Computers, Materials & Continua
DOI:10.32604/cmc.2021.017783
Article

Tech Science Press

Development of a Smart Technique for Mobile Web Services Discovery

Mohamed Eb-Saad[1], Yunyoung Nam[2,*], Hazem M. El-bakry[1] and Samir Abdelrazek[1]

[1]Department of Information Systems, Faculty of Computers and Information, Mansoura University, Mansoura, 35516, Egypt
[2]Department of Computer Science and Engineering, Soonchunhyang University, Asan, 31538, Korea
[*]Corresponding Author: Yunyoung Nam. Email: ynam@sch.ac.kr
Received: 11 February 2021; Accepted: 22 March 2021

Abstract: Web service (WS) presents a good solution to the interoperability of different types of systems that aims to reduce the overhead of high processing in a resource-limited environment. With the increasing demand for mobile WS (MWS), the WS discovery process has become a significant challenging point in the WS lifecycle that aims to identify the relevant MWSs that best match the service requests. This discovery process is a resource-consuming task that cannot be performed efficiently in a mobile computing environment due to the limitations of mobile devices. Meanwhile, a cloud computing can provide rich computing resources for mobile environments given its unlimited and easily scalable resources. This paper proposes a semantic WS discovery and invocation framework in mobile environments based on cloud and a relationship-aware matchmaking algorithm. The discovery algorithm enriches MWS and user requests semantically with the functional and non-functional properties of Ontology Web Language for Services, such as Quality of Web Service, device context, and user preferences. The WS repository is filtered based on logical reasoning and a parameter-based matching algorithm to minimize the matching space and improve runtime performance. The cosine similarity between the user request and services repository is then assessed to generate the most relevant WS. The relationships among concepts in the ontology are considered to improve the recall and precision ratio. After the WS discovery process, users can invoke and test these services in a mobile environment through a dynamic user interface. The interface of the invocation process is changed according to the WS description document. An application prototype is also developed to evaluate the framework based on a Cordova cross-mobile development framework.

Keywords: Cloud; web service; web service discovery; semantic matching; mobile web service discovery

1 Introduction

Web services (WSs) are software modules that perform specific tasks regardless of their implementation details and are used to facilitate an information exchange among different

applications. Mobile WS (MWS) discovery invokes WSs via mobile devices and is considered a resource-intensive process that cannot be efficiently handled by mobile devices due to their limited capability and resource constraints. Cloud computing can address the gap between the computing constraints of mobile devices and the discovery process by providing these devices with unlimited computing resources and infrastructure in a process called mobile cloud computing (MCC), wherein mobile applications are built, powered, and hosted by using cloud computing technology [1].

WS discovery involves two processes, namely, syntactic discovery and semantic discovery. On the one hand, syntactic discovery [2] uses a keyword matching technique that, despite its easy implementation, lacks depth and has lower precision and recall compared with semantic discovery due to keyword polysemy problems. On the other hand, semantic discovery enriches WS with semantic descriptions or ontologies, such as the Ontology Web Language for Services (OWL-S), Semantic Annotation Web Services Description Language (SAWSDL), and WS Modeling Ontology (WSMO) [3]. However, semantic matching requires more time compared with syntactic matching and is considered a heavyweight process that involves semantic reasoning [4] and the generation of the required ontology files. Nevertheless, semantic discovery methods are generally more efficient than syntactic discovery ones [5]. In this paper, we propose a cloud-based MWS discovery framework by using a semantic matchmaking algorithm and an invocation module that invokes WSs dynamically after the discovery process in mobile environments. We implemented a mobile application prototype to evaluate our framework. Our semantic discovery approach considers functional (e.g., input and output parameters, effect, and prediction) and non-functional properties (NFPs), such as Quality of WS (QoWS) parameters (e.g., price, availability, and service rank), device context information (e.g., location, screen resolution, and bandwidth), and user preference, which are critical to the delivery of appropriate services to users with the right QoWS expectations at the right place and time. To improve our proposed matching algorithm, we quantified the semantic relations between the concepts of user requests and the WSs instead of matching the semantic concepts. Our proposed system uses a logical reasoning filtering function that efficiently narrows the searching space of the WS repository and reduces the set of resulting WS that satisfies the user query, thereby improving the runtime performance and effectiveness of the ranking algorithm.

In our framework, we added a dynamic user interface component that allows users to invoke and test the WSs in a mobile device after the discovery process. We changed the interface dynamically according to the WS description file and validated the entered user data before invoking the WS in a mobile device through the validation checker component to reduce invocation failures. We also proposed a JavaScript (JS) frameworks WS generator component that allows users to choose among several JS frameworks, such as Angular, React, VueJS, and JQuery, and return the HTTP invocation code of the selected framework.

This paper is organized as follows. Section 2 discusses the related work. Section 3 presents an overview of the proposed framework and describes its components. Sections 4 and 5 discuss the discovery process and semantic matchmaking algorithm, the prototype implementation, and the evaluation results. Section 6 concludes the paper and proposes directions for future work.

2 Related Work

Discovery approaches based on Universal Description, Discovery, and Integration [6] have many limitations and are not designed to support mobile services, QoS properties, or semantic search [7]. Over the past few years, many studies have attempted to overcome the drawbacks and

optimize the WS discovery process, especially in mobile environments. Given that the semantic discovery of WS is a critical topic in the service science literature, researchers have exploited semantic web technologies to enrich WS with semantic descriptions, such as ontologies (e.g., OWL-S and WSMO). MWS discovery and selection processes need to consider functional and NFPs instead of merely matching input and output parameters. However, while most popular strategies focus on functional properties, NFPs have been relatively ignored [8]. The existing MWS discovery approaches for NFPs in mobile computing environments can be categorized into context-aware, QoWS-aware, and hybrid discovery.

Sangers et al. [9] built a service context comprising a group of keywords obtained from service descriptions and then applied natural language processing (NLP) approaches in a keyword-based discovery process. Elgazzar et al. [10] proposed a WS discovery framework for mobile environments by using WSDL WS description with a SOAP-based architecture and performed syntactic matching to discover services that completely match user requests. However, given that they treated the semantic signature as a collection of concepts and ignored its relation in ontology, their approach could not effectively discover services whenever logical reasoning is required.

Saadon et al. [11] proposed a cloud-based MWS discovery framework called CMDis, which focuses on semantic matching and ranking by using the context information provided by the service requester. However, the precision and recall results need further work because the relations among the concepts are neglected in the matching process. Moreover, they focused on the match-making process and ignored the testing and invocation of the WS after obtaining the algorithm results.

Plavila et al. [12] proposed a cloud-based MWS discovery framework for discovering related MWSs, applied the Wordnet database and text-based matching for the semantic matching of WSs, and added a module for invoking and testing the WSs on a mobile environment after retrieving these services from the matchmaking algorithm. This framework produced a dynamic user interface for testing and invoking based on the service selected by users. However, they did not test the service parameters data before invoking the WS given that users may invoke a service with invalid parameters data type or fail to set any parameters, which leads to a service invocation failure.

Fang et al. [13] proposed a service discovery approach focusing on the ontology of WSs and formulated a filtering scheme that reduces the amount of resulting WSs by exploiting the ontology referenced by requests and services. The refined WSs were matched with user requests by a relationship-aware parameter-based matching algorithm. However, they focused on the match-making algorithm itself and ignored the invocation of the WS or even the functional and non-functional requirements of the user environment.

