) are the sums of valid and invalid issues respectively. In effect, the uncertain issues are partially counted as defects according to the weight.

4.3. Workload Characterization

Calendar time is not an accurate representation for the workload in cloud service due to large usage variations. Also, the number of invocations for the service is not available for the clients. Therefore, we propose the workload characterization stage in Figure 4.1 to use an alternative workload measure, the number of clients, which consists of the following steps:

1. Extraction: Find and extract statistical information about clients count of the service.
2. Estimation: Estimate the daily clients count using a statistical model.
3. Accumulation: Calculate the cumulative clients count by the suggested formula.

The extraction step is to find alternative data sources for workload since the number of service invocations by end users or customers is not available for the clients. As a proxy of
cloud service usage, we extract the number of clients from the service provider report or from a public report that include usage statistics about the desired service. In other words, the report should provide the number of clients, such as websites or mobile applications, that embedded the desired service in their system during the investigation period. The number of clients will be used as a proxy for the workload in reliability models. The result of this step is the number of clients that accessed the service on some given days.

The estimation step concerns about the number of clients that use the cloud service each day. If the report does not show clients access for every day, we propose to estimate this number by applying a statistical model or some other estimation methods on the available data. In effect, we use such models as interpolations to estimate the number of clients for every day by filling the gaps in the available data.

The accumulation step is to calculate the cumulative number of clients that embed the service in their software for the investigation period using the following equation:

\[
U_i = \sum_{j=d_0}^{d_i} N_j
\]

(4.1)

where \( U_i \) is the cumulative usage or cumulative clients count up to day \( d_i \) and \( N_j \) is the number of clients in day \( d_j \) that we estimated in the previous step.

### 4.4. Reliability Modeling

Most SRGMs require time between failures or the actual time instance or period that the software failed. In our framework, the time is represented by the sum of the number of clients that used the service each day until the day of the specific defect discovery. Therefore, the cumulative weighted defects over the cumulative clients’ count will be used to plot the defects behavior over time for cloud services. Then, we assess the current reliability and predict the future reliability using the suggested models in Section 2.4. The reliability modeling stage in Figure 4.1 consists of the following steps:

1. Defect behavior profile: Plot weighted defects over clients count to examine the defect
2. Assessment: Use SRGMs to assess the current reliability of the service.

3. Prediction: Use SRGMs with partial data as training data set to predict future reliability.

The first step employs the results of the previous stages to plot the defect behavior profile. The result of defects characterization stage is the cumulative weighted defects in their arriving sequence. The result of workload characterization stage is the number of cumulative clients that subscribed to the service up to a given day. We link the weighted defects and cumulative clients count using the data. Plotting the defects that we extracted in the defect characterization stage over the cumulative clients’ count allows us to examine the shape and trend of the defect profile over time.

The assessment step uses the output of the previous step to assess the cloud service reliability using the selected SRGMs models in Section 2.4. The defects behaviors profile over cumulative clients count will be quantitatively assessed using reliability models including Goel-Okumoto, S-shaped, and Musa-Okumoto SRGMs. The goodness of fit to the actual data will also be examined.

The prediction step will use the selected models to predict future defect behavior. The models will use 75% of the clients’ count and associated defect observations as training data to make reliability prediction into the future and to test the models’ prediction accuracy.

4.5. Case Study

We chose Google Maps APIs as a case study since it is one of the mature, well developed, and widely used cloud services. The case study will demonstrate the applicability and effectiveness of our proposed approach.

4.5.1. Background and Data Availability

One of the earliest cloud services that provide geographic and location information is
Google Maps APIs [67]. According to Google, “it is a collection of APIs that enable you to overlay own data on customized Google Maps”. Frequently, Google is updating, adding, and terminating types and versions of APIs. However, each version has its own updates and it is backward compatible. Google introduced Google Maps as a website only in February 2005. In June 2005, Google Maps APIs was announced for public use. Nowadays, Google Maps APIs supports different environments using several programming languages. The three main environments are Android for smart devices, iOS for Apple devices, and Java-Script for web browsers. This case study focuses on Java-Script Version 3 (JS3) since it is the most popular and well developed in contrast to the other APIs.

Google does not offer specific information about usage or number of subscribers on a daily basis. Therefore, we estimate the daily usage by using a web analytics site called BuiltWith.com that provides general information about various services. One of its services is a usage statistical reports for thousands of web technologies including Google Maps APIs usage statistics\(^1\).

4.5.2. Defect Characterization

This stage in the framework has three steps as shown in Figure 4.1. First, we extracted defects information. Google Maps APIs uses a web application system called gmaps-api-issues\(^2\) for reporting and tracking all defect and enhancement requests by customers. We analyzed the collected data to exclude unrelated issues. We focused on the following fields in issue reports:

**ID:** Each issue has a sequence ID, which was given at the time of filing the report by a client.

**Type:** The issue type can be a defect or an enhancement. Defect type is an issue that causes a failure to the system or disables a functionality. Enhancement type is a request to

---

\(^1\) Google Maps API usage statistics. Accessed on 09/27/2015

\(^2\) Google Maps API bug reports and feature requests. Accessed on 09/27/2015:
https://code.google.com/p/gmaps-api-issues
add a new functionality. This latter issue type does not affect the reliability of the service, therefore it was excluded.

**Status:** This field has fifteen different categories. Each issue that was reported by a client will start as a “new” status. Then it will change from one to another status by Google’s team until it is closed. Table 4.1 contains all status types and our explanation based on the team response.

