A Self-Contained Configurable and Explainable Rule-Based Recommendation System Suitable For Deployment on Low-Resource Shared Web Hosting

1st Mirco Soderi
Data Science Institute
University of Galway
Galway, Ireland
mirco.soderi@universityofgalway.ie

2nd John Gerard Breslin
Data Science Institute
University of Galway
Galway, Ireland
john.breslin@universityofgalway.ie

2025 35th Irish Signals and Systems Conference (ISSC) | 979-8-3315-7593-9/25/$31.00 ©2025 IEEE | DOI: 10.1109/ISSC67739.2025.11291390

***Abstract*—Artificial intelligence (AI) is often associated with articulated services offered by cloud service providers that require specialised expertise to be used effectively and efficiently and involve charges per subscription, per resource usage, or both. However, small businesses might not be in a position to make significant investments in infrastructure, specialist staff, and cloud services. In this work, we propose a configurable rule-based expert system that delivers a sorted list of motivated recommendations based on data submitted through a web survey service and, most importantly, entirely implemented in PHP and deployable to a basic shared web hosting service, which the vast majority of companies, even the smallest ones, such as single-person brokerage companies, are already using to have a presence on the Internet. This has the potential to impact social sustainability, making AI technologies accessible to a broader variety of business players and to their customers, and to impact environmental sustainability by making the identification of the right provider for the right client less expensive from all the different points of view. Software artefacts are available on GitHub that can be used for demo purposes and as a starting point for the development of customised systems.**

***Index Terms*—Artificial intelligence, expert system, recommendation system, web survey, PHP, shared hosting, social sustainability, environmental sustainability, broker**

I. Introduction

The system described in this work is suitable for use in brokered markets. Typically, businesses that operate in brokered markets, referred to as agents or intermediaries, have multiple clients that they intend to offer possible goods or services to, and recommendations on these can either be provided by humans or by recommender systems of the type described in this manuscript. There are many examples of brokered markets; these include insurance, finance, real estate, fitness, private lessons, and freelance professionals, and there are examples of brokering that extend even beyond the perimeter of business, such as social or political brokerage [1].

Requirements can be obtained from clients in a variety of ways, including in-person meetings, calls, or Web surveys. Web surveys [2] are a convenient method of obtaining information from Web users, asynchronously. Users can be the general public as well as potential customers or partners that submit information in the context of a signed agreement. For example, a broker might use web surveys to collect information from companies that provide a service and potential customers of those companies that rely on the broker to find the best match. Most importantly, there are many different actors in these markets, from global giants to single-person companies [3]. Although big players can afford specialised staff, infrastructure investments, and cloud services, smaller players might benefit from solutions that support the identification of possible matches without involving relevant investments in staff, infrastructure, or cloud services. For this reason, it is worth investigating the possibility of delivering AI functionalities, such as rule-based expert systems [4], as part of low-resource cost-effective applications, such as a PHP-based web application hosted on a free shared hosting service. The software should be (i) modular [5], to make it possible to seamlessly integrate new functionalities or configuration options over time as needed without excessive effort; (ii) configurable, so that the reasoning of the system can be easily evolved over time to take into account new aspects or adjust their weight [6], and (iii) self-contained, which means that it should not rely on external services for data analytics or make use of software components that are not available in low-end web hosting services, unlike other web-based decision support systems described in the literature [7].

The proposed approach can positively impact *social sustainability* by making it possible for everyone to get quality recommendations at low cost thanks to the democratisation of traditionally sophisticated, access restricted, and expensive AI technologies [8]. At the same time, environmental sustainability can be positively impacted from many points of view. First, the proposed approach characterises itself for its

This publication has emanated from research supported in part by the European Digital Innovation Hub Data2Sustain, co-funded by Ireland's National Recovery and Resilience Plan (the EU's Recovery and Resilience Facility), the Digital Europe Programme, and the Government of Ireland, and by a grant from Taighde Éireann - Research Ireland under Grant Number 12/RC/2289_P2 (Insight).

efficiency in terms of energy usage compared to traditional AI solutions deployed on high-performance infrastructures. In addition, since it is Web-based, it reduces the need to travel [9], and since potentially hundreds or thousands of options are evaluated in a single click, it eliminates the need to address all the operators singularly, which leads to significant savings if multiplied by the number of clients [10].