Our work is similar to [11] given that our proposed system produces a semantic matchmaking and ranking algorithm that is migrated to a cloud to ensure improved performance and rich computational resources [14]. However, this framework focuses on QOWS characteristics and enhances the algorithm similarity measure to improve the recall and precision ratio. This system also focuses on the invocation of WSs in mobile environments similar to [12]. However, our framework solves the problem of validating the service parameters requirements before invoking the WS. We also added a JS frameworks code generator for the selected WS to provide users with the final HTTP invocation request code for the WS in their preferred JS framework.

3 Proposed Framework

In this section, we present the components, processes, and workflow of our framework. The proposed framework is shown in Fig. 1 and for convenience and simplicity, we divided our framework into several components and discussed the role of each component.

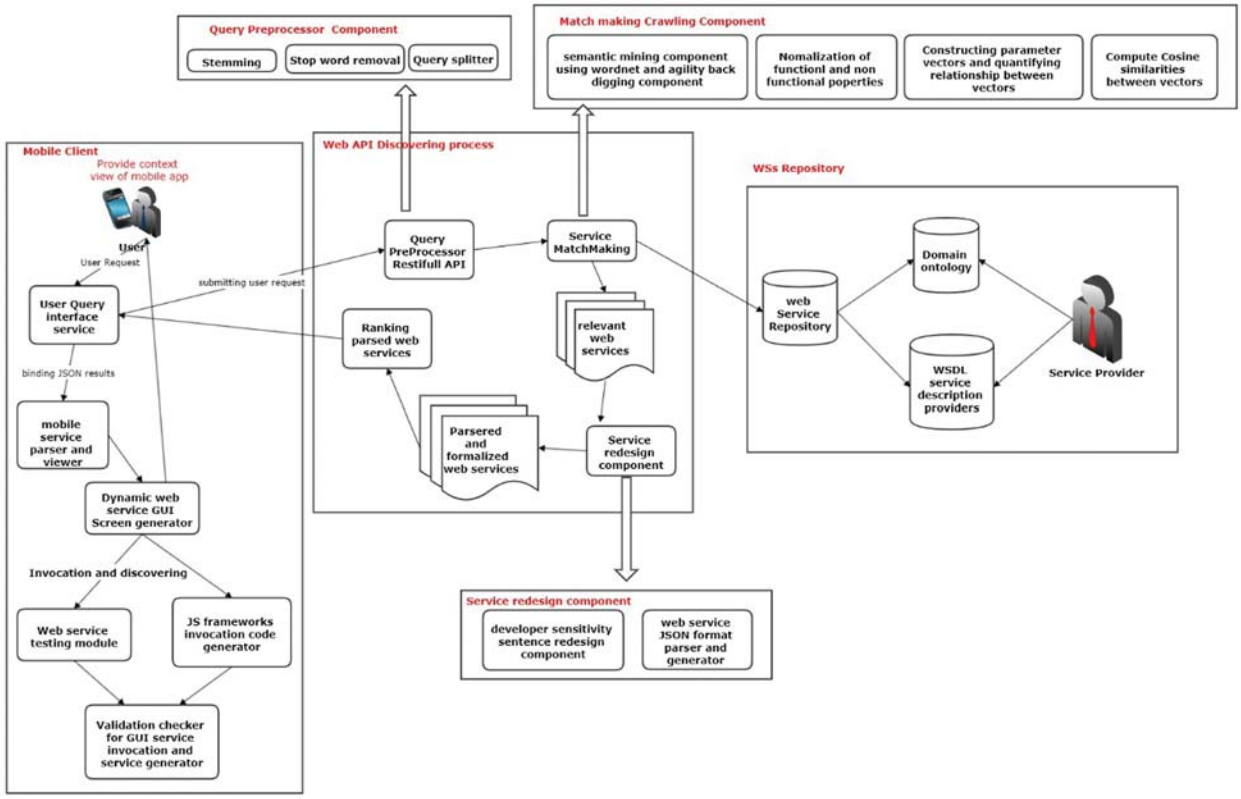


Figure 1: Proposed framework overview

3.1 Service Query Interface

The WS Query Interface is a mobile user interface that reflects the purpose of users in conducting the search process and helps them submit their requests by entering service request keywords in a mobile user interface by using natural language. This interface is designed by using framework7 [15], a free and open-source framework for developing mobile applications with a native look and feel. By using an Angular HTTP request [16], search keywords are sent to a cloud server that uses representational state transfer (REST) [17] as API to process the query via the discovery matchmaking algorithm as will be discussed in Sections 3.4 and 3.5. A list of relevant web services is then sent back to the mobile application after the search process, and the application binds the response list on the mobile interface by using the Angular two-way binding technique. After obtaining the results, users can select, test, and invoke any of the WSs. By pressing the test button, the dynamic WS GUI generator component generates a dynamic interface that allows users to test and invoke WS. This interface is generated depending on the JSON response data returned by REST API. To complete the WS invocation, users should pass the test of the validation checker component of the framework, which evaluates the required

parameters, parameter type, and model of the WS. These components will be discussed in Sections 3.7 and 3.8.

3.2 Query Preprocessor Component

The query preprocessor component aims to develop an interface between the user query and the service matchmaking algorithm. After users send a request written in plain text, the request should be cleaned and preprocessed before applying the service matchmaking algorithm to extract meaningful information. To this end, we followed the steps of some NLP techniques. First, we divided the request query into keyword tokens by using the query splitter component. Second, we applied the stop word removal technique. Stop or poison words, such as a, for, an, is, on, and this, are commonly used in plain text yet are useless in the processing stage. Therefore, these words should be tested and removed from the query. Third, we applied the porter stemming algorithm [18] to reduce the keyword tokens down to their roots. Fourth, we used the Syn.WordNet [19] library for the stemming process. These steps are illustrated in Fig. 2.

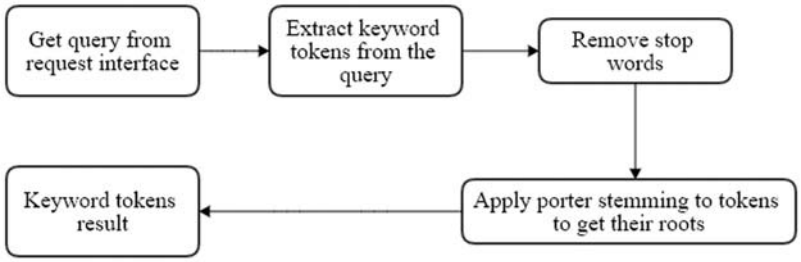


Figure 2: Query preprocessor component

3.3 WS Repository

Many WS descriptions are written in semantic and non-semantic ways. In our framework, we used hREST-TC3 [20] test collection as a WS repository and also applied the Agility back digging component [21] to read and process the hREST files and to obtain the input and output parameters, WS title and description, also WS invocation method. In order to filter similar functionalities of WS descriptions to similar and related categories a service classification component is used which includes several steps of NLP techniques to categories similar WSs in the repository to the same category same as [11].

3.4 Service Matchmaking and Ranking Algorithm

We performed a search optimization of the user keywords by using the WS repository and obtained the search results. Comparing with traditional keyword matching models, our proposed system produces more rich semantic information to increase recall and precision ratio. To provide accurate results for WSs, all information used in this section can be applied to any common model of a traditional WS description. Our proposed matchmaking algorithm is a semantic discovery approach that considers functional properties and NFPs, QoWS, and context attributes. This algorithm is also a relationship-aware algorithm that constructs vector parameter concepts for requests and services and considers the relationship among concepts as an impact aspect. We applied cosine similarity to determine the vector similarities. Our algorithm involves several steps. First, we included functional properties, such as input, output description, preconditions,

effects, and normalization, similar to [22]. We also considered basic information, such as service title, name, category, and description. Second, we extracted NFPs, such as QoWS properties and context information that were extracted from the device profile, mobile environment, and user preference. Context properties are implicit information related to both the service provider and mobile application and warrant consideration given their potential effects on the performance of the returned results [23]. Third, after considering the functional properties and NFPs, we calculated the similarities between vectors and the degree of match (DOM) for all properties. Fourth, we produced WSs with high values depending on the requirements of mobile users. To determine the appropriate WSs, we matched the inputs of functional properties via keyword-and semantic-based matching and treated the outputs as a result of the effect.