**API Type:** It includes Java Script, Java Script v2, Java Script v3, Android2, IosSDK, and other languages that support different environments. This case study covers Java Script v3 only. Therefore, all other API types are excluded.

**Time Open:** When the report is issued or opened. This field will allow us to link the defect to the usage in reliability modeling.

The second step is classifying records according to defect validity to three categories: Valid, Invalid, and Uncertain. The classification process uses the status as an indication for the provider treatment for each issue as shown in Table 4.1. Table 4.2 shows each class with its collection of statuses and the number of observations in the data.

The final step is to weigh each defect according to its classification. All valid issues were included, while invalid issues were excluded. We used the percentage of valid defect issue to invalid defect issue as a weight for each uncertain issue. According to Table 4.2, 524 issues are valid, while 1214 issues are invalid. The weight of uncertain issues is $\frac{524}{524+1214} = 0.30$. Therefore, uncertain issues were weighted by 0.30 corresponding to the ratio of valid issues in the past.

To examine our weighting process for uncertain defects, we followed up on all uncertain issues six months after the initial investigation period. 171 of the 346 uncertain issues have been resolved, where 45 issues became valid and 126 became invalid. The result shows that the ratio of invalid issues (73%) to valid issues (27%) is very close to our estimation (70% vs. 30%).
Table 4.1. Google Maps APIs issue status

<table>
<thead>
<tr>
<th>Status</th>
<th>Num</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>116</td>
<td>just posted, no action taken</td>
</tr>
<tr>
<td>Accepted</td>
<td>60</td>
<td>team has accepted as bug</td>
</tr>
<tr>
<td>Acknowledged</td>
<td>3</td>
<td>team is aware of this issue</td>
</tr>
<tr>
<td>Cannot reproduce</td>
<td>172</td>
<td>team can’t reproduce the same bug, so it is closed</td>
</tr>
<tr>
<td>Confirmed</td>
<td>27</td>
<td>team understands the bug and it remains open</td>
</tr>
<tr>
<td>Duplicate</td>
<td>220</td>
<td>team merged the bug with other ones and closed it</td>
</tr>
<tr>
<td>Fixed</td>
<td>433</td>
<td>team fixed the bug and closed it</td>
</tr>
<tr>
<td>Fixed not released</td>
<td>1</td>
<td>team fixed the bug and closed it but not released yet</td>
</tr>
<tr>
<td>Invalid</td>
<td>495</td>
<td>team sees no problem since there is a workaround</td>
</tr>
<tr>
<td>Needs more info</td>
<td>160</td>
<td>team needs more details before it is confirmed</td>
</tr>
<tr>
<td>Obsolete</td>
<td>173</td>
<td>team sees that it is not an issue anymore because of new updates</td>
</tr>
<tr>
<td>Pending further review</td>
<td>70</td>
<td>team leaves it for future review</td>
</tr>
<tr>
<td>Post elsewhere</td>
<td>94</td>
<td>team sees it as not related to this type of API or it is a browser issue, so it is left open</td>
</tr>
<tr>
<td>Won’t fix</td>
<td>15</td>
<td>team can’t fix this bug due to an unsupported browser or internal and external constraints</td>
</tr>
<tr>
<td>Working as intended</td>
<td>45</td>
<td>team sees it not as a bug and it works as it should be, so it was closed</td>
</tr>
</tbody>
</table>

4.5.3. Workload Characterization

The next stage in our framework is workload characterization, which contains three steps. We extracted the number of clients that accessed Google Maps API JS3 in the first step. Since API JS3 is offered for website environment, we need statistics about the number of subscribers to this API or the number of websites that embedded Google Maps in their pages during the investigation period. As stated earlier, BuiltWith.com provides statistical information about technology usage. One of their statistics is a report showing the number of websites using Google Maps APIs, as shown in Figure 4.2.

The second step is to estimate the clients daily usage for the service. Figure 4.2 shows the number of websites using Google Maps APIs from May 2013 to April 2015 with some repetitions. The repetitions appeared in several periods such as from July 2013 to October
Table 4.2. Issues classification & defect weight

<table>
<thead>
<tr>
<th>Class</th>
<th>Status</th>
<th>Total</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>accepted, acknowledged, confirmed, fixed, and fixed not released</td>
<td>524</td>
<td>1</td>
</tr>
<tr>
<td>Invalid</td>
<td>cannot reproduce, duplicate, invalid, obsolete, post else where, won’t fix, and working as intended</td>
<td>1214</td>
<td>0</td>
</tr>
<tr>
<td>Uncertain</td>
<td>new, need more inof, and pending further review</td>
<td>346</td>
<td>0.30</td>
</tr>
</tbody>
</table>

2013 and from December 2014 to March 2015. These repetitions are likely caused by a lack of updates. Therefore, we only included first occurrence point and excluded each repeated ones. We observed a linear trend in the results, so we applied a linear regression to estimate the daily usage in Figure 4.3. This regression is fitted with high accuracy, where $R^2 = 0.97$. Then, we used the regression to predict the number of websites in each day from May 2013 to April 2015.

Finally, we calculated the cumulative websites count up to each day according to Equation 4.1 from Section 4.3. The result is the sum of the number of the websites that had access to Google Maps API up to each day in the investigations period. The result is plotted in Figure 4.4.

4.5.4. Reliability Assessment

We begin with an examination of defect behavior by plotting the cumulative weighted defects over calendar time. Each defect will increase the $y$ value according to its weight with respect to the arriving sequence. Figure 4.5 shows the defects behavior over calendar time which contains 428 records with a total of 255.1 cumulative defects from 1-May-2013 to 21-Sep-2015. There is no clear trend of reliability, which can be explained by the fast growth of usage as characterized by Figure 4.4. Therefore, alternative workload measurement instead of calendar time should be used in reliability modeling.

