

Quality of Information Matters: Recommending Web Services for Performance and Utility

Zheng Song, Owen Rowader, Zhengquan Li, Maryam Tello
Department of Computer and Information Science
University of Michigan at Dearborn
Dearborn, Michigan
{zhesong,orowader, zqli, maryamt}@umich.edu

Eli Tilevich
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
tilevich@cs.vt.edu

Abstract—Widely used in modern software systems, web services have become a standard means of provisioning remote resources. As the number of available web services increases, multiple services that satisfy the same functional requirement can be used interchangeably. Given a set of interchangeable services, a software developer needs to find a web service that would provide the best performance and utility. However, web services are recommended based only on their system-related performance characteristics (so called QoS, whose properties include latency, reliability, availability, etc.), while their data-related performance characteristics (e.g., data freshness, correctness, coverage, etc.) are often overlooked. As a consequence, a recommended service may end up delivering information that is inaccurate or outdated, but with high performance. To address this problem, this paper introduces Quality of Information (QoI), a quality metric complementary to QoS, that measures to which degree a web service satisfies data-related non-functional requirements. To minimize the manual effort required to evaluate the results of invoking individual services, we introduce a comparative testing methodology based on the new concept of Objects of Interest (OI). By using OI, developers can normalize the relevant information obtained from dissimilar services, so it can be automatically compared. To concretely realize our ideas, we create QiSR, a system that recommends web services based on their QoI metrics. QiSR helps developers in determining how to match services’ input and output with application data requirements and how to measure the information quality of services. To evaluate the effectiveness of QiSR, we test it on representative manually selected web services. Our evaluation shows that services recommended based on both QoI and QoS exhibit better combined performance and utility than services recommend on QoS alone.

Index Terms—Quality of Information, Web Service, Comparative Testing

I. INTRODUCTION

Having been introduced more than twenty years ago, web services remain the primary state-of-the-practice approach for applications to request data or functionality from cloud-based servers over the Internet. RapidAPI [1] and ProgrammableWeb [2], the two most popular web service markets, each provides 30K and 24K registered web services, respectively, to serve over 400,000 developers [3]. Many of these web services provide similar functionalities. For example, the RapidAPI team manually cluster 516 set of APIs that provide similar functionalities [4]. Examples of these collections include: 19 APIs [5] related to real-time or historical flight

data, 52 APIs [6] related to image processing and facial recognition, 14 APIs related to sending emails and validating email addresses [7]. Some of these APIs provide equivalent functionalities and can be used interchangeably.

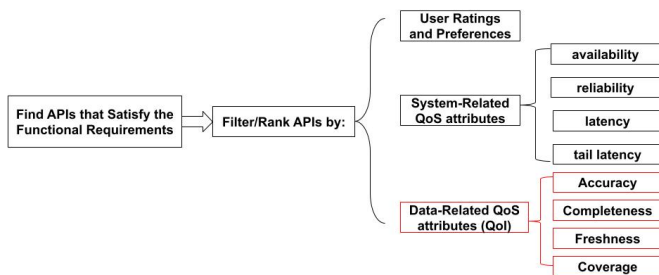


Fig. 1. Web Service Recommendation Workflow

Software developers are facing the challenge of finding the most suitable services that fit the application requirements. As shown in Fig. 1, existing web service recommendation systems first find the services that satisfy functional requirements, and then filter or rank them by features that fall into two major categories: 1) system-related Quality of Service (QoS) attributes, including reliability, responding time, availability, and cost [8]–[10], 2) user’s ratings and preferences [11]–[13].


Image	API	Results
	Microsoft-face1	disgust
	Faceanalysis	anger
	Luxand-cloud-face-recognition	disgust

Fig. 2. Results Given by Face Recognition APIs (accessed in Feb 2022)

Through a preliminary study, we found that equivalent web services may return different results for the same query. For example, face recognition services may return different results for the same image (see Fig. 2), real-time stock price tracing services may return different prices for the same stock, and Amazon product search services may miss some products. Such differences reflect the services’ diverse **data-related**

QoS attributes, in terms of their accuracy, data freshness, and coverage. Although such data-related QoS attributes can make a significant impact on the user's experience, they are totally neglected by existing server recommendation approaches.

This problem is likely caused by the following reasons: 1) software developers mix non-functional performance characteristics of web services with Quality of Service, a concept inherited from network performance measurement. In the mindset of most software developers, QoS refers naturally to the system-related performance characteristics, including reliability, latency, and availability. Lacking a specific term to describe the data-related performance characteristics of web services, developers may find themselves unable to reason about this phenomenon. As an analogy, the Himba people are known for their inability to differentiate between green and blue, as their language has no word that describes the color blue [14]; 2) Compared with system-related performance metrics, data-related metrics are harder to measure. QoS can be easily measured by invoking web services periodically using a few selected input values and recording the invocation latency and result status. Measuring data-related performance characteristics require not only to generate inputs for web services that cover the software developers' interest domain, but also to analyze the results returned by web services and decide whether they are correct and up-to-date.