The precondition of the functional requirement indicates the condition or specification of WSs that must be satisfied before invoking these services [24]. Suppose two services $S1$ and $S2$. If $S1$ requires weather information from the registry, then the weather information of a particular city will be presented in $S2$. Otherwise, the precondition for $S1$ is not satisfied. The effect in FP denotes the condition of WS that must be satisfied after invoking and completing this service. Suppose three services (S_1, S_2, S_3), which represent money transfer, currency conversion, and weather forecast, respectively. These services have a sample input, output, precondition, and effect. The currency service (WS_2) has Euro currency as its input and the conversion of Euro to dollar as its output. Therefore, the correct currency type is a precondition for the system to convert Euro to dollar. The effect of this successful conversion will be converted into the specified currency (e.g., dollar). These effects should be considered after the currency conversion [22].

In normalization, those values that are measured at different scales of a WS are adjusted to a general scale. The different scales and factors of QoS, such as availability and reliability, are measured in percentages, penalty and price are measured in rupees, response time is measured in milliseconds, and authorization and privacy are measured as either 0 or 1, where 1 indicates authorized or available and 0 indicates otherwise. Therefore, the QoS parameters need to be normalized. The effect of negative attributes also differs from that of positive ones. Given that positive attributes bring positive economic value, increasing their value will improve QoS. By contrast, given negative attributes result in negative economic value, increasing their value will reduce QOS. In this case, positive and negative attributes should be normalized separately.

$$q_p = \begin{cases} \dfrac{q - q_{min}}{q_{max} - q_{min}} & if \;\; q_{max} - q_{min} \neq 0 \\ 1 & if \;\; q_{max} - q_{min} = 0 \end{cases} \tag{1}$$

Normalization of Positive Attributes

$$q_n = \begin{cases} \dfrac{q_{max} - q}{q_{max} - q_{min}} & if \;\; q_{max} - q_{min} \neq 0 \\ 1 & if \;\; q_{max} - q_{min} = 0 \end{cases} \tag{2}$$

Normalization of Negative Attributes

where

- q_{min} and q_{max} denote the minimum and maximum range of each attribute for each service;
- q represents the attribute value of the service; and
- q_n represents the normalized value of the service attribute.

The normalization process [25] produces a list of services whose values of quality factors are expressed in common units. The QoS value of each service can be summed to obtain higher cumulative values.

After normalization, the algorithm creates concept vectors [13] for the input and output parameters. The association among concepts is determined from related terms under the "model-Reference" annotation. The result contains a vector input parameter $(x_1, x_2, \ldots x_n)$ and a vector output parameter $(x_1, x_2, \ldots, x_m)$, where x_i is a concept that represents the parameter, and $inS_i = (x_{1,i}^{inS}, x_{2,i}^{inS}, \ldots x_{m,i}^{inS})$ is the concept of the input vector for the service that has the role. The concept of an input vector of the user request query is $inR = (x_1^{inR}, x_2^{inR}, \ldots x_q^{inR})$, whereas those of the output vectors are $outS_i = (x_{1,i}^{outS}, x_{2,i}^{outS}, \ldots x_{n,i}^{outS})$ and $outR = (x_1^{outR}, x_2^{outR}, \ldots x_t^{outR})$. The algorithm also matches the WSs in the service repository with user requests and eventually produces a set of services that semantically match the query of users. Consider user request R and service profile S. To validate the degree of relevance between S and R, the properties of services (i.e., type, input, output, and contextual attributes) should be matched with the facts in R. The results of the matching process between R and S can be classified into five categories as shown in Tab. 1.

Table 1: Service matchmaking categories [26]

#	Category name	Matching relevance
1	Exact	If the service and the request are equivalent (R = S)
2	PlugIn	If request R is a super-concept of service S (R ⊃ S)
3	Subsume	If request R is a sub-concept of service S (R ⊂ S)
4	Intersection	If the intersection of service S and request R is satisfactory (R ∩ S)
5	Fail	If service S and request R are not equivalent concepts (R ≠ S)

In our algorithm, we quantified the relationship among concepts into categories and assigned weights to the relationships among concepts. The matching algorithm constructs input and output matching graphs by using a weighted bipartite graph [27] $G = (V + V', E)$, where V denotes inS and V' denotes inR. The vertices of $e(e \in E)$ belong to V and V', and an edge is present between each pair of vertices belonging to V and V'. Every edge is associated with a weight, which represents the matching degree between two concepts as shown in Fig. 3. G also has a subgraph denoted by Gi, $G_i = (V + V', E_i)(E_i \subseteq E)$, which represents input and output matching graphs. If both conditions exist, then the first condition is satisfied when the vertex cannot be shared by more than one edge, whereas the second condition is satisfied if each element in the vector represents a vertex of an edge. If the constructed graph from R and S contains at least one input matching graph that does not contain "fail" weight edges, then S is chosen. Otherwise, S is avoided. Fig. 3 shows the input matching degree of both the request and service concepts, and Fig. 3b shows both input matching graphs of Fig. 3a. Taking into account that the same conditions rules were applied on both input and output matching graphs, after constructing the input and output matching graphs, we computed the cosine similarity between vectors to match the request with the WS repository. The next section explains this process in more detail.

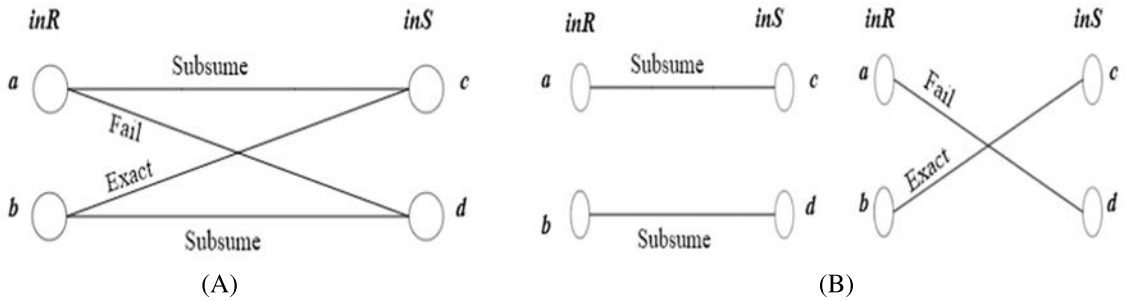


Figure 3: Example on an input matching graph. (a) Example on input matching graph (b) Matching degrees of input requests

3.5 Ranking Parsed WSs

In the matching algorithm, we built parameter concept vectors for both services and requests and used cosine similarity to compute the similarity between two vectors. We also used the device profile and context environment in ranking the listed WSs with additional context information. We applied our algorithm in two steps. In the first step, we selected candidate services according to a parameter-based service matching component for ranking WSs based on the similarities between $inR(i=1,2\ldots M)$ and inS_i and those between $outS_i(i=1,2\ldots M)$ and outR, with M denoting the number of WSs chosen by the service filter. This parameter-based service matching component initially computed the input vector space inV_{space} equivalent to the union of $inS_i=(i=1,2\ldots M)$ and inR as in Eq. (3). We performed the same procedure for the output vector space $outV_{space}$ as in Eq. (4). The dimensionality of the input and output vector spaces is denoted by n.