Before applying any reliability model, we used the results from both previous stages to
plot defects behavior over time, plotting weighted defect over the cumulative websites counts in Figure 4.5. Substituting calendar time with cumulative websites usage yields a stable plot that resembles a typical reliability growth curve. The data can be used now in SRGMs to assess and predict the reliability of Google Maps API JS3.

Table 4.3 shows fitted models, estimated defects, failure rate, and $R^2$ value respectively. The estimated defects at the end of the investigation period $N(t)|_{t=T}$ are 454.98, 237.02, and 255.89 using GO, S-shaped, and MO models respectively, while the actual defects are 255.10. Also, we used the slope of the models at last data point $\lambda(t)|_{t=T}$ to estimate the failure rate at that point. It is clear from Figure 4.6 that the service is estimated to have a high reliability using both GO and MO models. Both models have a high goodness of fit to the actual defects where $R^2=0.996$. S-shaped model has $R^2 = 0.969$, fitting the actual data less closely than MO and GO models. This is expected because the service has already passed the learning curve that S-shaped is considering. In other words, the investigation
Figure 4.3. Regression line to estimate the number of websites using Google Maps API period does not cover the learning period since Google Maps APIs have been in the market for a long time.

4.5.5. Reliability Prediction

The prediction step uses 75% of websites count and associated defect observations as a training set to make predictions. Figure 4.7 shows the actual data with the three selected models with a vertical line to separate the training data set from the testing data set. The summary of prediction results is shown in Table 4.4. It includes the fitted model equations for each selected model, number of predicted defects in last data point, and the failure rate $\lambda(t)_{t=T}$ of the model from the last data point. Also, we estimate the failure rate of the actual data by fit linear regression in the last ten data points and use the slope $\lambda$ of this regression as the failure rate for actual data.

Figure 4.7 and Table 4.4 show that the models are underestimating the actual defects.
Therefore, we investigated further and discovered that Google Maps APIs added sixteen new functions during our investigation period. Each newly added function can invite more failures to the system. Therefore, prediction in such cases would underestimate the actual defects.

4.6. Long Term Validation

To provide a long term validation of our approach, we continued monitoring the defect and usage trend for Google Maps APIs JS3 since September 2015, the ending date for our initial case study [13] described in the previous section. The models fitted to the entire period of the previous case study were extended to predict defects and reliability for the new period from September 2015 to August 2016. These predictions were compared to actual defect observations over time.
Table 4.3. Assessment results

| Models | Fitted models                                                                 | $N(t)|_{t=T}$ | $\lambda(t)|_{t=T}$ | $R^2$ |
|--------|------------------------------------------------------------------------------|---------------|---------------------|-------|
| GO     | $\mu(t) = 493.8(1 - e^{-1.16E^{-09}t})$                                     | 254.98        | 2.770E-07           | 0.996 |
| S-shaped | $\mu(t) = 258.3(1 - (1 + 6.6E - 09t)e^{-6.66E^{-09}t})$                     | 237.02        | 1.130E-07           | 0.969 |
| MO     | $\mu(t) = 249.4\ln(1.7E-09t + 1)$                                         | 255.89        | 1.464E-07           | 0.996 |
| Actual |                                                                               | 255.10        | 5.105E-07           |       |

Table 4.4. Prediction results based on models fitted to partial data

| Models | Fitted models                                                                 | $N(t)|_{t=T}$ | $\lambda(t)|_{t=T}$ |
|--------|------------------------------------------------------------------------------|---------------|---------------------|
| GO     | $\mu(t) = 384.3(1 - e^{-1.57E^{-09}t})$                                     | 240.79        | 2.257E-07           |
| S-shaped | $\mu(t) = 206.3(1 - (1 + 8.7E - 09t)e^{-8.7E^{-09}t})$                     | 200.51        | 4.251E-08           |
| MO     | $\mu(t) = 189.8\ln(2.3E - 09t + 1)$                                         | 244.96        | 1.032E-07           |
| Actual |                                                                               | 255.10        | 5.105E-07           |

4.6.1. Defect Characterization

Google migrated issues data to a new system that contains all Google cloud service products issues. However, this new system did not change the concept or the policy of issues tracking system. But it did change the interface and some structures by renaming or adding fields. For instance, issue “Id” was restructured to keep it unique and to avoid repetition with other service issues. Also, “summary” field was renamed to “title”. The most relevant change to the case study is the categories of status which changed according to Table 4.5. Actually, they moved all status data to a new field called “triaged” and reset the pending status to “new”. Then, they used triaged field to extract the old statuses for the resolved ones. Also, they added a new status called “assigned” to distribute the responsibility of issues among Google team members.

We started the defect characterization stage by collecting all issues from the new Google issues tracker system for the new period and used the new field called “component” to filter the result by Google Maps APIs JS3 issues only. Then we classified the issues according to the new validity classes as shown in Table 4.5, providing a new mapping from individual

35
statuses in the new system to our validity classes.

The new period we added included 378 issues. We included all valid issues (78), and excluded the invalid ones (238). For uncertain issues (62), we used the same weight (0.30) we used in the first period. The total of cumulative weighted defects is 109.