To solve these problems, this paper introduces Quality of Information (QoI), a concept complementary to QoS that measures to which degree a web service satisfies data-related non-functional requirements. We define a few QoI characteristics, and describe some of their features. To help developers measure QoI for their application scenarios, we design and implement QiSR, a semi-automatic recommender system for web services. QiSR's design is motivated by the facts that 1) if the results of several web services reach a consensus, their correctness will not impact the recommendation results; 2) if the results are different, their comparison can sometimes help the developers to decide which result is better. We evaluate QiSR on three sets of web services, and our results confirm its effectiveness.

This paper makes three contributions: (1) it identifies the problem of overlooking data utility of web services and solves this problem by recommending services based on the new metric of QoI; (2) it presents a new procedure for systematically measuring the QoI of web services; (3) it describes the design, implementation, and evaluation of a recommender based on QoI. The rest of this paper is organized as follows. Section II introduces the technical background and briefly summarizes existing approaches for recommending web services. Section III demonstrates the problem via running examples, defines QoI and its characteristics, and discusses the observed interesting properties of QoI. Section IV presents the design of QiSR and our procedure for measuring QoI. Section V presents our evaluation, and Section VI concludes.

II. BACKGROUND AND RELATED WORK

We first cover the main technical concepts used in this research as well as the prior state of the art.

A. Web Service Marketplace

Originally, a service-oriented architecture included a web service registry as "a phone book" of web services. However, in modern deployments, large-scale web service registries have been replaced by human-maintained web service marketplaces. Using a marketplace, a service developer uploads detailed information about their services, including description of service functionalities, input and output parameters, invocation code examples, and invocation charges.

Application developers search a marketplace for the required functionalities by using keywords. Given a list of search results, developers select one service that best fits the requirements and subscribe to it, so their application can invoke the service. All service invocations are routed through the marketplace, which serves as a gateway for delegating the invocations. The marketplace first invokes a service from its provider's platform, and returns the results to the client. The marketplace also records the number of invocations and their performance (including latency, availability, and popularity), publicly displaying this statistics for each hosted service. A service's popularity score is a function of the number of invocations as well as the number of its active users over a period of time.

B. Web Service Recommendation

With service marketplaces hosting over 30K web services, application developers need effective tools for finding the most suitable web services for different application scenarios. Service recommendation also finds suitable components for service mashups, which chain existing services to provide new functionalities. Existing approaches for recommending web services can be broadly classified into three basic categories:

a) Content-based: Content-based recommendation approaches analyze service descriptions and other related content to identify services whose content is most relevant to the specified keywords. For a given keyword, the TF-IDF model [15] calculates how frequently it appears in a service description. To improve clustering accuracy, Woogole [16] discovers a small number of latent function factors (LFFs). To improve recommendation accuracy, another approach [17] uses additional information generated by users (e.g., service tagging) along with service descriptions.

b) QoS-based: QoS-based recommendation approaches analyze the QoS of preselected services and identify the ones with the highest QoS attributes. Existing approaches study how to monitor services' QoS [8], [9] and how to predict services' QoS based on historical data [10], [18], [19]. We observe that although some papers define QoS as a multi-dimensional performance metrics, most of them actually only consider reliability and latency in their evaluations. This observation also applies to service marketplaces, which display only the latency and availability(service level) QoS attributes.

c) *User Preferences based*: The user preferences-based approaches assume that application developers know already which web services work better for their applications. For example, a developer may choose to use web services with high data accuracy, whenever the veracity of data is critical to meet the requirements, such as in biomedical applications [12]. Developers often share this domain knowledge with each other by means of user rating [13] or user feedback [11]. We observe that the service pages of RapidAPI also display user rating as popularity, which is calculated by the number of invocations and the number of active users.

Data quality assessment has been studied by [20]. The authors introduced several metrics to measure data quality, which include accessibility, amount of data, believability, completeness, conciseness, consistency, correctness, interpretability, objectivity, relevancy, reputation, security, timeliness, and understandability. However, to the best of our knowledge, no prior work has explored how to recommend web services based on their data quality.

C. Comparative Testing

Comparative testing—comparing software products with similar functionalities—has been applied to evaluate face detection algorithms [21], find the correct system behavior on given test inputs [22], and analyze the performance of deep learning models [23]. Given multiple products with the same functionality as candidates to accomplish a specific task, testers rank these products to choose the most suitable ones in the testing context. However, to the best of our knowledge, no prior work has explored how to comparatively test web services for data quality.

III. QUALITY OF INFORMATION

In this section, we demonstrate the necessity of Quality of Information by examples and then formally define this concept.

A. Problem Motivation

Consider the following examples: the first example comes from the domain of the Amazon keyword search, and the second one comes from the domain of language translation.