$$inV_{space}=inR\cup inS_1\cup inS_2\ldots\ \cup inS_{m=}\left(x_1^v,x_2^v,\ldots x_n^v\right) \tag{3}$$

$$outV_{space}=outR\cup outS_1\cup outS_2\ldots\ \cup outS_{m=}\left(x_1^v,x_2^v,\ldots x_n^v\right) \tag{4}$$

In the second step, after generating the input and output vector spaces, we transformed $inR, inS_i=(i=1,2\ldots M), outR$, and $outS_i=(i=1,2\ldots M)$ into weight vectors of N-dimension, where N is the dimensionality of inV_{space} or $outV_{space}$. The dimensionality N of the vector space represents the number of distinct keywords in the $corpusV_{space}=(k_1,k_2\ldots k_N)$, where k_i is a keyword that describes queries and service documents. We used $d_i=\left(w_{1,i},w_{2,i}\ldots w_{N,i}\right), r=(w_{1,r},w_{2,r}\ldots w_{N,r})$ to define the input vector keywords weight of the ith chosen service documents and the request, respectively. The weight takes a value of 0 or 1 similar to the rules in the matching algorithm. A higher value of $w_{j,i}$ corresponds to a higher importance of x_{j}^v to the service. W computed $w_{j,i}$ based on the semantic similarity between concepts given that this parameter is assigned the maximum similarity value of x_{j}^v and $x_{\mathrm{q,i}}^{inS}$. We used cosine similarity for matching queries and documents [28] and to measure the similarity between two vectors of N dimensions. Cosine similarity can also be defined as the dot product and magnitude between two vectors as shown in Eq. (5).

$$cos\ similarity\,(A,B)=cos\,(\theta)=\frac{A.B}{\|A\|\,\|B\|} \tag{5}$$

where A and B are two vectors of n dimensions, and θ is the similarity angle between two vectors. To compute the similarity between two concepts, if these concepts do not belong to the same

ontology, then their similarity is equal to 0. Otherwise, we used Eq. (6) to calculate the relative similarity between the x and y concepts by converting concept similarity into a unit interval.

$$similarity\,(x,y) = \rho\frac{|\alpha\,(x) \cap \alpha\,(y)|}{|\alpha\,(x)|} + (1-\rho)\,\frac{|\alpha\,(x) \cap \alpha\,(y)|}{|\alpha\,(y)|} \tag{6}$$

where $\rho \in [0,1], \alpha\,(x)$ is the set of upward nodes that are reachable from $x.\alpha(x) \cap \alpha(y)$ is a set of reachable nodes shared by both x *and* y and indicates the similarities between these two concepts.

Given the weight of input vectors of the selected request and services ($d_i = (w_{1,i}, w_{2,i}, \ldots w_{n,i})$, $r = (w_{1,r}, w_{2,r}, \ldots w_{n,r})$), by using Eqs. (7) and (8), $w_{j,i} = 1$ when $x_j^v \in inS_i$ and $w_{j,r} = 1$ when $x_j^v \in inR$. We then measured the similarity of the input and output parameters by using Eq. (9).

$$w_{j,i} = maxsim\left(x_j^v, x_{q,i}^{inS}\right) \tag{7}$$

$$w_{j,r} = maxsim(x_j^v, x_p^{inR}) \tag{8}$$

$$Similarity\,(inS_i, inR) = \frac{d.r}{\|di\|\,|\,|r|\,|} \tag{9}$$

In the first step of the ranking algorithm, we calculated the similarity between the service and request by using Eq. (10).

$$Rank1 = Similarity\,(S_i, R) = \theta \cdot Similarity\,(inS_i, inR) + (1-\theta) \cdot Similarity\,(outS_i, outR) \tag{10}$$

where $\theta \in [0,1]$ denotes the user's preference for input and output parameters. We assigned the value measure of similarity to each service and used this measure in the second step of our algorithm. The first step produces a ranking list that narrows the matching space of the service repository by reducing the number of candidate services, hence improving the overall runtime performance. We then ranked this list against the context information extracted from the mobile device.

We used the NFP matching elements obtained from the matching algorithm, such as QoWS and context properties, to determine which WS is appropriate for each mobile device. During the ranking process, we ran a component in the mobile client side to extract context information, such as device profile, user preferences, and environment context. Using such information, we could select the ranked WS properly in consideration of device status. Given that the environment context (e.g., longitude and latitude) and network status may change while the mobile device is running, this device should track such information and treat them. For example, when a bad network exists in our framework, we used the local storage [29] of the mobile device instead of the local variables to store the information retrieved from the server in order for users to deal with the application despite a network failure or store the retrieved data on the rootScope of the Angular framework. Our framework may even send alerts to inform users about the network status, to conserve battery, and to limit the processing capability of mobile devices. We migrated most of the processing to the cloud server to optimize the usage of our framework [30]. As we mentioned before, in the matchmaking algorithm, we divided the QoWS into positive and negative attributes. Therefore, we normalized the QoWS according to Eqs. (1) and (2). The value scale resulting from the normalization phase was in the range of [0, 1]. Fig. 5 shows the QoWS parameter classification. For each relevant MWS where $[MWS = mws_1, mws_2, mws_3, ..mws_i)]$ and context $[C = C_1, C_2, C_3, \ldots, C_j)]$, based on the context information extracted from the context manager, we used a ranking value obtained in the second step of the algorithm to determine the

most precise WS. Afterward, our algorithm constructed the matrix in Eq. (11), which represents the context information for WSs where each row represents a $MWS(mws_i)$ and each column represents the context features for this row.

$$S_0 = \begin{bmatrix} mws_{1,1} & mws_{1,2} & mws_{1,3} & mws_{1,j} \\ mws_{2,1} & . & . & . \\ mws_{3,1} & . & . & . \\ mws_{i,1} & . & . & mws_{i,j} \end{bmatrix} \tag{11}$$

Afterward, we calculated the rank of each MWS mws_i by using Eq. (12) and represented the relationship between a WS request and each relevant WSs by using function f. The value of $f(x, y)$ ranges between 0 and 1 and indicates how value x is relevant to y.

$$Rank2 = \sum w_i * f(MWS_{ij}, C_j) \tag{12}$$

where Wi is the weight associated with each property in the outputted services, and this value is associated with each *mws*. For additional details about Eq. (12), see [11]. In the third step of the algorithm, we computed the average similarity between Eqs. (10) and (12) to obtain the final rank for each service by using Eq. (13).

$$DOM = \frac{(Rank1 + Rank2)}{100} \tag{13}$$

In the last step of our algorithm, we arranged the filtered WSs in a decreasing order according to the rank of similarity determined from Eq. (13) because a larger *DOM* corresponds to a more appropriate WS for the user query. This step outputs the ranked services. The matchmaking algorithm steps are illustrated in Fig. 4.