Table 4.5. Google Maps APIs JS3 issue status before and after migration with our classification

<table>
<thead>
<tr>
<th>Old Status</th>
<th>New status</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>New</td>
<td>Uncertain</td>
</tr>
<tr>
<td>Accepted</td>
<td>Accepted</td>
<td>Valid</td>
</tr>
<tr>
<td>Acknowledged</td>
<td>Accepted</td>
<td>Valid</td>
</tr>
<tr>
<td>Cannot reproduce</td>
<td>Won’t fix (not reproducible)</td>
<td>Invalid</td>
</tr>
<tr>
<td>Confirmed</td>
<td>Accepted</td>
<td>Valid</td>
</tr>
<tr>
<td>Duplicate</td>
<td>Duplicate</td>
<td>Invalid</td>
</tr>
<tr>
<td>Fixed</td>
<td>Fixed</td>
<td>Valid</td>
</tr>
<tr>
<td>Fixed not released</td>
<td>Fixed</td>
<td>Valid</td>
</tr>
<tr>
<td>Invalid</td>
<td>Won’t fix (infeasible)</td>
<td>Invalid</td>
</tr>
<tr>
<td>Needs more info</td>
<td>New</td>
<td>Invalid</td>
</tr>
<tr>
<td>Obsolete</td>
<td>Won’t fix (obsolete)</td>
<td>Invalid</td>
</tr>
<tr>
<td>Pending further review</td>
<td>New</td>
<td>Invalid</td>
</tr>
<tr>
<td>Post elsewhere</td>
<td>Won’t fix (infeasible)</td>
<td>Invalid</td>
</tr>
<tr>
<td>Won’t fix</td>
<td>Won’t fix (infeasible)</td>
<td>Invalid</td>
</tr>
<tr>
<td>Working as intended</td>
<td>Won’t fix (intended behavior)</td>
<td>Invalid</td>
</tr>
<tr>
<td>Non</td>
<td>Assigned</td>
<td>Uncertain</td>
</tr>
</tbody>
</table>

4.6.2. Workload Characterization

We used the same statistical website (BuiltWith.com) to extract the usage statistics, with the result plotted in Figure 4.8. One noticeable difference between data in Figure 4.8 for the new observation period and that in Figure 4.4 is the lack of clear trend or pattern. Therefore, we used the latest observation data for each day between these points instead of polynomial or other nonlinear interpolations to avoid overfitting [7].

To build the cumulative usage, we used the last cumulative usage we reached in the
previous period and added up the new estimated usage for each day after, with the result plotted in Figure 4.9.

4.6.3. Reliability Modeling

We extended the defect behavior profile for the previous case study. The defects for the new period were added to these from our initial investigation period and plotted over calendar time in Figure 4.10. The defect behavior profile contains 607 records with a total of 364.8 cumulative defects. The same defects were also plotted in Figure 4.11 against the cumulative website count we obtained above.

Figure 4.10 shows that defect behavior over calendar time does not demonstrate a trend of reliability growth, while the same defects over cumulative website count has a clear concave shape signifying reliability growth as shown in Figure 4.11. These observations reconfirm the appropriateness of using the website count as a proxy for cloud service usage in our framework.

Then, we extended the models we fitted to the data from the entire initial investigation period in Section 4.5.4 to the new period as shown in Figure 4.12. The result in Table 4.6 shows the fitted models, defect estimation, and slope. The estimated defects at the end $N(t)|_{t=T}$ are 366.57, 257.37, and 387.77 using GO, S-shaped, and MO models respectively, while the actual defects are 364.8. Also, we used the slope of the models at the last data point $\lambda(t)|_{t=T}$ to estimate the failure rate at that point and compared it with the linear regression slope of the last actual eight defects. It shows that reliability predictions based on GO model conformed well to the actual data. MO model also performed reasonably well.

In section 4.5.5, we discussed that the underestimation of the models using partial data from the initial period is due to adding new functions to the service in the last 25% part of that period. We investigated further and discovered that Google Maps APIs did not add any new functions during the new investigation period. Consequently, this led to accurate reliability predictions in the long term, without suffering the same underestimation problem.

To summarize, the reliability modeling results provide a long term validation of our
Table 4.6. Long term validation results

| Models | Fitted models                  | $N(t)|_{t=T}$ | $\lambda(t)|_{t=T}$ |
|--------|--------------------------------|--------------|---------------------|
| GO     | $\mu(t) = 493.8(1 - e^{-1.16E-09t})$ | 366.57       | 1.479E-07           |
| S-shaped | $\mu(t) = 258.3(1 - (1 + 6.6E - 09t)e^{-6.6E-09t})$ | 257.37       | 5.771E-09           |
| MO     | $\mu(t) = 249.4 \ln(1.7E - 09t + 1)$ | 387.77       | 5.880E-07           |
| Actual |                                | 364.8        | 2.052E-07           |

approach. In particular, our method of defect characterization can be adapted to work effectively after the migration of issue report system for Google Maps APIs and changes to the detailed data fields. Our proxy for cloud service usage, the cumulative website count, can provide appropriate usage measurement for reliability modeling. Finally, the close fit between the model predictions and the actual defect observations provides evidence that our method will provide accurate long term predictions.
Figure 4.5. Weighted defects over time and over cumulative websites
Figure 4.6. Reliability assessment using selected models

Figure 4.7. Reliability prediction using selected models fitted to partial data
Figure 4.8. Google Maps APIs usage for the second period

Figure 4.9. Cumulative website counts for both periods
Figure 4.10. Defect behavior over calendar time

Figure 4.11. Defect behavior over website count
Figure 4.12. Long term prediction
As described in the related work in Chapter 2, APIs technique has been used in a platform and operating systems for decades. There are several APIs elements that practitioners and researchers proposed to improve APIs quality. However, there are few research results that addresses cloud service APIs quality, and fewer empirical studies that show the impact of these elements on APIs quality. This chapter will include an investigation of cloud service APIs quality characteristics and APIs elements in cloud services. Also we propose a framework to collect empirical data from crowd source to measure the impact of APIs elements on quality.