II. Overview

The system is suitable for scenarios in which there are several options, physical goods or services, that are made available to potential clients, and a recommendation must be provided to the latter about which option best suits their needs. In fact, the recommendation system always returns all the options, ranked from the most suitable to the least suitable for the specific potential client under consideration. Associated with each option is a human-friendly description made up of a few statements that describe the elements that have been taken into consideration to evaluate that specific option against that specific potential client. The statements are ordered from the one that had the highest impact on the evaluation of the option to the one that had the least impact.

The system consists of three key components: (i) rules, which are made of conditions and consequences; (ii) recommendation engine; (iii) data access layer. An overview of the system is provided in Fig. 1; examples are reported in italics.

A. Rules

Each rule includes conditions, consequences, and a human-friendly motivation expressed in natural language. For each potential client and for each option, all conditions are tested, and they must be all verified for the rule to apply. If all conditions are verified, the consequences are applied. If consequences are applied, points are added or subtracted to the specific option, and the motivation of the rule is added to the advantages or disadvantages of the option, respectively. The number of points can be fixed, or it can result from a computation based on one or more responses that the potential client has given about their needs, and based on one or more features of the specific option under consideration. The motivation of the rule is meant to be shown to the potential client, so the phrasing of the motivation will need to take that into account. For example, if the semantics of the rule is *if the number of bathrooms that the client needs is the same as the number of the bathrooms in the house then the house gains 100 points*, a good motivation to be included in the rule would be *The house has exactly the number of bathrooms that you need*.

B. Ranking

When all rules have been applied to all options for a specific potential client, the options are ranked according to the points they have. The more points obtained, the higher the position of the option in the list of recommendations provided to the specific potential client. In addition, for each option, the statements that describe the advantages and disadvantages are ranked according to the points that have been added or subtracted from the option by the rule that has added the statement to the advantages or disadvantages of the option, with the most impactful listed first.

C. Recommendation engine

The recommendation engine accepts in input the preferences expressed by the potential client, it retrieves the rules, it retrieves the full list of the options with details, then for each option it goes through all the rules, determines for each of them if it applies on the basis of the conditions, and if it does, it gives or detracts points from the option, and appends a statement to the list of the advantages or disadvantages, according to the consequence specified in the rule. For each option, if multiple rules apply, all of them are enforced. In the end, the recommendation engine ranks the options, and inside of each option, it ranks the advantages and disadvantages, and it returns the recommendation in the form of a JSON that the Web application will then present to the potential client in a suitable formatting. This work does not cover the presentation aspect; it stops at provisioning the recommendation in the form of a JSON document.

D. Data access layer

The data access layer contains functions that implement the retrieval of the rules, the retrieval of the responses given by the potential client about their needs and preferences, and the retrieval of the options, each with all its related details, along with a few configuration parameters that are required for retrieving the mentioned information. The preferences of the potential client and the details of each option must be representable in the form of questions and answers, and each question must be representable through *two* texts: (i) the formulation of the question that is presented to the user who fills out the survey; (ii) a reformulation (*chunk*) that is kept hidden and that is suitable for referring to that specific piece of data within decision rules in such a way that the overall formulation of conditions and consequences results as natural as possible. It is important to stress that *chunks* are structural information; they are part of the questions, not of the answers. For production use, security measures must be taken or external services must be relied on.

III. Rules enforcement

At the core of the system is the enforcement of the rules that govern the ranking of the options. In this section, the enforcement of the rules is covered in detail.

A. Assessment of conditions

The recommendation engine iterates over all the options, for each option iterates over all the rules, and for each rule iterates over all the conditions to determine if all of them are satisfied, in which case the rule consequences are applied.