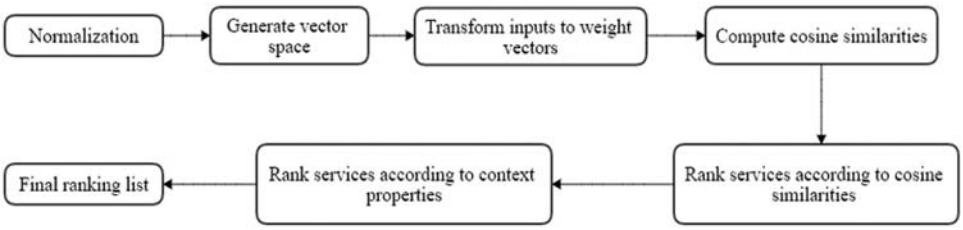


Figure 4: Matchmaking algorithm steps

3.6 Service Redesign Component

The service redesign component is responsible for the final result of REST API. After retrieving the relevant MWSs from the ranking process instead of sending the ranked WS files and crowding the network and client storage with these data and cached files, to avoid performance issues, we translated the results to a well-formatted JSON [31] array that contains all needed information from the WS files. This array also contains information about the WS objects. As shown in Fig. 6, every object in the array contains the name of the WS, URL, and WS invocation method (e.g., POST, PUT, and GET) as strings and another array of WS input and output parameters. We used this object to generate a dynamic mobile user interface based on the WS selected from the users. This dynamic user interface is described in Section 3.7. For now, we still

face a problem in the names of the parameters and the title of the WSs because developers can write the names of parameters in different formats (e.g., camel case, Pascal, kebab, and snake case formats). Moreover, the words are not separated by any spaces. To display these parameters clearly in the user interface, we added a component for word splitting and redesigning the parameters style for the users. The word splitter component splits and rewrites the names of parameters (input, output, and title) of the WS according to a regex expression [32] that redesigns the code styles (e.g., camel, Pascal, or other coding styles) to uppercase words and then splits the joined words with whitespaces. We presented the resulting title name and parameters on the dynamic GUI screen generator for readability as shown in Fig. 6.

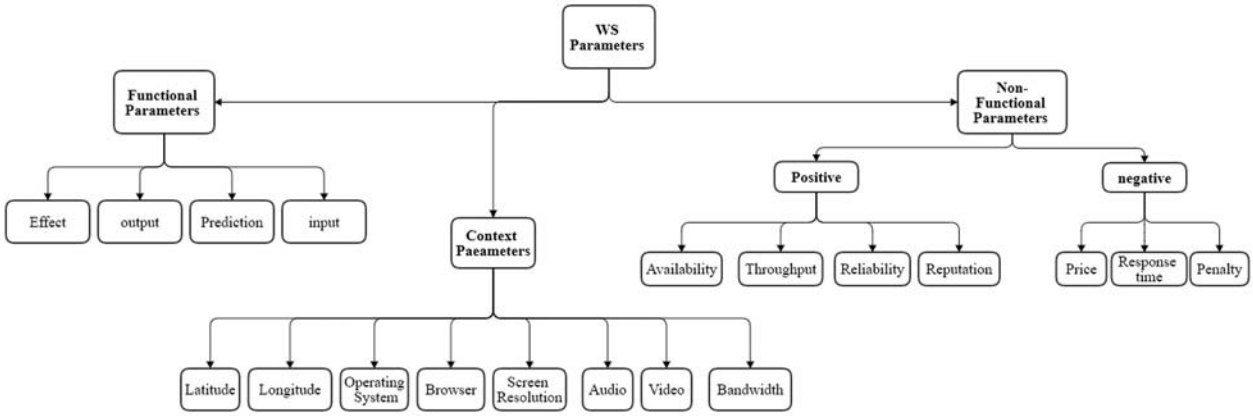


Figure 5: QoWS parameter classification

Figure 6: Returned JSON array after applying the matchmaking algorithm

3.7 Dynamic WS GUI Screen Generator

After presenting the organized JSON results on the application user interface, our application bound these results via two-way binding by using Angular in a table containing the title of the WS and the number of required inputs and outputs as shown in Fig. 7. Users can test any of the returned WSs by pressing the test button. Our application would then generate a dynamic user interface containing textboxes for the input parameters according to the number of inputs in the JSON object of the WS returned by the redesign component. We assigned each input to a label containing the name of the input presented on the service invocation screen (the dynamic user interface) as shown in Figs. 8a and 8b. We changed the user interface according to the WS selected by the user, that is, if these users select a WS with two inputs, the mobile user interface will generate two text boxes for each WS and two labels containing the parameter names assigned to them as shown in Fig. 8a. Meanwhile, if users select a WS with three input parameters, then the application will generate three text inputs for each input parameter. Users may also invoke the WS by entering the required input parameters and then clicking the invoke button to test the WS result. Based on the method type in the WS JSON object of the JSON array returned from REST API, the screen generator checks whether this WS is a POST, Put, or something else. After the invocation test, the output pops up on the screen.

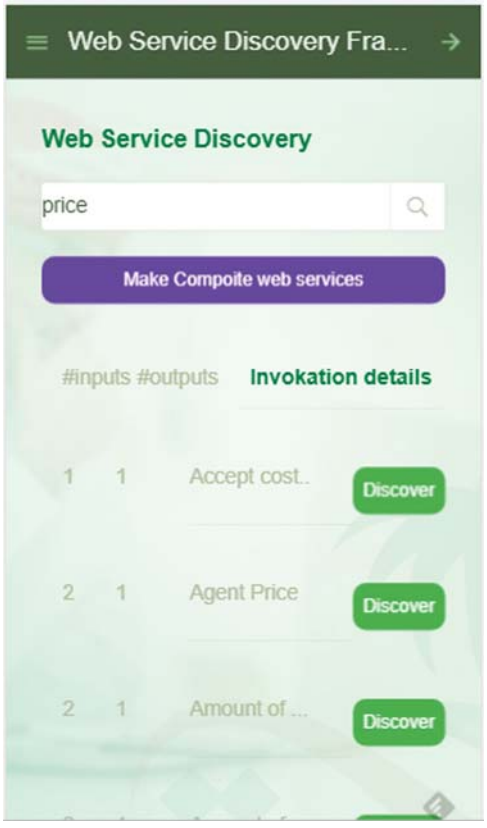
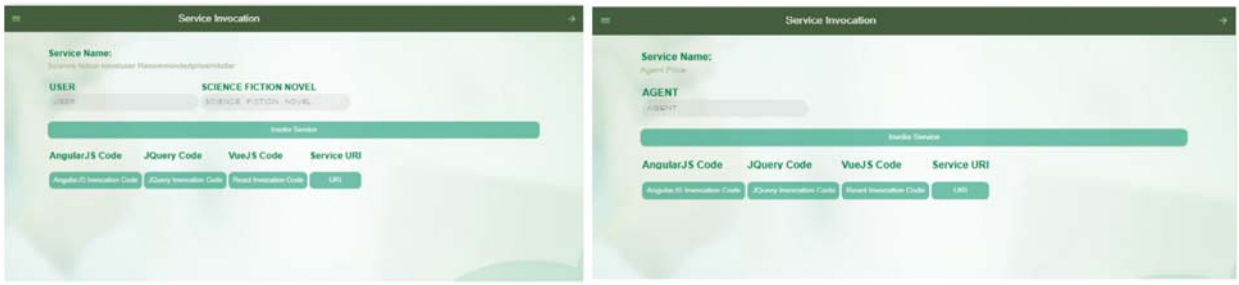