5.1. New Approach

Our new approach includes four parts. First, we identify relevant APIs quality attributes under the cloud service environment. Second, we decompose cloud service APIs to measurable elements. Then, we define metrics to quantify these quality attributes. Lastly, we measure and analyze cloud service APIs quality using existing data source.

5.1.1. Identify Cloud APIs Quality Attributes

The most common quality characteristic used for APIs in different environment is usability [58] [65]. Using ISO definition [38], the usability of cloud service APIs can be defined as to which degree the APIs can be used by client or developer to integrate the service in their system with effectiveness, efficiency, and satisfaction at the programming environment. They divide usability to six subcharacteristics or attributes:

- Recognizability: the degree to which users can recognize whether a product or system is appropriate for their needs.
• Learnability: the degree to which product or system can be used by specified users to achieve specified goals of learning to use the product or system.

• Operability: the degree to which a product or system has attributes that make it easy to operate and control.

• Error protection: the degree to which a system protects users against making errors.

• User interface aesthetics: the degree to which a user interface enables pleasing and satisfying interaction for the user.

• Accessibility: the degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a goal in a specified context of use.

Some of these subcharacteristics are also related to other quality attributes such as reusability, reliability, and performance. We exclude accessibility because of its limited effect on APIs quality and the lack of its use as an APIs quality attribute in literature. We also exclude aesthetics due to its subjective nature and our focus on quantifiable objective metrics in this paper.

5.1.2. Defending Cloud APIs Metrics

To define cloud service APIs metrics, we start by decomposing the available components into small elements that are measurable. First level of decomposition consists of three components: documentation, design, and calling protocol as shown in Figure 5.1. Documentation can be decomposed to a lower level which contains words, code examples, date of publication. Digging deeper into the documentation, code example can be decomposed to smaller elements that provide metrics such as the number of examples, size of examples, comment density. The second top level component is design which can be decomposed to classes then methods. Methods contain elements such as arguments and returned variables. These elements can produce a metrics such as the number of methods per class, number of arguments
for each method, size of return for each method, etc. Finally, calling protocol can be de-
composed into APIs programming languages that provider offers such as REpresentational
State Transfer (REST) using http, Java Script, Ruby, .NET, etc.

Next, we define metrics to measure each proposed attribute of usability using these
decomposed APIs elements. All proposed cloud service APIs quality attributes and its
metrics are shown in Table 5.1.

• Recognizability: how the users identify the functionality of the service is captured by
  naming style of the method [58] [83].

• Learnability: can be addressed by how documentation will provide needed knowledge
to use API. The metrics are the size of documentation captured by the number of words
in the documentation, number and size of examples provided by the documentation,
density of comments in the examples, and documentation aging which occurs when an
API changes but document fails to keep pace with the API [65] [83].

• Operability: can be addressed by the size of conceptual chunk and how easy the struc-
ture of the service is to operate and control. The metrics include the specific languages
supported, the number of the methods in each class, the number of arguments in each
method, and the complexity of method return captured by the number of returned
variables at XML file in most cases [83].

• Error protection: can be addressed by what is making design misunderstood. The
metrics include the number of overloaded functions and naming which can be mislead-
ing [58] [83].

5.1.3. Measurement and Analysis Framework

We propose a framework shown in Figure 5.2 to investigate the impact of API metrics
defined above on the perceived usability by developers from different organizations by mining
crowd source Q&A such as Stack Overflow. We start by finding the most relevant questions
about each API method using appropriate matching techniques. Then we use multivariate
statistical analysis to understand the impact of API metrics on the crowd behavior such as the number of questions, the number of viewers and other variables that might estimate the overall usability linked to the developers’ effort to understand and learn how such method can be used.

There are several variables in crowd source Q&A that might represent the overall APIs usability as perceived by the developers. We assuming that a developer in most cases will not ask a question unless he or she has difficulty to find, understand, or use the method. Based on this assumption, the number of discussions that address such a method reflects the difficulty of that method. Another assumption is that the developer in most cases will view the discussion related to the method that he or she wants to use. Therefore, we define the overall APIs usability as the number of questions and the sum of viewers for all discussions that linked to the method.

However, developers’ perception of usability can be biased by usage. For example, there might be a method with complex design and low quality documentation but it is not used by
Table 5.1. The quality attribute and metrics framework for cloud service APIs

<table>
<thead>
<tr>
<th>Quality attribute</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>recognizability</td>
<td>naming style</td>
</tr>
<tr>
<td></td>
<td>size of documentation</td>
</tr>
<tr>
<td></td>
<td>number of examples</td>
</tr>
<tr>
<td></td>
<td>size of example</td>
</tr>
<tr>
<td></td>
<td>comment density</td>
</tr>
<tr>
<td></td>
<td>documentation aging</td>
</tr>
<tr>
<td>learnability</td>
<td>APIs language type</td>
</tr>
<tr>
<td></td>
<td>methods per class</td>
</tr>
<tr>
<td></td>
<td>method arguments count</td>
</tr>
<tr>
<td></td>
<td>method return complexity</td>
</tr>
<tr>
<td>operability</td>
<td>overloaded method</td>
</tr>
<tr>
<td></td>
<td>misleading naming</td>
</tr>
</tbody>
</table>

most developers. Therefore no question has been asked about it. Or there is a method with a convenient design and high quality documentation but it is used very frequently, resulting in some questions been asked. We solve this issue by normalizing these counts according to the usage or operational profile. The operational profile is a quantitative characterization of the usage of a given software [55]. We can use it to represent each method usage. To estimate operational profile for each method, we used the number of appearances for each method in code repository such as Github. The ratio between method appearances to all methods appearances can be used to estimate operational profile.