Assume that a user needs to search for products on Amazon by keywords. Multiple APIs provide this functionality, so we select three top-rated APIs, as recommended by the RapidAPI search engine: Amazon Data [24], Amazon Price [25], and Amazon23 [26]. Given the keyword “*fidget toys*” as input, the aforementioned APIs return the information that is summarized in Fig. 3. Each API returns a number of results, with each of them being an Amazon item that features a unique ASIN (Amazon Standard Identification Number), its price, whether it is a prime product, its number of user reviews, and its average user review rating, as shown in Fig. 4.

We notice three major differences in the obtained results:

- 1) The number of results: Amazon23 gives 44 results, Amazon-data gives 18 results, and Amazon-price

Input : “ fidget toys”

Amazon23	Amazon-data	Amazon-price
<ul style="list-style-type: none"> • 44 Results • ASIN: B07F6G3F1D, 1st • ASIN: B086CBQD2M, 36th 	<ul style="list-style-type: none"> • 18 Results • ASIN: B07F6G3F1D, 3rd • ASIN: B086CBQD2M, 7th 	<ul style="list-style-type: none"> • 47 Results • ASIN: B07F6G3F1D, 22rd • ASIN: B086CBQD2M, 1th

Fig. 3. List of Amazon Search Results

Amazon23	Amazon-data	Amazon-price
<ul style="list-style-type: none"> • asin: B07VLKMMJ5 ◦ Price: 0 ◦ Prime: False ◦ Rating: 4.7 ◦ Reviews: 18717 	<ul style="list-style-type: none"> • Asin: B07VLKMMJ5 ◦ Price: 130.79 ◦ Prime: True ◦ Rating: 4.7 ◦ Reviews: 18717 	<ul style="list-style-type: none"> • Asin: B07VLKMMJ5 ◦ Price: \$119.99 ◦ Prime: true ◦ Rating: 4.7 ◦ Reviews: 18718

Fig. 4. Amazon Item Information (accessed in Feb 2022)

gives 47 results. The difference in the numbers of these results reflects how complete the data of these APIs are.

- 2) The ranks of result items: All these APIs include two Amazon items with ASINs B07F6G3F1D (marked in green) and B086CBQD2M (marked in yellow). However, these three APIs rank the two results quite dissimilarly.
- 3) For the same item (i.e., ASIN B07VLKMMJ5 in Fig. 4, the three APIs give different prices, prime statuses, and numbers of reviews. These differences reflect how fresh the data provided by these APIs are.

The second example shown in Fig. 5 is about translating an English sentence into Spanish. For two input sentences I am Tom and Hello, world!, the results given by The-best-translator differ from those of the other two APIs. Yo soy Tom and Soy Tom for the first input are both correct, while Hola, Mundo! given by The-best-translator for the second input is wrong.

	Google-translate1	Google-translate20	The-best-translator
I am Tom =>	Yo soy Tom	Yo soy Tom	Soy Tom
Hello, world! =>	¡Hola Mundo!	¡Hola Mundo!	Hola, mundo!

Fig. 5. Translation API Results (accessed in Feb 2022)

These two examples demonstrate that APIs providing seemingly the same functionality may in fact differ quite a lot in the quality of their results.

B. Quality of Information

Among all QoS attributes, only “accuracy” is related to the quality of data, while the remaining attributes (latency, tail latency, reliability, availability, etc.) mostly come from the networking domain as a common means of measuring system performance. To address this problem, we introduce the concept of “Quality of Information” (or QoI for short) that measures how well an API satisfies data-related non-functional requirements. Based on our observation, we divided QoI attributes into the following four major categories:

- 1) **Accuracy**: measures the quality of the results being correct or meeting the user’s expectations. Accuracy can

be both subjective or objective. For example, the face recognition results (Fig. 2) and the language detection results (Fig. 5) have the only correct answers. On the contrary, how the order of the Amazon search results (Fig. 3) fits the user’s expectation is more subjective.

- 2) **Completeness:** measures the level of missing information in the results. For example, the number of the Amazon search results (Fig. 3) reflects the completeness of these APIs. Another example is finding the number of faces in an image. Assume all faces are identified correctly, so the number of faces in the results reflects the completeness of the API’s face identification algorithm.
- 3) **Coverage:** measures how an API reacts to different inputs. For example, some COVID statistics update APIs may not have data for certain zip codes; another example is, some language translation APIs may fail to give back results when certain Spanish characters are included in the query.
- 4) **Freshness:** measures how often a data-related API updates its data from the data source. For example, the stock price APIs give different real-time stock prices when being invoked simultaneously; the Amazon item’s details are different as well (Fig 4).