Figure 7: Result of the data binding process

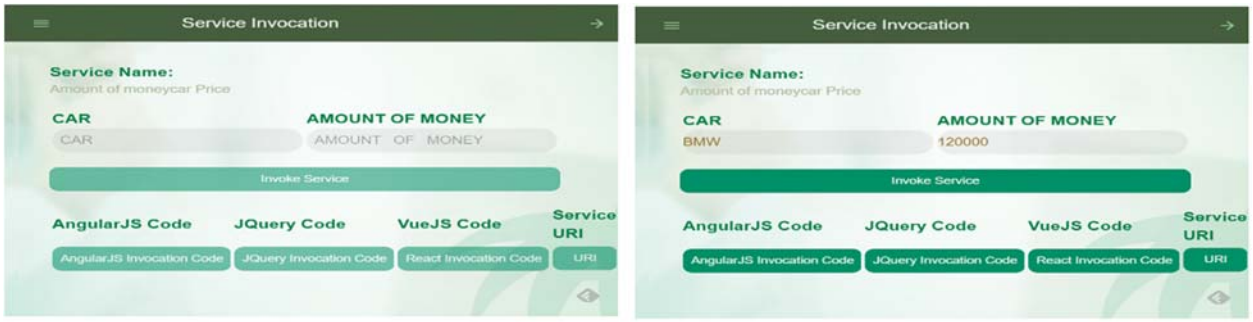


(A) (B)

Figure 8: User interfaces designed by the dynamic WS GUI generator. (A) User interface for a WS with two input parameters generated by the WS GUI generator (B) User interface for a WS with one input parameter

3.8 Validation Checker Component

The validation checker component ensures the reliability of the WS invocation process by checking if the type of input parameters entered by users is valid. This component also checks the required input parameters before invoking the WS. If users fail to submit a required parameter, then they will be notified via message. Our framework handles request failures by ensuring that the WS will never fail. The invocation codes of the JS frameworks are not generated unless the validation checker operations are successful. We created this component dynamically based on the WS object (Fig. 6) of the JSON array within the WS screen generator. Fig. 9 shows that users cannot invoke the WS or retrieve the request codes until the required parameters are submitted because the invocation buttons remain disabled until the user sets valid parameter data.



(A) (B)

Figure 9: Validation checker component. (A) User interface with a validation error (B) User interface with no errors

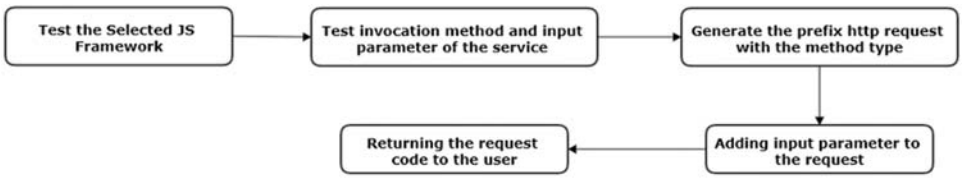
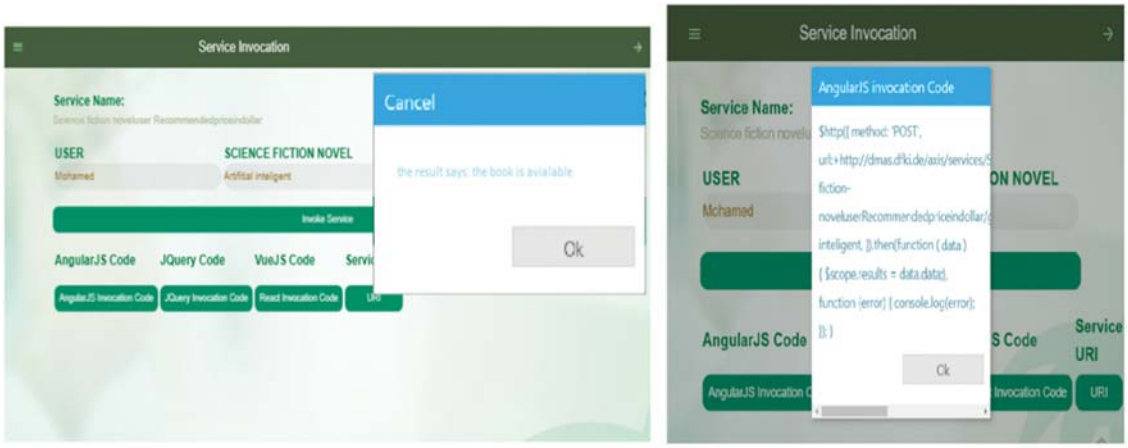


Figure 10: JS service generator

3.9 JS Frameworks Web Service Generator

In this component, users can select a framework or library from any of the JS frameworks offered by our application. The mobile application generates an HTTP invocation request code for the selected WS based on the default HTTP service in the selected JS framework. Examples of these frameworks include JQuery, Angular, React, and VueJS. User can simply copy-paste the invocation request code of the specified WS by using their preferred JS framework in its application or website, and the result will be returned to them. However, the binding process in the application depends on the user; in the outputted request code, users should merely change the input parameter value. This component depends on the JSON object of the WS. In this part, our algorithm tests the method type of the WS (e.g., Post and Put) and then checks its input parameters and data type. Afterward, according to the WS selected by users, this algorithm generates an invocation code for the selected JS framework. The algorithm steps are shown in Fig. 11.



(A) (B)

Figure 11: WS invocation. (a) The output result for the invocation of the web service (b) Angular invocation code from service generator

4 Prototype Implementation

4.1 User Interface

We developed our framework based on the Apache Cordova cross-platform framework [33]. Our framework can be operated on multiple platforms, such as Android, iOS, or Windows Phone. We designed the interface screens of our application by using framework7 [15]. Fig. 7 shows the client interface of the request screen where users can enter search keywords. After entering these keywords, users will be presented a response list that is bound into the table via Angular two-way binding. Figs. 8 and 10 show the dynamic screen generator where one can observe the effect of the service redesign component on the service name and input parameters. Users can well understand the result even though the WS description is written in different developer formats as mentioned above. These screens were taken from an Android device with portal and landscape screen modes.

4.2 Data and Implementation Details

We implemented our project by using the .NET Core framework with C# language and REST API in the backend [34]. We used the Syn.WordNet library for NLP processing and Wordnet support, and employed the AgilityPack [19] library to read and parse the XML files of the hREST-TC3 test collection. In the frontend development, we implemented the application by using the Apache Cordova cross-platform framework in order for this application to run on many mobile platforms, such as Android, iPhone, and Windows Phone. We used framework7 to design the mobile interfaces and used the Angular framework in the frontend processing, binding, and REST API service calls. We deployed our framework on the Amazon EC2 cloud server and conducted an experiment on hREST-TC3 test collection, which comprises 1080 services, 38 ontologies, and 42 descriptions defined in SAWSDL. These WSs contain different fields, including food, communication, medical, economy, education, and travel.

5 Evaluation Criteria

To evaluate the effectiveness of our framework, we analyzed the following aspects in multiple experiments:

- suitability of this framework to different mobile platforms;
- recall and precision of the proposed algorithm; and
- required runtime performance for completing the discovery process.

5.1 Suitability to Mobile Platforms

The framework proposed in [11,12] was implemented in Java, thereby limiting its application to Android devices. By contrast, we developed our framework by using a cross-platform framework to ensure its applicability to all platforms, such as Android, iPhone, and Windows Phone. Moreover, our proposed system uses REST API to process requests. All the processing is performed on a cloud, thereby allowing developers to integrate our framework in their work by calling the REST API service and then bind their results on any platform.

5.2 Accuracy of the Proposed Framework

We analyzed the accuracy of our framework in three steps. First, we tested the accuracy of the parameter matching algorithm, which is a critical component of our framework, by computing the correlation coefficient of its corresponding values. Second, we computed the recall and precision of our proposed framework. Third, we performed a runtime evaluation.