Examples of conditions follow, in the case for example of an estate agent that is using the system to provide recommendations to people interested in buying a house: (i) *the region where the client is looking for a house is not the same as the*

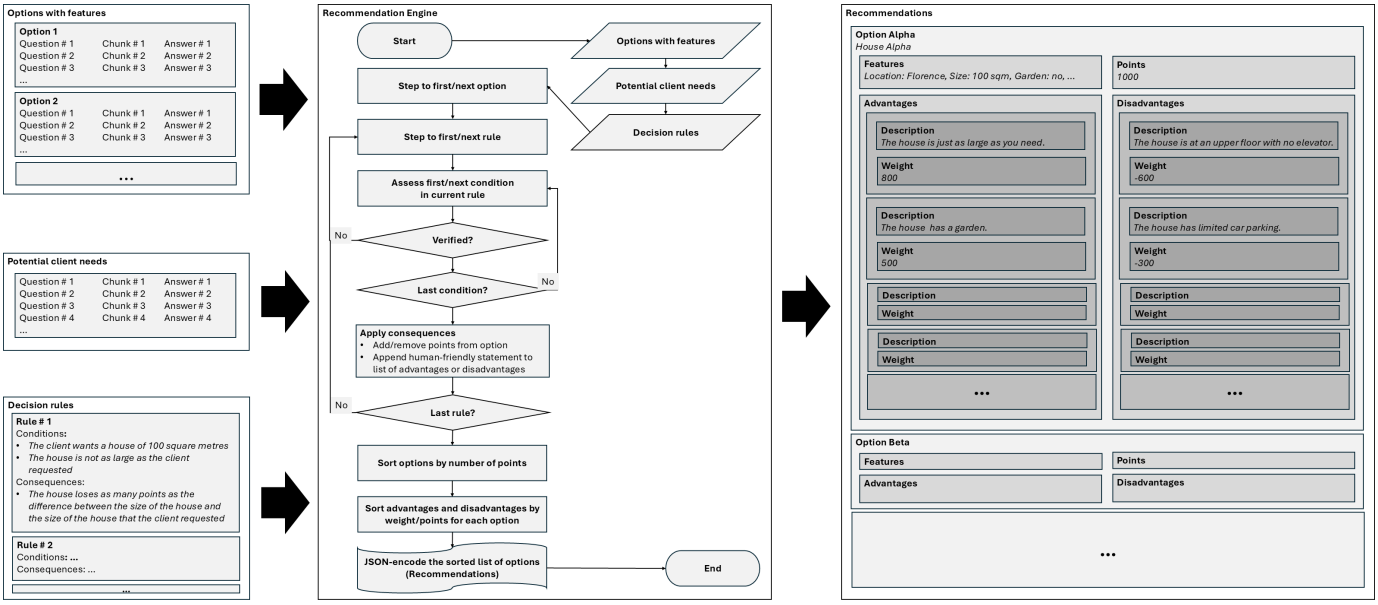

Fig. 1. System overview

region where the house locates; (ii) *it is true that the client wants a bus stop near the house*; (iii) *the price that the client is ready to pay is higher than the price of the house*; (iv) *the size of the house that the client needs is lower than the size of the house*; (v) *the client's preference in terms of the furniture in the house is *yes, there must be furniture**; (vi) *the number of bathrooms that the client needs is higher than the number of the bathrooms in the house*.

To determine whether a condition is verified, the recommendation engine accesses the software module where conditions are implemented using reflection. Each condition is implemented by a function that returns *true* if the condition is verified. To identify the function that is suitable for evaluating the given condition, regular expressions are used. The recommendation engine iterates over the functions that are implemented in the conditions module, and for each of them, it retrieves the comment that describes the function, and it builds a regular expression applying predefined transformations to such comment. Then, it uses the regular expression to determine if there is a match with the condition that has to be evaluated. If there is a match, that means that the function is the appropriate one to evaluate the condition.

Examples of comments are: (i) *... is not (the same as) ...*; (ii) *it is true that ...*; (iii) *... is higher than ...*; (iv) *... is lower than ...*; (v) *... is (the same as) ...*; (vi) *... is higher than ....* Comparing the example comments with the example conditions provided above, it is intuitive to understand how a regular expression can be written starting from the comment, to determine if the condition is one that can be evaluated using the function that is associated with the comment. Basically, the regular expression is built in a way such that any sequence of characters of any length is allowed in place of every occurrence of three dots, and in place of any part of the comment that is enclosed within brackets. For example, the regular expression constructed from *... is not (the same as) ...* is `/^()(.*)(is not )(.*)()(.*)()$/`, whereas the regular expression that is constructed from *it is true that ...* is `/^(it is true that )(.*)()$/`. The functions are tested in order, and the search stops as soon as a match is found. Due to this, ambiguities can be avoided by wisely sequencing functions in the condition module. For example, the function described by the comment *... is (the same as) ...* is implemented *after* the function described by the comment *... is not (the same as) ....*