C. QoI Characteristics

Based on our observation, we summarize several characteristics of QoI and QoI measurement:

- 1) It is harder to measure QoI than QoS. Both QoS and QoI reflect how well an API satisfies a service’s non-functional requirements. Measuring QoS only requires invoking an API and recording the response time and status code, while measuring QoI requires generating meaningful inputs and qualitatively assessing the invocation results.
- 2) QoI depends on the service users’ contexts. If a particular user of a language translation service will never need to translate Spanish, then the defect caused by certain Spanish characters should not impact QoI for that user. Similarly, some face recognition service may be more accurate for high-resolution images or photos taken in brighter environments. When measuring QoI for recommending services, it is important to generate inputs similar to the users’ real execution context.
- 3) Compared with measuring precise QoIs for each and every API, a practical way for recommending services is to compare the QoI of equivalent APIs. Instead of manually validating each invocation result of these APIs, we only want to focus on assessing the dissimilar results. If all APIs reach a consensus on the results of an input, this data point would not impact the QoI comparison of the APIs.

These observations have motivated us to design and implement QiSR, a tool that helps API users to compare the QoI of equivalent APIs for selecting the most suitable API for their application scenarios.

IV. MEASURING QOI VIA COMPARATIVE TESTING

This section describes the design and detailed implementation of QiSR, a tool for recommending equivalent web services based on their respective QoI.

A. Objects of Interest

When web services provide equivalent functionality, their input and output parameters can be heterogeneous. The source of heterogeneity may come from multiple angles:

- **Parameter Names:** Parameters that have the same meaning can be named dissimilarly by different service providers. For example, Amazon Price names the input search keyword as “*keywords*”, Amazon Data “*keyword*”, and Amazon23 “*query*”.
- **Parameter Formats:** Parameters that have the same meaning can be differently formatted. For example, Amazon Data formats the `price` parameter as a number, while Amazon23 as an array of {“before price, current price, currency, discounted, saving amount, saving percent”}.
- **Parameter Meanings:** Parameters that seemingly represent the same information can have dissimilar representations. For example, *faceanalysis* (Fig. 2) returns 7 categories: “*angry*”, “*disgust*”, “*fear*”, “*happiness*”, “*sadness*”, “*surprise*”, and “*neutral*”, while *microsoft-face1* also returns the category of “*contempt*”.

To be able to invoke such heterogeneous web services, so their results can be meaningfully compared, QiSR’s design is based on the concept of “Objects of Interest” (OI for short). An object of interest is similar to a database record which also features one primary key and one or multiple attributes. It is an atomic data object that QiSR parses from the output of web services and uses for subsequent comparison and ranking.

For example, for the face recognition APIs in Fig. 2, we define an OI as {`image:PK, emotion`}. For the Amazon search APIs in Fig. 3 and Fig. 4, we define one OI “*Result List*” as {`keyword:PK, result amount, result 1, result 2, result 3, result 4, result 5`}, and another OI “*Item Details*” as {`ASIN:PK, price, prime, rating, reviews`}.

We apply the following principles when defining an OI:

- An OI should have one primary key. The primary key can be the input to web services, or it can be the unique identification of the objects to compare. For example, the *Item Details* is an OI that we parse from the results of a keyword based search, with its primary key (ASIN) being the unique identification of the Amazon product that is not related to the service input.
- An attribute field cannot be an array or a list. If an attribute can only be presented by an array, then it should become a separate OI. In the Amazon search example, we break the results of one round of API invocation into two OIs — *Result List* and *Item Details*.
- An attribute field can be a set of predefined constants (i.e., *enum* type). However, we need to project the different *enum* definitions in the results returned by different APIs

to one set of predefined constants. For example, all emotions can be classified into a combination of six basic emotions, including fear, anger, joy, sadness, disgust, and surprise [27]. We convert the returned emotions into a six-dimensional vector, use the one with the highest accumulative weight as the main emotion, and compare the main emotion returned by different APIs (see Fig. 6).

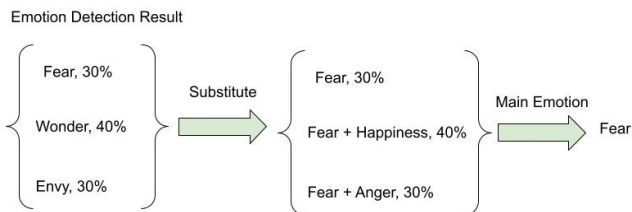


Fig. 6. Normalizing the Emotion Attribute

B. General Workflow

Fig. 7 shows the general workflow of QiSR that proceeds in the following 6 steps:

- 1) Specifying OI and connecting them to the selected equivalent services. In this step, the user defines OI as a JSON configuration file, defines lambda functions that connect each OI attribute to the input/output parameters of web services, and defines the comparison function for each OI attribute. For example, the user can define an OI for face emotion recognition services as `{Image:PK, emotion}`. The emotion will be normalized from the service invocation results by a lambda functions specified in Fig. 6. Each service may require its dedicated lambda function, as its returned emotion categories may be named differently. The comparison function for the only attribute `Emotion` can be: `majority voting + human validation` (See Sec. IV.C for details).
- 2) Generating OI Primary Keys. In this step, the user specifies a dictionary of input values (image URLs in our example) to generate OI objects. If the user wants to invoke each web service 100 times, 100 OI objects will be created from the dictionary that contains the provided 100 image URLs.
- 3) Invoking Web Services. In this step, QiSR invokes the web services for each OI object. The corresponding lambda functions for converting OI to service specific input parameters are applied before each invocation.
- 4) Processing Outputs. QiSR applies the lambda functions associated with each web service on the service's invocation result, to create OI replicas. As the example has three services to compare for each of the 100 OI, there will be 300 OI replicas.
- 5) Comparing the results. QiSR applies the comparison functions on the corresponding OI attributes of services' OI replicas. For example, the `majority voting + human validation` comparison function applies the majority voting first in order to find the OIs that are not

agreed upon by web services. If two services agree on one result and the other service returns a different result, the first two services will be marked as correct while the last marked as incorrect. If three web services give three different results, human operators will be asked to manually grade the results.

- 6) Reporting the final scores of web services. The final score of each web service is calculated as $s = \sum_{a \in \mathcal{A}} w_a * s_a$, where \mathcal{A} denotes all OI attributes, w_a denotes the weight of the OI attribute a , and s_a denotes the score a receives from its comparison function.

To summarize, QiSR takes as input a JSON file that contains: 1) OI definition; 2) lambda functions for converting OI attributes to/from the input/output parameters of each web services; 3) lambda functions for comparing the same attribute of multiple OI replicas of various web services; 4) a dictionary for generating OI primary keys. After executing the aforementioned steps, QiSR outputs the QoI ranking of the web services by comparing their final scores.

C. Result Comparison Functions

We provide several comparison functions as built-in modules of QiSR, including majority voting, sorting, ground truth, and human validation.

- 1) The majority voting module can take in as the parameter the acceptance rate r , whose default value is 0.5. When more than r percentage of web services have the same result, the result is marked as correct and the other results are marked as incorrect. If no result is agreed upon by more than r percent of services, the human validation procedure takes over.
- 2) The sorting module takes the sorting order (ascending or descending) as input. For example, the number of user reviews shown in Fig. 4 will only increase with time. Hence, if one web service's result features a larger number of reviews as compared with other services, we mark the service as fresher than others.
- 3) The ground truth module takes as input the ground truth value d of all OI objects and the tolerance scope e . The module marks a service result as correct if it falls into the range of $[d - e, d + e]$.
- 4) The human validation module asks the user to manually mark the correct results for each OIs. The user interface is shown in 8.

Instead of using these built-in modules, users can also define their own result comparison functions and pass them to QiSR via a custom configuration.

D. Reference Implementation

We implement QiSR in almost 3,000 lines of Python code, written on top of Restler [28], an open source tool for fuzz testing web services. Restler takes as input the swagger/OpenAPI specification of the web services under test, and QiSR further takes as input a configuration file that contains the OI specification and other lambda functions. QiSR interacts

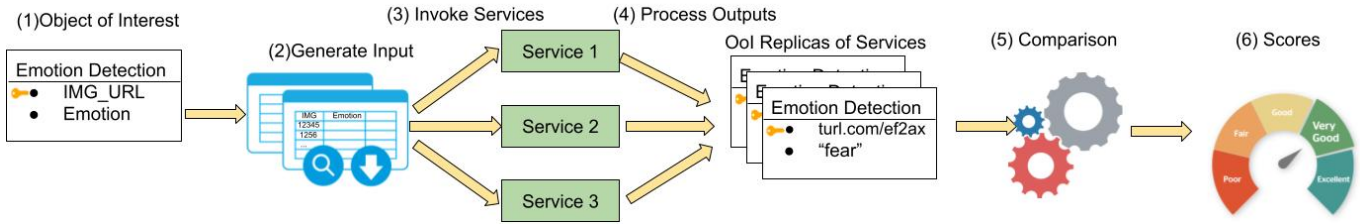


Fig. 7. Workflow of QiSR

```
Judging APIs for Inputs:
email: phbalichef@indo.net.id
0 : True
1 : False
Enter choice: 0
```

Fig. 8. Human Validation Workflow

```
Using Majority Voting:
amazon23 had the best price: 1.0 percent of the time.
amazon-data had the best price: 0.6324324324324324 percent of the time.
amazon-price had the best price: 1.0 percent of the time.

Using reviews as the basis for judgement:
amazon23 had the most reviews, or was tied for the most reviews: 0.9942468619246861 percent of the time.
amazon-data had the most reviews, or was tied for the most reviews: 0.9697292972929729 percent of the time.
amazon-price had the most reviews, or was tied for the most reviews: 0.9957142857142857 percent of the time.

Using majority voting to judge the output objects per input:
amazon23's objects were judged best, or tied for best: 0.98 percent of the time.
amazon-data's objects were judged best, or tied for best: 0.2 percent of the time.
amazon-price's objects were judged best, or tied for best: 1.0 percent of the time.
```

Fig. 9. Execution Result Demonstration

with the user via a command line interface. Fig. 9 demonstrates the results of an example execution of QiSR.