In the experiment, we compared five methods for matching distinct requests. The first method was our matchmaking algorithm, which preprocesses the data before applying the matchmaking and ranking algorithm. The functional and non-functional requirements and the relationship among vectors were considered in this method. The second method is PBSM_R presented in [13], which, similar to our approach, takes the relationship among vectors into consideration. However, unlike our method, PBSM_R does not consider the mobile context environment, the functional properties and NFPs, or the preprocessing steps. The third method is the PBSM algorithm, which examines the cosine similarities between vectors similar to our approach but ignores the vector relationship. The fourth and fifth methods, CMDis and LOG4SWS.KOM presented in [12,35], depend on the standard DOMs, such as Exact, Subsume, and Fail, after mapping the degree to a numeric scale. We measured the correlation of these five methods as presented in Tab. 2.

Table 2: Correlation coefficient for the four methods

Method	In1	In2	In3	In4
Proposed parameter-based matchmaking	0.90	0.89	.84	.97
PBSM_R	0.87	0.85	.76	.96
PBSM	.57	.42	.39	.96
CMDis	0.82	0.80	.73	.94
LOG4SWS.KOM	0.71	0.79	.70	.93

In Tab. 2, In1 denotes instance1, in2 denotes instance2, and so on. Our proposed algorithm outperforms PBSM and PBSM_R because of its preprocessing steps and its consideration of functional and non-functional requirements and context environment. Our proposed algorithm also outperforms CMDis and LOG4SWS.KOM because when quantifying the relationship among concepts, our algorithm achieves high correlation in matching and obtains additional semantic information for improved accuracy.

Second, we applied information retrieval techniques, such as precision and recall. Precision refers to the ratio of the number of relevant WSs retrieved to the total number of relevant and irrelevant WSs retrieved as computed using Eq. (13) [36].

$$precision = \frac{|S_{Relevant}| \bigcap |S_{Retrieved}|}{|S_{Retrieved}|} \tag{14}$$

where $|S_{Relevant}|$ is the number of WSs relevant to the request, and $|S_{Retrieved}|$ is the number of retrieved services. The precision value varies between 0 and 1 and is usually expressed in percentage. Recall denotes the fraction of relevant services obtained from a request and can be calculated as the ratio of the number of relevant services retrieved to the total number of relevant services. Similarly, the recall value varies between 0 and 1.

$$Recall = \frac{|S_{Relevant}| \bigcap |S_{Retrieved}|}{|S_{Relevant}|} \tag{15}$$

In our experiment, we used many service requests from different domains of the dataset to calculate the average precision and recall. We introduced five requests, with each request having average precision and recall. From the evaluation results, we found that our framework had the best precision among the compared methods when the same recall is considered. Moreover, our framework obtained the best recall when the same precision is applied by taking into account the relationship among concepts (similar to the OFPM framework) and considering the preprocessing steps and the functional and non-functional requirements. The evaluation results are presented in Fig. 12.

5.3 Runtime Efficiency

We measured computational time by using the same requests with different dataset sizes for all compared algorithms. The response time of OFPM and our algorithm slowly increased for large datasets because these algorithms narrow the searching space in the matching process. Nevertheless, OFPM outperformed our algorithm in terms of runtime because the latter takes the functional and non-functional requirements of the mobile context into consideration. This result may also be attributed to the preprocessing steps in our framework. However, the differences

between these algorithms in terms of precision and recall were acceptable. Meanwhile, unlike the two other algorithms, LOG4SWS.KOM demonstrated poor performance in large data sizes, which corresponded to an improved runtime performance. The analysis results are shown in Fig. 13.

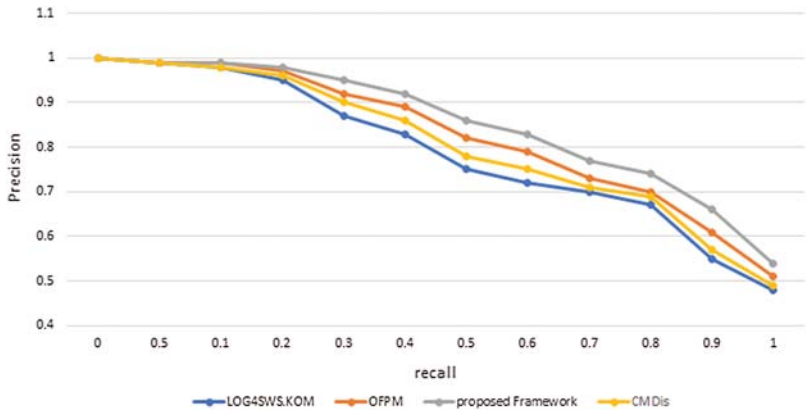


Figure 12: Resulting precision and recall from requests

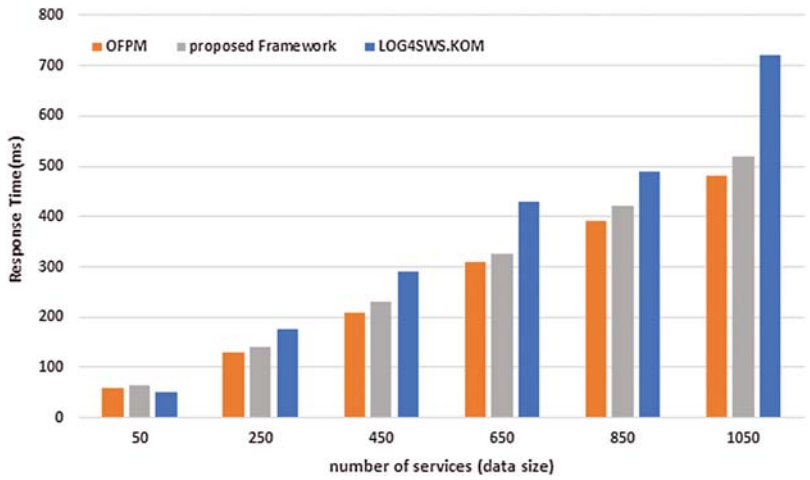


Figure 13: Computational time evaluation

6 Conclusion and Future Work

In this paper, we built a framework for MWSs discovery based on a semantic cloud matchmaking and ranking algorithm. This algorithm semantically enriches services and user requests with functional properties (e.g., input and output parameters, effect, and prediction) and NFPs of OWL-S (e.g., device context, user preferences, and QoWS), which are normalized to negative and positive attributes according to their effects. Our proposed system uses a filtering approach for the service repository matching space to reduce the set of selected WSs and to satisfy the user demand for high runtime efficiency. The relationship among ontology concepts is also considered in the ranking algorithm to obtain more relevant results and increase recall and accuracy ratio.

After retrieving the relevant services from the server and displaying them to users, these users can invoke any service by using the service invocation component that generates dynamic GUI screens. We also included a validation checker component to test and validate the user data requests before invoking a service. We developed our prototype on a Cordova-based cross-platform mobile environment to evaluate our framework. In our future work, we will integrate our framework with a composition planner that selects the most suitable service part from different services to construct a new service with a suitable functionality for users.

Funding Statement: This research was supported by X-mind Corps program of National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (No. 2019H1D8A1105622) and the Soonchunhyang University Research Fund.

Conflicts of Interests: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] N. El-Rashidy, S. El-Sappagh, S. M. Islam, H. M. E. Bakry and S. Abdelrazek, "End-To-End deep learning framework for coronavirus (covid-19) detection and monitoring," *Electronics*, vol. 9, no. 9, pp. 1439, 2020.

[2] F. Chen, C. Lu, H. Wu and M. Li, "A semantic similarity measure integrating multiple conceptual relationships for web service discovery," *Expert Systems with Applications*, vol. 67, no. 3, pp. 19–31, 2017.