Once the function has been identified, it is necessary to extract the parameters that must be passed to the function so that it can determine if the condition is verified or not in the specific case. The parameters are segments of the condition that fill the gaps (three dots) that are present in the comment. If the comment includes more than one gap, the corresponding number of parameters is extracted. In the six examples of condition/comment presented above, the following parameters would be extracted: (i) (1) *the region where the client is looking for a house*, (2) *the region where the house locates*; (ii) (1) *the client wants a bus stop near the house*; (iii) (1) *the price that the client is ready to pay*, (2) *the price of the house*; (iv) (1) *the size of the house that the client needs*, (2) *the size of the house*; (v) (1) *the client's preference in terms of the furniture in the house*, (2) **yes, there must be furniture**; (vi) (1) *the number of bathrooms that the client needs*, (2) *the number of the bathrooms in the house*.

In more technical terms, to extract the parameters for example from the condition *the region where the client is looking for a house is not the same as the region where the house locates*, once that it has been determined that the function that is suitable for evaluating the condition is the

one that is described by the comment *... is not (the same as) ...*, what happens is that the comment is split using the three dots as a separator, obtaining in this example the following segments: (i) *empty string*; (ii) *is not (the same as)*; (iii) *empty string*. Then, the list of the segments is traversed taking the segments in groups of contiguous two, and each pair of segments is used for constructing a regular expression that is suitable for extracting what is present in the condition in place of the three dots that separate the two segments. If the comment contains parts in brackets, the two cases are treated separately; a first attempt of extraction is made including the part in brackets (but removing the brackets themselves), and if it fails, a second attempt of extraction is made excluding the brackets and whatever it is that they contain.

The extracted parameters can be literals or references to answers. This is where the *chunk* comes into play. For each extracted parameter, the recommendation engine goes through the questions that are asked to the potential client, and if the parameter that was extracted from the condition exactly corresponds to the *chunk* that is associated with one of those questions, then the parameter is resolved to the answer that the potential client has given to that question. If no match is found, the same is performed in the options form. If no match is found, the parameter is interpreted as a literal value.

B. Application of consequences

If all conditions are verified for a given potential client, option, and rule, then consequences of that rule are applied to the option for that potential client.

Similarly to conditions, consequences are expressed in natural language. Examples of consequences follow: (i) *the house loses 1000 points*; (ii) *the house loses as many points as 10 times the difference between the age of the house and the age of the house that the client wants*.

Similarly to conditions, there is a software module where all the functions that implement the application of consequences are wrapped. Given a consequence expressed in natural language, the identification of the appropriate function for applying the consequence, and the extraction of the parameters happen in the same way as has been described for conditions.

The function that applies the consequences accepts three input parameters: (i) *parameters*, which is the list of parameters that have been extracted from the statement that describes the consequence in the rule; (ii) *rule*, which is an object that represents the rule to be applied; (iii) *status*, described in the following of this section. Speaking of the extraction of the parameters, for example, if the consequence that is present in the rule is *the house loses 1000 points* and the function that implements the consequence is described by the comment *the function loses ... points*, then the list of the parameters has length 1 and contains the only element *1000*, while if the consequence present in the rule is *the house loses as many points as 10 times the difference between the age of the house and the age of the house that the client wants* and the function that implements the consequence is described by the comment *the house loses as many points as ... times the difference between ... and ...* then the list of the extracted parameters has length 3, and it contains (i) the literal value *10*, (ii) the value that results from the resolution of the parameter *the age of the house*, and (iii) the value that results from the resolution of the parameter *the age of the house that the client wants*.