V. EVALUATION

Our evaluation seeks answers to the following questions:

- Does QiSR save programmer effort in assessing the QoI of equivalent web services?
- How do the results returned by QiSR compare in terms of accuracy to those produced via a manual assessment?
- How do service recommendations differ when they are based on QoS vs. QoI? Which recommendations better meet the programmer’s expectations?

For our evaluation, we select typical use cases of using web services and demonstrate how QiSR can be applied for QoI-based service recommendation.

A. Use Case 1: Language Detection

We manually pick five APIs that provide the functionality of language detection: take as input a sentence and output the language it is written in. Table I gives the information of these APIs (text-processing-1¹, language-detection-2², detect-language3³, language-detection4⁴, and natural-language-detection⁵) as presented on their RapidAPI web pages.

¹rapidapi.com/webknox/api/text-processing-1/

²rapidapi.com/detectlanguage/api/language-detection-2/

³rapidapi.com/simontribus1996/api/detect-language3

⁴rapidapi.com/symanto-symanto-default/api/language-detection4/

⁵rapidapi.com/spekulatus/api/natural-language-detection-nld/

TABLE I
QOS OF LANGUAGE DETECTION APIS

API Name	Popularity	Latency	Reliability
text-processing-1	6.1	206 ms	100%
language-detection-2	8.2	277 ms	100%
detect-language3	4.6	2873 ms	100%
language-detection4	7.6	461 ms	100%
natural-language-detection	4.8	698 ms	100%

We choose the majority voting module with the default 50% threshold as the result comparison function. We randomly pick 100 sentences and use them as input to run QiSR. Among all of these 100 OIs, only one OI requires human validation: two APIs agree on one result, two APIs agree on another result, and the remaining one API returns a third result; therefore, no result is agreed upon by more than 50% of all APIs. Hence, we conclude that QiSR can indeed save programmer effort in assessing the QoI of web services. Rather than analyzing the results of executing all 100 OIs by hand, a programmer would need only to manually examine the result of one OI, for which the automatic assessment cannot arrive to a definitive result.

We manually examine the correctness of all the OI results. One of the co-authors of this paper served as a human validator. This person is an upperclassman majoring in Computer Science, without any commercial software development experience, but with working knowledge of Spanish, French, and German. The validator reports the following observations:

- QiSR is incorrect for only one OI. For the input sentence of “Ridicule!”, detect-language3 detects it to be “French”, natural-language-detection returns “undetermined”, while the other three APIs return “English”. By applying majority voting, detect-language3 and natural-language-detection are marked as incorrect. However, “Ridicule!” can be both English and French.
- Language-detection-2 and Language-detection-4 return null results for 8 French sentences and 2 Spanish sentences. After carefully looking into the pattern of these sentences, we found that these two services cannot correctly process “\u00E9”, the Unicode character of the Latin small letter e with an acute accent (é).
- Natural-language-detection and text-processing-1 have the highest error rate. They misidentify Spanish, French, English, and German as other languages quite a lot.

The overall scores (S) of these web services, as graded by QiSR and the human validator are given in Table II. We

observe that the scores given by QiSR are similar to those given by the human validator, with an average difference of 1.2%. We also observe that the user preferences for web services (popularity) are related to both QoI and QoS. For tp1 (text-processing-1): although it has the best QoS, it is ranked the 3rd in user preference due to its lowest QoI ranking among its peers. For dl3 (language-detection3): although its QoI is 12% percent better than the second best service (ld2), its latency is 10 times larger than that of ld2, making its user preference the lowest. By introducing QoI, we can answer the question of why some services with high QoS may not be rated very highly by its users.

TABLE II
QOI, QoS, AND USER RANKING OF LANGUAGE DETECTION APIS

	S (QiSR)	S (Human)	Ranking by QoI	QoS	User
tp1	63%	65%	5	1	3
ld2	88%	89%	2	2	1
dl3	99%	100%	1	5	5
ld4	88%	88%	3	3	2
nld	71%	69%	4	4	4

B. Use Case 2: Amazon Product Search

We created 100 keywords as inputs for web services that search for Amazon products. The keywords range from daily products like “LED lights”, “candy”, and “coffee maker” to more complicated phrases including “mother day gifts”, “LED lights for bedrooms”, and “maxi dresses for women summer”.

As discussed above, we use two OIs (Result List and Item Details) to measure the QoI of the results. The first OI is graded by the number of the search results, while the second OI is graded by the amount of reviews, as deleting reviews happens rarely on Amazon.