Finally, we divide the usage, the number of question and number of viewers, by operation profile to yield questions score and viewers score as our metrics for the overall APIs usability. The result represents a large sample of developers from various organizations who use the investigated cloud service APIs.
5.2. Case study

We chose YouTube APIs as a case study since it is one of the most used cloud services APIs. The case study will demonstrate the applicability and effectiveness of our proposed approach.

5.2.1. Background and Data Availability

One of the most popular cloud service APIs is YouTube APIs. According to programmableweb.com, YouTube is the third most used APIs after Google Maps and Twitter. YouTube APIs contain 50 methods in 19 classes if we exclude error class. It offers eight APIs programming languages include: Apps script, Google APIs Client Library (Go), java, javaScript, .NET, PHP, Python, and Ruby. The documentation of YouTube APIs is associated with individual API methods. Each API documentation includes a brief description of the method, method arguments, method return, examples, and date of publication.

YouTube APIs use Stack Overflow as the primary channel to support their customers.
They state that “Google engineers monitor and answer questions with the YouTube-API, YouTube-data-API, and YouTube-v3-API tags”. Therefore, we assume that most developers will search for additional knowledge at Stack Overflow if documentation did not offer what they need. This assumption is supported by MS survey [65]. Stack Overflow saves discussions data in a table called Post where the type of post field distinguishes between questions and answers. Both types have a score which is an integer number that reader can increase or decrease by one. Only questions have Tag field to index the discussion by programming languages such as Java, SQL, or others, or by services such as YouTube-API, Google-maps, etc. Also, questions can have accepted answers that are selected by who asked the question. On the other hand, answers have parent field to link them to their questions. Stack Overflow offers SQL access which allowed us to extract desirable data.

5.2.2. Data Collection and Measurement

We extracted YouTube APIs metrics and all Stack Overflow discussions that related to YouTube APIs. Then, we matched between API and its discussions as shown in Figure 5.2. We also calculated the overall usability metrics.

We extracted YouTube APIs metrics from its documentation. Size of Documentation is reflected by the number of words in API document excluding examples. Several examples are offered at the end of the document using one or more programming languages. We extracted from each example type of programming language, source lines of code (SLOC), and lines of comments. The metrics we used for the example are sum of SLOC, average for SLOC, comments density witch is the ratio between sum of SLOC and sum of lines of comments, and flag for each language that offered by documentation. Each API receives several arguments and returns an XML result that contains structured variables. We extracted the number of arguments and the number of variables in XML as metrics for this API. However, not all proposed metrics in Section 5.1.2 would be used in this case study. The naming style is very consistent and follows standards and guidelines. Also, The YouTube APIs documentations are updated very frequently, so there are no aging issues with documentation. Finally,
YouTube uses a unique name for each method, so there are no overload issues that can be addressed.

We extracted the discussions data using SQL access from Stack Overflow. Since tags are the distinguish key for the questions that related to YouTube API, we used Google proposed tags to extract all questions and their accepted answers. The result is 19713 questions and 7690 accepted answers for these questions. This data will be used to extract all discussions that addressed every API method.

We considered two scenarios to match between each method and all discussions that address it. First, a developer asked a question about a specific API method. Therefore, we search for questions that have at least one calling protocol syntax including class.method, class().method, class/method, or class− > method. These syntaxes should cover most APIs programming languages that are used for integration. The result was 1486 questions. However, there are 227 questions linked to more than one methods. After some investigation, we found that more than 190 of these questions are addressing more than one method on purpose. Therefore, we kept these records duplicated since they are addressing more than one methods. The second way to link discussions and method is searching in the answers considering that developers do not know the name of the method and they are looking for a functionality that is offered by the service. The result is 474 new discussions that calling protocol syntax appeared in its answers only. The total discussions we have are 1960 discussions that addressed YouTube APIs.

Github is a suitable code repository for YouTube since it is the official resource of open-source projects according to YouTube itself. It has more than 50K of source code files that using YouTube APIs from different domains. Therefore, it can represent a good sample of systems or applications that use YouTube APIs. The number of YouTube method calling syntaxes appeared in Github codes repository is used to calculate the operational profile for each method. The variables we use for analysis are the number of questions for each method and the total number of viewers for each method. We divided each one by the operation profile for its method. The results are our overall usability metrics for questions and viewers:
questions score and viewers score for each method.

5.2.3. Data Summary and Analysis

First, we summarize the metrics in Figure 5.3 in box plots and give the median and mean for each metric in Table 5.2. Figure 5.3 also shows the first and third quartile for the distribution and outliers outside the whiskers.

We also performed correlation analysis for our APIs metrics and overall usability metrics. Table 5.3 contains seven APIs metrics we used in our study, and two overall usability metrics questions score and viewers score.