The *status* object is part of the representation of each option, together with the properties that describe the peculiar features of the option. It is passed by reference to the function that implements the consequence and contains three properties: (i) *points*, (ii) *advantages*, and (iii) *disadvantages*. The application of the consequence consists of: (i) adding or subtracting a certain number of points by modifying the value of the property *points* in the *status* object; (ii) adding a new object to the list of advantages or disadvantages, which contains two properties, namely *weight*, and *description*. The weight is set to the number of points that the consequence function has given or subtracted from the option, in absolute value. The description is retrieved from the rule that the consequence function has applied, and it consists of a client-friendly motivation for the addition or subtraction of points from the specific option.

IV. Proof of concept

For a proof of concept, we have considered the case of an estate agent, so in our case the potential client is a person that is looking for a house and the options are all the different houses that are on sale at a given point in time.

A. Software artifacts

All artefacts related to this proof of concept are available on GitHub[1]. It is a Docker Compose application, based on PHP and the Apache web server. The ZIP archive contains the Docker Compose configuration file, the Dockerfile, and a TAR archive with the artefacts of the application that are copied to the Web root of the Apache web server through commands contained in the Dockerfile. The TAR archive contains the following files and folders: (i) *rules.json*, which contains the rules that are used for the ranking for the purposes of the proof-of-concept; (ii) *conditions.php*, which contains the functions that implement the conditions that are used in the proof-of-concept; (iii) *consequences.php*, which contains the functions that implement the consequences that are used in the proof-of-concept; (iv) *recommender.php*, which contains the implementation of the recommendation engine; (v) *config.php*, which contains a few configuration parameters and functions that implement the data access layer; (vi) the *forms* folder, which contains subfolders and PHP pages that simulate the TypeForm APIs, as it will be described in detail in the following of this section; (vii) the *analysis* folder, which includes a Postman collection and a spreadsheet with performance measurements, as it is described in detail in Section V.

B. Data source

The functionalities and configuration parameters related to the access to data are wrapped in the config.php page. We

[1] https://github.com/mircosoderi/estateagent

considered the case where all data are stored in a TypeForm account. TypeForm is an online surveying system. It exposes APIs for schema and data retrieval, but the free account comes with too stringent limitations to carry out scalability tests, and opting for a paid account would have forced the reader to do the same, so we finally decided to simulate the strictly needed subset of the TypeForm APIs in the *forms* folder.

The relevant TypeForm APIs are: (i) GET /form/*form_id*, with no arguments, to retrieve the structure of a form/survey; (ii) GET /form/*client_form_id*/responses, with the *included_response_ids* parameter that identifies the survey response of interest, and the *page_size* parameter that sets the maximum number of survey responses that should be included in the API response.

With that in mind, we structured the *forms* folder in a way such to expose the following APIs: (i) GET /form/exad2bN7, which returns the structure of the survey submitted to clients; (ii) GET /form/exad2bN7/responses, which returns a fictional response given by a client; (iii) GET /form/xNPPsDO0, which returns the structure of the survey used to input the options/houses; (iv) GET /form/xNPPsDO0/responses, which returns *page_size* fictional houses.

A detailed discussion of the TypeForm representation of forms and responses is beyond the purposes of this work. However, it is important to mention that in the JSON document that describes the structure of the form, each question includes two properties: (i) *title*, which contains the question displayed to the user; (ii) *ref*, where we put the *chunk*, as defined above.

The config.php also contains a function that accesses the rules.json document to retrieve the full set of fictional rules. For scalability evaluation, the parameter *pretend_you_have_these_many_rules* can be included in the GET request made to the recommendation engine, and it is passed down to the function that retrieves the rules, which returns the indicated number of rules, duplicating them if necessary, then randomly picking the required number of them.

C. Operation

The proof-of-concept web application is run by cloning the GitHub repository, navigating to the root of the local repository, and running the command *docker compose up*. The application listens on port 80.

The recommendation engine in particular returns a JSON object with two properties: (i) *recommendations*, a JSON array of options/houses, with each option containing the properties that describe the house, along with the *status* object as described in Section III-B; (ii) *exe_time*, with separate indication of the computation and data access time.

The *analysis* folder in particular contains the export of the Postman collection that we have used to verify the correctness of the system and for performance analysis.