[3] B. Sheng, C. Zhang, X. Yin, Q. Lu, Y. Cheng *et al.,* "Common intelligent semantic matching engines of cloud manufacturing service based on OWL-S," *The International Journal of Advanced Manufacturing Technology*, vol. 84, no. 1–4, pp. 103–118, 2016.

[4] S. El-Sappagh, J. M. Alonso, F. Ali, A. Ali, J.-H. Jang *et al.,* "An ontology-based interpretable fuzzy decision support system for diabetes diagnosis," *IEEE Access*, vol. 6, no. 1, pp. 37371–37394, 2018.

[5] S. El-Sappagh, M. Elmogy and A. M. Riad, "A fuzzy-ontology-oriented case-based reasoning framework for semantic diabetes diagnosis," *Artificial Intelligence in Medicine*, vol. 65, no. 3, pp. 179–208, 2015.

[6] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi *et al.,* "Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, 2002.

[7] N. Cheniki, A. Belkhir and Y. Atif, "Mobile services discovery framework using DBpedia and non-monotonic rules," *Computers & Electrical Engineering*, vol. 52, no. 1, pp. 49–64, 2016.

[8] N. A. Saadon and R. Mohamad, "WSMO-lite based web service discovery algorithm for mobile environment," *International Journal of Advances in Soft Computing and its Applications*, vol. 5, no. 3, pp. 76–89, 2013.

[9] J. Sangers, F. Frasincar, F. Hogenboom and V. Chepegin, "Semantic web service discovery using natural language processing techniques," *Expert Systems with Applications*, vol. 40, no. 11, pp. 4660–4671, 2013.

[10] K. Elgazzar, H. S. Hassanein and P. Martin, "Daas: Cloud-based mobile web service discovery," *Pervasive and Mobile Computing*, vol. 13, no. 1, pp. 67–84, 2014.

[11] N. A. Saadon and R. Mohamad, "Cloud-based mobile web service discovery framework with semantic matchmaking approach," in *Conf. MySEC*, Langkawi, Malaysia, pp. 113–118, 2014.

[12] S. D. Plavila and R. Bajaj, "Cloud based mobile web service discovery using semantic matching and key-word based approach," *International Journal of Computer Applications*, vol. 144, no. 6, pp. 18–22, 2016.

[13] M. Fang, D. Wang, Z. Mi and M. S. Obaidat, "Web service discovery utilizing logical reasoning and semantic similarity," *International Journal of Communication Systems*, vol. 31, no. 10, pp. e3561, 2018.

[14] A. I. Ebada, S. Abdelrazek and I. Elhenawy, "Applying cloud based machine learning on biosensors streaming data for health status prediction," in *Conf. IISA50023*, Piraeus, Greece, pp. 1–8, 2020.

[15] A. Biørn-Hansen, T. M. Grønli and G. Ghinea, "A survey and taxonomy of core concepts and research challenges in cross-platform mobile development," *ACM Computing Surveys*, vol. 51, no. 5, pp. 1–34, 2018.
[16] C. Griffith, *Mobile App Development with Ionic, Revised Edition: Cross-platform Apps with Ionic, Angular, and Cordova*. Highway North, Sebastopol, Calefornia, USA: O'Reilly Media, 2017. [Online]. Available: https://www.amazon.com/Mobile-Development-Ionic-Revised-Cross-Platform/dp/1491998121.
[17] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. Highway North, Sebastopol, Calefornia, USA: O'Reilly Media, Inc, 2011. [Online]. Available: https://www.amazon.com/REST-API-Design-Rulebook-Consistent-ebook/dp/B005XE5A7Q.
[18] A. R. Kulkarni and S. D. Mundhe, "An application of porters stemming algorithm for text mining in healthcare," *International Journal of Management, IT and Engineering*, vol. 7, no. 11, pp. 223–228, 2019.
[19] Nuget, "Syn.WordNet 1.0.5," 2018. [Online]. Available: https://www.nuget.org/packages/Syn.WordNet.
[20] J. Kopecký, K. Gomadam and T. Vitvar, "Hrests: An html microformat for describing restful web services," in *Conf. IEEE/WIC/ACM*, Sydney, NSW, Australia, vol. 1, pp. 619–625, 2008.
[21] Hap, "Html Agility Pack," 2017. [Online]. Available: https://html-agility-pack.net.
[22] G. Sambasivam, J. Amudhavel, T. Vengattaraman and P. Dhavachelvan, "An QoS based multifaceted matchmaking framework for web services discovery," *Future Computing and Informatics Journal*, vol. 3, no. 2, pp. 371–383, 2018.
[23] H. Elhoseny, M. Elhoseny, S. Abdelrazek, A. M. Riad and A. E. Hassanien, "Ubiquitous smart learning system for smart cities," in *Conf. ICICIS*, Cairo, Egypt, pp. 329–334, 2017.
[24] S. M. A. El-Razek, H. M. El-Bakry, W. F. A. El-Wahed and N. Mastorakis, "Collaborative virtual environment model for medical e-learning," in *Proc. ACACOS '10*, Hangzhou, China, pp. 191–195, 2010.
[25] N. El-Rashidy, S. El-Sappagh, T. Abuhmed, S. Abdelrazek and H. M. El-Bakry, "Intensive care unit mortality prediction: An improved patient-specific stacking ensemble model," *IEEE Access*, vol. 8, no. 1, pp. 133541–133564, 2020.
[26] D. Hussein, S. N. Han, G. M. Lee, N. Crespi and E. Bertin, "Towards a dynamic discovery of smart services in the social internet of things," *Computers & Electrical Engineering*, vol. 58, no. 1, pp. 429–443, 2017.
[27] H. Zha, X. He, C. Ding, H. Simon and M. Gu, "Bipartite graph partitioning and data clustering," in *Proc. CIKM'01*, Atlanta, Georgia, USA, pp. 25–32, 2001.
[28] W. Sun, B. Wang, N. Cao, M. Li, M. Li *et al.,* "Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking," in *Proc. ASIA CCS'13*, Hangzhou, China, pp. 71–82, 2013.
[29] A. Elgazar, M. Aazam and K. Harras, "Edgestore: Leveraging edge devices for mobile storage offloading," in *Conf. CloudCom*, Nicosia, Cyprus, pp. 56–61, 2018.
[30] A. Ismail, S. Abdlerazek and I. M. El-Henawy, "Big data analytics in heart diseases prediction," *Journal of Theoretical and Applied Information Technology*, vol. 98, no. 11, pp. 1970–1980, 2020.
[31] K. Maeda, "Performance evaluation of object serialization libraries in xml, json and binary formats," in *Conf. DICTAP*, Bangkok, Thailand, pp. 177–182, 2012.
[32] D. D. Jemima and G. R. Karpagam, "Conceptual framework for semantic web service composition," in *Conf. ICRTIT*, Chennai, India, pp. 1–6, 2016.
[33] S. Bosnic, I. Papp and S. Novak, "The development of hybrid mobile applications with apache cordova," in *Conf. TELFOR*, Belgrade, Serbia, pp. 1–4, 2016.
[34] A. Freeman, *Essential Angular for Asp. Net Core MVC*. London, UK: Springer, 2017. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4842-2916-3_1.
[35] S. Schulte, U. Lampe, J. Eckert and R. Steinmetz, ""LOG4SWS,KOM: Self-adapting semantic web service discovery for SAWSDL," in *Conf. 6th World Congress on Services*, Miami, FL, USA, pp. 511–518, 2010.
[36] H. Elhoseny, M. Elhoseny, S. Abdelrazek and A. M. Riad, "Evaluating learners' progress in smart learning environment," in *Conf. AISI*, Cairo, Egypt, pp. 734–744, 2017.