Based on QiSR’s outcomes, we find that amazon23 gives the most search results 88% of the time, compared with 8% for amazon-data and amazon-price each. The percentages add up to more than 100%, as sometimes two or three services return the same number of results. amazon23 has the most reviews for 97.1% of the times, compared with 92.9% for amazon-data and 99% for amazon-price. Overall, we recommend amazon23 or amazon-price to our users based on their QoI assessment results, which is in line with the user preferences for these APIs. As shown in Table III, although amazon-23 (A23 in the table) and amazon-price (AP) have comparable latency, amazon-23 is rated higher by the users. The reason for these differences can be explained by the respective QoIs of these services (i.e., number of search results and data freshness in the table).

TABLE III
QOI, QoS, AND USER RANKING OF AMAZON SEARCH APIS

	# of results	Freshness	latency	reliability	popularity
A23	88%	97.1%	4556ms	100	9.7
AP	8%	92.9%	5664ms	100	8.6
AD	8%	99%	1065ms	99	9.7

C. Use Case 3: Emotion Detection

We choose three web services (Microsoft face ⁶, faceanalysis ⁷, and luxand-cloud-face-recognition ⁸) for detecting emotions. We randomly pick 100 photos from the CK+ Dataset [29], which contains photos with expert-annotated emotion labels. We choose the majority voting function to compare their results.

As shown in Table IV, Microsoft face and Luxand are graded by QiSR to be correct for all 100 OIs, while faceanalysis is graded to be correct for 81 OIs. We compare all results with the expert-annotated labels and confirm that QiSR’s results are all correct. We also manually look into the errors of faceanalysis and find that the results given by faceanalysis are pretty close to the ground truth. For example, microsoft-face and Luxand label an image as disgust, while faceanalysis labels it as anger, but it would be hard for anyone but an expert in emotion analysis to decide which label describes the image better. From Table IV, we also observe that, although faceanalysis (FA) shows better QoS performance than Luxand in terms of both reliability and latency, its popularity is a bit lower than that of Luxand. The reason is that in terms of QoI, faceanalysis is 20% lower than Luxand.

TABLE IV
QOI, QoS, AND USER RANKING OF EMOTION DETECTION APIS

	QoI	latency	reliability	popularity
MS Face	100%	1842ms	100	9.2
FA	81%	1008ms	98	8.3
Luxand	100%	1766ms	92	8.6

D. Discussion

Based on the quantitative results and observations in the use case studies, we revisit the questions above:

RQ1: Does QiSR save human effort in assessing the QoI of equivalent services?

QiSR has reduced the human effort in all use cases. In case 1, only 1 OI out of 100 required a manual resolution, while cases 2 and 3 required none.

RQ2: How do the results returned by QiSR compare in terms of accuracy to those produced via a manual assessment?

QiSR has provided highly accurate results, as confirmed by the manual validation for case 1 and the comparison between QiSR’s results and emotion labels. It would be impractical to verify the voluminous results of Amazon product search in case 2, QiSR’s recommendations correspond to the reported popularity of these services. QiSR’s accuracy is affected by the rationale behind which result comparison function to use. For example, to achieve high comparison accuracy by using majority voting, the web services under test have to provide highly accurate results, independent of each other. Otherwise, QiSR may mark as correct an incorrect result agreed upon by

⁶rapidapi.com/microsoft-azure-org-microsoft-cognitive-services/api/microsoft-face1

⁷rapidapi.com/promityai-promityai-default/api/faceanalysis

⁸rapidapi.com/aboykov/api/luxand-cloud-face-recognition

two low-accuracy web services.

RQ3: How do service recommendations differ when they are based on QoS vs. QoI? Which recommendations better meet the programmer's expectations?

In all three cases, the QoS rankings of the evaluated services differ by a large margin from their popularity, as based on their actual usage. The popularity of a service is determined by how well the service satisfies the requirements, both in terms of performance and utility. This finding strongly motivates the need for QoI, as QoS-based rankings differ greatly from those based on popularity. With QoI and a tool like QiSR in place, programmers can better understand and measure the data-related performance and utility of web services.

VI. CONCLUSION

This paper presents QiSR, a comparative testing tool for evaluating the Quality of Information of web services. We define QoI as a concept complementary to QoS, that measures to which degree a web service satisfies data-related non-functional requirements. QiSR facilitates the measurements of QoI by defining Objects of Interest and specifying lambda functions that convert web services' input/output parameters to OI attributes and compare these attributes. We apply QiSR on three sets of manually selected web services and our evaluation showed that user's rating on web services can be better explained by considering both QoS and QoI.

VII. ACKNOWLEDGEMENT

This research is supported by NSF through the grants #2104337, #2203825, and # 2232565 as well as by the Summer Undergraduate Research Experience Program of UMDearborn.