There are positive high correlation 0.99 between the number of arguments and the number of words. This is expected since more arguments in the method need more explanation. Also, the number of arguments correlates positively with the number of examples at 0.81. This shows that documentation offers more examples when a method has more arguments. Moreover, there is a correlation 0.84 between words and examples. In fact, this is a transitive relation through arguments. In other words, since the number of words is correlated with the number of arguments and the number of the arguments is correlated with the number of examples, then the number of the words is correlated with the number of examples.

Finally, there is a high correlation of 0.84 between the two overall usability metrics we used, questions score and viewers score. This shows that the method with more questions has more viewers. However, the correlation between API metrics and overall usability metrics are relatively low, with the correlation between questions score and returns at 0.72 and all others below 0.38.
Figure 5.3. Data summary for all metrics
### Table 5.2. Median and mean for all metrics

<table>
<thead>
<tr>
<th>Words</th>
<th>Arguments</th>
<th>Returns</th>
<th>Examples</th>
<th>SLOC</th>
<th>Comment density</th>
<th>class size</th>
<th>Questions scour</th>
<th>View scour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median</td>
<td>886</td>
<td>11.00</td>
<td>8.00</td>
<td>6.00</td>
<td>630.0</td>
<td>36.50</td>
<td>3.000</td>
<td>46.67</td>
</tr>
<tr>
<td>Mean</td>
<td>1151</td>
<td>13.08</td>
<td>31.66</td>
<td>6.401</td>
<td>489.6</td>
<td>56.57</td>
<td>3.617</td>
<td>47.12</td>
</tr>
</tbody>
</table>

### Table 5.3. Correlation table between variables

<table>
<thead>
<tr>
<th></th>
<th>Words</th>
<th>Arguments</th>
<th>Returns</th>
<th>Example</th>
<th>SLOC</th>
<th>Comments</th>
<th>Class size</th>
<th>Questions Score</th>
<th>Viewer Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words</td>
<td>1.000</td>
<td><strong>0.995</strong></td>
<td>-0.235</td>
<td><strong>0.844</strong></td>
<td>0.536</td>
<td>-0.085</td>
<td>-0.556</td>
<td>-0.092</td>
<td>-0.159</td>
</tr>
<tr>
<td>Arguments</td>
<td>1.000</td>
<td>-0.305</td>
<td><strong>0.814</strong></td>
<td>0.497</td>
<td>-0.042</td>
<td>-0.587</td>
<td>-0.119</td>
<td>-0.159</td>
<td></td>
</tr>
<tr>
<td>Returns</td>
<td>1.000</td>
<td>0.050</td>
<td>0.222</td>
<td>-0.163</td>
<td>0.536</td>
<td><strong>0.724</strong></td>
<td>0.383</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>1.000</td>
<td><strong>0.804</strong></td>
<td>-0.144</td>
<td>-0.274</td>
<td>0.209</td>
<td></td>
<td>0.171</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLOC</td>
<td>1.000</td>
<td>-0.218</td>
<td>0.197</td>
<td>0.341</td>
<td></td>
<td></td>
<td>0.258</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comments</td>
<td>1.000</td>
<td>-0.180</td>
<td>-0.010</td>
<td>0.207</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class size</td>
<td>1.000</td>
<td>0.426</td>
<td></td>
<td>0.387</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Questions Score</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0.841</strong></td>
<td></td>
</tr>
<tr>
<td>Viewer Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.000</td>
<td></td>
</tr>
</tbody>
</table>

Note: The values in bold indicate a strong correlation.
5.2.4. Predictive Analysis with Tree-based Models

It would be interesting to examine the relationship between our APIs metrics and overall usability metrics. In particular, we would like to establish predication relationships between our API metrics as predictor variables and our overall usability metrics as response variables. Because of the low correlation between these two groups of variables noted above and in Table 5.3, we explored the use of Tree-based Models (TBMs).

TBMs finds the predictive relationship between predictor and response variables. In our case, we would like to know what are the metrics of APIs driving overall usability metrics. There are two response variables that we want to study, and we need to know how they get influenced by APIs metrics. First, we used all seven APIs metrics plus seven language flags to predict questions score in Figure 5.4. The result shows that methods that returning more than 138 variables driving most discussions. After an investigation, we found out that one method out of 50 is returning XML file with 182 variables, and there are 257 out of 1960 discussions that addressed this method. We investigated the usage or the operation profile of this method which might play a role in increasing questions score. We found out that this method has 2% operation profile which is relatively low. This means that methods or APIs with large numbers of return variables and low usage can also make a high impact on quality and overall usability. The second metrics that played an important role in developers questions are the number of arguments and size of examples. Where three arguments is the threshold linked to increase the number of questions.

We applied TBMs with all metrics as predictors and viewer score as response a variable. The result is shown in Figure 5.5. The size of documentation is the major driver for viewers. Documentation that has more than 611 words drives 1561 of 1960 discussions for higher viewer score. We investigated these methods and found out that nine methods have more than 611 words. However, the sum of operation profile for these methods is 35% which is relatively high. Also, the number of arguments is the second driver for viewer score where more than eight arguments is the threshold that increases the number of viewers.

In general, we can tell from TBMs results that complicated design can decrease developer
Figure 5.4. TBM relating cloud APIs metrics to questions score
Figure 5.5. TBM for cloud APIs metrics to viewers score
cognition for API regardless of the usage. Also design metrics such as return and arguments are more related to the score of the questions.

5.3. Summary of APIs Usability Study

Cloud service APIs have become a major way to integrate, use, and share computing resources. High demand for cloud services increases the need to ensure their quality and usability. Cloud service APIs quality need to be addressed by applicable quality attributes and metrics. Also, these APIs quality metrics need to be linked to some overall metrics of usability.