The collection contains a few variables whose names start with *results*, which were generated during our local performance tests; the reader should delete them before performing their own tests. The other variables contain the following configuration parameters: (i) *website_baseurl*, the hostname and path to the root of the proof-of-concept web application; (ii) *typeform_baseurl*, to be set to *localhost* to use the simulated TypeForm APIs; (iii) *client_typeform_response_ids*, the identifier of the TypeForm response that contains the potential client preferences; (iv) *benchmark_counter*, used by the performance test script that we have implemented in the collection; (v) *benchmark_options*, the number of options/houses that we want to pretend to be available, which can be set manually, but is also used by the performance test script; (vi) *benchmark_rules*, the number of rules that we want to pretend to be configured, which can be set manually, but is also used by the performance test script.

The collection contains one request only, the *Recommender* request, which includes the performance test script as a post-request script. The request can be run manually to inspect the response body or automatically through the collection runner to assess the system scalability. In the latter case, the recommended initial values of the collection variables are: (i) *benchmark_counter* set to 0; (ii) *benchmark_options* set to 2; (iii) *benchmark_rules* set to 10. If the recommended configuration is used, the performance test script makes it happen the following: (i) 10 requests are sent for each combination of number of options and rules, and the average response time is stored in a new collection variable; (ii) the number of options ranges from 2 to 1024 and doubles at each step; (iii) the number of rules ranges from 10 to 100 at steps of 10.

V. Results

Both correctness and performance have been evaluated. The correctness has been verified by manual inspection of the responses provided by the recommendation engine for a variety of client preferences, options, and rules. Performance has been verified (i) by running the Postman collection with the configuration indicated in Section IV-C to evaluate local performance; (ii) by issuing browser requests to evaluate the performance of the application deployed on free web hosting *infinityfree.com*; Postman could not be used because the provider only allows requests from web browsers for free plans.

In terms of performance, the response time includes: (i) data access; (ii) computation of the recommendation; (iii) network/client overhead. Data access and computation times are measured by the recommendation engine for each request and returned alongside actual recommendations. The computation times measured in the real free web hosting service for a growing number of options and rules are represented in Fig. 2 and in Fig. 3, respectively.

It is worth noticing that: (i) the computation time grows approximately as fast as the number of options does for a fixed number of rules, whatever the number of rules is; (ii) the computation time roughly grows as much as the number of rules does for a fixed number of options, whatever the number of options is; (iii) the computation time observed for a system with 100 rules and 1024 options is 1699 ms.

The same analysis has been performed for the Docker Compose application in our local system (Intel Core i5-

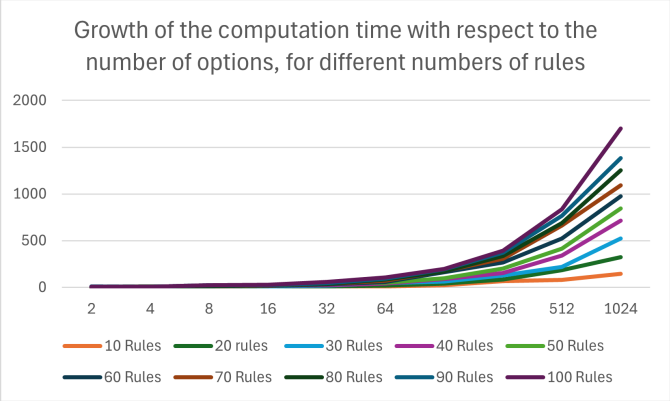

Fig. 2. Computation time (ms) for growing number of options on real hosting

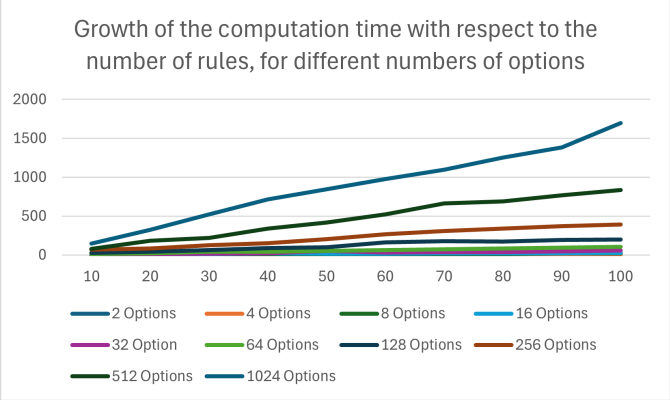

Fig. 3. Computation time (ms) for growing number of rules on real hosting

1035G1, 16 GB RAM, Windows 10), and it has been observed that: (i) the computation time is significantly higher in the local system than in the real web hosting; (ii) the scalability is the same observed in the real web hosting.