REFERENCES

- [1] "Rapidapi - the next-generation api platform," <https://rapidapi.com/>, accessed: 2022-03-30.
- [2] "Programmableweb - apis, mashups, and the web as platform," <https://www.programmableweb.com/>, accessed: 2022-03-30.
- [3] "Sneak peek: New api provider dashboard," <https://rapidapi.com/blog/new-api-provider-dashboard-sneak-peek/>, accessed: 2022-03-30.
- [4] "All api collections (rapidapi)," <https://rapidapi.com/collections>, accessed: 2022-03-30.
- [5] "Flight data apis," <https://rapidapi.com/collection/flight-data-apis>, accessed: 2022-03-30.
- [6] "Top image processing and facial recognition apis," <https://rapidapi.com/collection/top-image-recognition-apis>, accessed: 2022-03-30.
- [7] "Best email apis," <https://rapidapi.com/collection/email-apis>, accessed: 2022-03-30.
- [8] Z. Zheng, H. Ma, M. R. Lyu, and I. King, "Qos-aware web service recommendation by collaborative filtering," *IEEE Transactions on services computing*, vol. 4, no. 2, pp. 140–152, 2010.
- [9] X. Chen, Z. Zheng, Q. Yu, and M. R. Lyu, "Web service recommendation via exploiting location and qos information," *IEEE Transactions on Parallel and distributed systems*, vol. 25, no. 7, pp. 1913–1924, 2013.
- [10] J. Zhu, P. He, Z. Zheng, and M. R. Lyu, "A privacy-preserving qos prediction framework for web service recommendation," in *2015 IEEE International Conference on Web Services*. IEEE, 2015, pp. 241–248.
- [11] Y. Liu, A. H. Ngu, and L. Z. Zeng, "Qos computation and policing in dynamic web service selection," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, 2004, pp. 66–73.
- [12] S. Galizia, A. Gugliotta, and J. Domingue, "A trust based methodology for web service selection," in *International conference on semantic computing (ICSC 2007)*. IEEE, 2007, pp. 193–200.
- [13] X. Liu and I. Folia, "Incorporating user, topic, and service related latent factors into web service recommendation," in *2015 IEEE International Conference on Web Services*. IEEE, 2015, pp. 185–192.
- [14] "It's not easy seeing green," <https://languageolog.ldc.upenn.edu/nll/?p=17970>, accessed: 2022-03-30.
- [15] Y. Elshater, K. Elgazzar, and P. Martin, "godiscovery: Web service discovery made efficient," in *2015 IEEE international conference on web services*. IEEE, 2015, pp. 711–716.
- [16] Q. Yu, H. Wang, and L. Chen, "Learning sparse functional factors for large-scale service clustering," in *2015 IEEE international conference on web services*. IEEE, 2015, pp. 201–208.
- [17] L. Chen, Y. Wang, Q. Yu, Z. Zheng, and J. Wu, "Wt-lda: user tagging augmented lda for web service clustering," in *International conference on service-oriented computing*. Springer, 2013, pp. 162–176.
- [18] M. Zhang, X. Liu, R. Zhang, and H. Sun, "A web service recommendation approach based on qos prediction using fuzzy clustering," in *2012 IEEE ninth international conference on services computing*. IEEE, 2012, pp. 138–145.
- [19] Y. Ma, S. Wang, F. Yang, and R. N. Chang, "Predicting qos values via multi-dimensional qos data for web service recommendations," in *2015 IEEE International Conference on Web Services*. IEEE, 2015, pp. 249–256.
- [20] L. L. Pipino, Y. W. Lee, and R. Y. Wang, "Data quality assessment," *Communications of the ACM*, vol. 45, no. 4, pp. 211–218, 2002.
- [21] N. Degtyarev and O. Seredin, "Comparative testing of face detection algorithms," in *International Conference on Image and Signal Processing*. Springer, 2010, pp. 200–209.
- [22] E. G. Sizer and B. N. Bershad, "Using production grammars in software testing," *ACM SIGPLAN Notices*, vol. 35, no. 1, pp. 1–13, 1999.
- [23] L. Meng, Y. Li, L. Chen, Z. Wang, D. Wu, Y. Zhou, and B. Xu, "Measuring discrimination to boost comparative testing for multiple deep learning models," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 385–396.
- [24] "Amazon data," <https://rapidapi.com/magedata/api/amazon-data/>, accessed: 2022-03-30.
- [25] "Amazon price," <https://rapidapi.com/ajmorenelarosa/api/amazon-price1/>, accessed: 2022-03-30.
- [26] "Amazon 23," <https://rapidapi.com/restlyer/api/amazon23>, accessed: 2022-03-30.
- [27] P. Ekman, "Basic emotions," *Handbook of cognition and emotion*, vol. 98, no. 45-60, p. 16, 1999.
- [28] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758.
- [29] P. Lucey, J. F. Cohn, T. Kanade, J. Saragih, Z. Ambadar, and I. Matthews, "The extended cohn-kanade dataset (ck+): A complete dataset for action unit and emotion-specified expression," in *2010 IEEE computer society conference on computer vision and pattern recognition-workshops*. IEEE, 2010, pp. 94–101.