In this chapter, we develop a comprehensive framework to measure and analyze cloud service APIs quality. First, we identify relevant quality attributes applicable to cloud service APIs. Second, we decompose cloud service APIs to measurable elements. Then we define metrics to quantify these quality attributes using decomposed elements. Lastly, we measure and analyze cloud service APIs quality using existing data source from crowd source Q&A.

We applied our framework on YouTube APIs and Stack Overflow to demonstrate its applicability and effectiveness. The results demonstrated the internal link among these metrics and the predictive relationship between the API metrics as a group and the overall usability metrics. The framework will allowed cloud APIs service providers to identify and remedy the weaknesses in their design and documentation.
Measuring quality of cloud services is a challenging problem. Clients and developers using these services in their systems or applications have limited access to data sources and appropriate measurements derived from these data sources. Also, to improve the usability of these services, we need to identify quality attributes and link them to some overall metrics of usability.

6.1. Summary

In this dissertation, we developed a framework to measure cloud service reliability using alternative data sources. This framework consists of three stages: defects characterization, workload characterization, and reliability modeling. Defects characterization uses issue system reports as a defect data source to obtain weighted defects according to their validity after screening, classifying, and consolidating these reports. Workload characterization uses published statistical reports to obtain clients count as a new proxy to estimate usage time or workload and calculate the cumulative usage. Finally, the reliability modeling stage uses SRGMs to assess and predict cloud service reliability using the outcome of the previous stages.

Google Maps APIs was used as a case study to demonstrate the applicability and effectiveness of our new framework. The result of this case study shows a high accuracy for reliability assessment and prediction, which is an indication that weighted defects and cumulative client counts are appropriated substitutes for the unavailable data and related metrics for this environment. The work was published in the paper below:

- A. Bokhary and J. Tian, “Measuring Cloud Service Reliability by Weighted Defects over the Number of Clients as a Proxy for Usage,” 32nd International Conference on
The paper has been selected to appear in a special issue for International Journal of Computers and Their Application where only seven papers were selected out of 53 papers. The study has been extended by a long term validation. We monitored the defect behavior over a longer period compared to the initial period of our case study. The models fitted to the data from our initial case study were extended to provide long term reliability prediction. This prediction is compared to the actual data capturing the new defects and new workload after similar screening and data processing. We observed a close match between the prediction and actual data to provide a long term validation of the applicability and effectiveness of our approach. The paper has been published in the journal as bellow:


Then, we completed a framework to measure and analyze cloud service APIs usability by identifying the relevant quality attributes and metrics. We proposed quality and metrics framework, and analyze overall cloud service usability on crowd source Q&A. Our case study used YouTube APIs as the investigated cloud service, and it used Stack overflow to extract overall usability metrics. This case study was used to demonstrate the applicability and effectiveness of our new approach. The result shows the most influential elements of cloud service APIs on overall usability. The work was published in the paper below:


To summarize the main contribution, we did the following:

- a new framework to measure cloud service reliability using alternative data sources.

- a new quality and metrics framework for cloud service APIs.
• a new approach to analyze overall cloud service usability on crowd source.

6.2. Future Work

To generalize our reliability framework to measure other cloud services, we need to examine the similarities and differences in data availability and application environments before appropriate data sources and related metrics can be obtained and calculated. This framework and its follow up improvements will offer a clear vision to the developers about the reliability of the cloud service before it is used or embedded in their applications. It also should offer a new method for providers to predict the future reliability of their services, which will help them take remedial and proactive actions to improve their services and to reduce cost.

Our quality framework for cloud service APIs using the most common and applicable quality attributes and metrics for cloud service APIs. However, the empirical study framework can be improved to increase the number of related discussions with higher accuracy, and to extract other overall usability metrics.

However, there are other cloud service quality attributes that should be considered in future to improve cloud service. Although security is one of the fastest growing research areas, it needs more research on cloud service especially for a critical organization such as governments, finances, and health care. Scalability and maintainability are other motivations for business to use cloud services and improving these attributes will play an important role to attract more businesses.

In conclusion, cloud services became part of many systems and applications and used by all type of organizations and customers. The high demand on cloud service also demands improved quality. Therefore, this research addressing this important problem has a high impact on the society. It ensures the satisfaction of clients and customers and the effectiveness and efficiency of the services.


[41] JAFFAL, W., AND TIAN, J. Defect analysis and reliability assessment for
transactional web applications. In Software Reliability Engineering Workshops


[43] KITCHENHAM, B., AND PFLEEG, S. L. Software quality: the elusive target

[44] KORU, A. G., AND TIAN, J. Comparing high-change modules and modules with the
highest measurement values in two large-scale open-source products. IEEE


measurement for web software and source contents. Journal of Systems and Software


quality. volume i. concepts and definitions of software quality. Tech. Rep. TR-77-369,

800-145, 2011.

[51] MOSQUEIRA-REY, E., ALONSO-RIOS, D., MORET-BONILLO, V.,
FERNÁNDEZ-VARELA, I., AND ÁLVAREZ-ESTÈVEZ, D. A systematic approach to
API usability: Taxonomy-derived criteria and a case study. Information and Software

on Software Engineering SE-1, 3 (Sept 1975), 312–327.

[53] MUSA, J. D. Progress in software reliability measurement. In Proc. 2nd Software Life

[54] MUSA, J. D. Validity of execution-time theory of software reliability. IEEE

[55] MUSA, J. D. Operational profiles in software-reliability engineering. IEEE Softw. 10,
2 (Mar. 1993), 14–32.