All measured values are reported in a spreadsheet in the *analysis* folder, separately for the local and the remote deployment. For each measurement, the following data are provided: (i) number of rules; (ii) number of options; (iii) computation time; (iv) data access time; (v) total response time; (vi) scalability for growing number of options; (vii) scalability for growing number of rules. Each sheet contains 100 measurements, each of which is averaged over 10 requests. The scalability with respect to the number of options is computed for each measurement n as $(t_n/t_n-1)/(o_n/o_n-1)$, with t_n being the computation time in measurement n, t_n-1 the computation time in measurement $n-1$, o_n the number of options in measurement n, o_n-1 the number of options in measurement n-1. A similar approach is adopted for the scalability with respect to the number of rules.

Data access times measured during our performance tests are not relevant because the simulated TypeForm APIs were accessed. As a side note, in real applications backed by TypeForm, responses cannot be interpreted without first retrieving the structure of the form, so four requests are needed: (i) client form; (ii) client answers; (iii) option form; (iv) options with details. Form structures change sporadically, so we can expect cache hits in most cases. We have observed that the TypeForm APIs to retrieve forms and responses execute in about 400 ms on cache hit, and in about 800 ms otherwise, for a total of about 2.5 seconds for small numbers of options; the scalability for growing numbers of options could not be measured. Data access is performed only once per recommendation request, before the computation of the recommendation starts.

All of the above holds in the absence of concurrency. If multiple recommendations are requested exactly at the same time, the response time is affected by the limitations that the web hosting provider imposes on the computational resources that can be used by a single website. The *Reload all tabs* of Google Chrome has been used to simulate multiple users requesting a recommendation to the remotely hosted application exactly at the same time, and it has been observed that if more than 8 users send a request exactly at the same time, approximately half of them get penalised in terms of response time, with approximately 25% of them being heavily penalised. However, this largely depends on the hosting provider and plan.

VI. Conclusions

In this work, a recommendation system has been presented that is (i) self-contained, in the sense that it does not rely on any external provider of data analytics functionalities; (ii) Web-based; (iii) suitable for integration in web applications that run on basic web hosting plans, even shared free web hosting plans; (iv) configurable, with rules written in natural language; (v) explainable, as it provides human-friendly descriptions of the advantages and disadvantages of each option; (vi) even though, sufficiently scalable and performant.

References

[1] K. Stovel and L. Shaw, "Brokerage," *Annual review of sociology*, vol. 38, no. 1, pp. 139–158, 2012.
[2] M. Callegaro, K. L. Manfreda, and V. Vehovar, *Web survey methodology*. Sage, 2015.
[3] Y. Kirkels and G. Duysters, "Brokerage in sme networks," *Research Policy*, vol. 39, no. 3, pp. 375–385, 2010.
[4] A. Abraham, "Rule-based expert systems," *Handbook of measuring system design*, 2005.
[5] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 99–108, 2001.
[6] J. E. Vargas and S. Raj, "Developing maintainable expert systems using case-based reasoning," *Expert Systems*, vol. 10, no. 4, pp. 219–225, 1993.
[7] Y. Zeng, Y. Cai, P. Jia, and H. Jee, "Development of a web-based decision support system for supporting integrated water resources management in daegu city, south korea," *Expert Systems with Applications*, vol. 39, no. 11, pp. 10 091–10 102, 2012.
[8] P. K. Yu, "The algorithmic divide and equality in the age of artificial intelligence," *Fla. L. Rev.*, vol. 72, p. 331, 2020.
[9] P. Zhu and S. G. Mason, "The impact of telecommuting on personal vehicle usage and environmental sustainability," *International Journal of Environmental Science and Technology*, vol. 11, pp. 2185–2200, 2014.
[10] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.