The JavaScript Package Selection Task: A Comparative Experiment Using an LLM-based Approach

J. Andrés Diaz-Pace
ISISTAN Research Institute, CONICET & UNICEN University,
Tandil, Buenos Aires, Argentina
andres.diazpace@isistan.unicen.edu.ar

Antonela Tommasel
ISISTAN Research Institute, CONICET & UNICEN University,
Tandil, Buenos Aires, Argentina
antonela.tommasel@isistan.unicen.edu.ar

Hernan C. Vazquez
Faculty of Sciences, UNICEN University
Tandil, Buenos Aires, Argentina
hvazquez@exa.unicen.edu.ar

Abstract

When developing *JavaScript (JS)* applications, the assessment and selection of JS packages becomes challenging for developers due to the growing number of technology options available. Given a technology-related task, a common developers' strategy is to query Web repositories (e.g., from GitHub) via a search engine (e.g., NPM, Google) and then shortlist candidate JS packages. However, this search might return a long list of results and not all of them might be relevant. Thus, these results often need to be (re-)ordered according to the developer's criteria. To address these problems, in prior work, we developed a recommender system called `AIDT` that assists developers in the package selection task. `AIDT` relies on meta-search and machine learning techniques to infer the relevant packages for a query. An initial evaluation of `AIDT` showed good search effectiveness, but the tool was unable to explain its choices to the developer. Research on Large Language Models (LLMs) has recently opened new opportunities for this kind of recommender systems. Anyway, human developers should judge whether the recommendations (e.g., JS packages) of these tools (either `AIDT` or LLMs) are fit to purpose. In this paper, we propose a Retrieval Augmented Generation (RAG) architecture for using LLMs in the domain of technology selection that enhances the `AIDT` original design. Furthermore, we report on a user study using both `AIDT` and different LLM-based variants (`ChatGPT`, `Cohere`, `Llama2`) on a sample of JS-related queries, in which we compared their results and also validated them against developers' criteria for the task. Our findings show that, although the ranking capabilities of LLMs are not yet on par with `AIDT` or human efforts, the RAG architecture can achieve a decent performance and is good at providing explanations for the package choices in the rankings. The latter feature makes it more transparent than `AIDT` and, thus, potentially more flexible to support developers' tasks.

Keywords: Package Selection, JavaScript, LLMs, RAG Architecture, User Study.

1 Introduction

In software development in general, and JavaScript (JS) applications in particular, the use of libraries and frameworks can greatly improve developers' productivity by accelerating development cycles and delivering

value to customers. Nonetheless, choosing (and reusing) a JS package that fulfills the needs of a development task can be a complex decision-making activity for developers. In addition, an inappropriate selection can negatively affect the application design, the product quality, and the organizational goals [1]. This complexity stems from the large number of technology options available in Web repositories, such as GitHub[1] or NPM[2] (Node Package Manager) [2]. Thus, JS developers have to search, evaluate, and compare several packages suitable for their tasks, and keeping up-to-date with technology becomes challenging. This activity can be perceived as a "technological fatigue"[3] by developers.

Although some JS search engines (e.g., NPM) have been enhanced over the last years, their poor effectiveness still contributes to technological fatigue. With the hope of having better results, developers also resort to general-purpose search engines (e.g., Google or Bing). However, the downside of such engines is that they tend to return long lists of documents, and developers have to navigate within each result to find candidate JS packages, leading to information overloading issues. Once a developer identifies a set of candidate packages, she must analyze each one to decide the best fit for her need or task. Typically, this decision is driven by package features, such as popularity in the community, contributors, or number of downloads, among others. Weighting these features for comparison purposes is not straightforward.

In prior work [3], we proposed a recommender system called `AIDT`[4] to assist developers in searching and ranking JS packages. We refer to this `AIDT` implementation as the *vanilla* version. Given a developer's query expressing a technological need, the problem is how to return a ranking of relevant packages that satisfy the query – we refer to it as the *JS package selection task*. To tackle this problem, `AIDT` works in two phases: (i) it applies a meta-search strategy [4] that combines results from multiple engines, (ii) based on the recovered packages, it ranks them by relevancy by means of a Machine Learning (ML) model, which relies on a learning-to-rank method [5]. The ML model can infer a package ranking by analyzing features extracted from JS projects available on GitHub repositories. We performed an initial evaluation of the `AIDT` effectiveness using a predefined set of queries and a database with 1000 GitHub projects. In these experiments, we obtained an average precision improvement of 20% when compared to NPM, and `AIDT` recommended a larger number of relevant packages than NPM. Furthermore, `AIDT` showed the feasibility of using a data-driven strategy that "learns" selection criteria from features from the (open-source) JS community.

The emergence of assistive technologies based on Large Language Models (LLMs), such as OpenAI's ChatGPT [6], GitHub's Copilot[5], or Meta's Llama2 [7], has brought new opportunities and challenges for development-related tasks. Recent evidence shows that LLMs can be useful, although they also have limitations and pitfalls [8]. For example, ChatGPT [9] can work as a general-purpose search engine, which can additionally provide recommendations, rankings, and even justify them [10]. In the context of our previous experiences with *vanilla* `AIDT`, a natural question arises: *how do LLMs perform in the package selection task? Can they do better than humans or than `AIDT`?* Thus, in this paper, we focus on the LLMs capabilities to assist developers in selecting and ranking JS packages.

In initial experiments [11], we asked a group of JS developers to work with a sample of queries and compared their results against those produced by `AIDT` and `ChatGPT` (for the same queries) to assess the pros and cons of both tools. In this setting, we used `ChatGPT` in a zero-shot mode. Although the results were encouraging, we found out that `ChatGPT` was often imprecise and returned JS packages that were very different from the rankings suitable for the queries. To address this limitation, in this work, we propose a Retrieval Augmented Generation (RAG) [12] architecture for the JS package selection task that subsumes both *vanilla* `AIDT` and the LLM-based (zero-shot) approach. The RAG choice was driven by the need of grounding the LLM results on established knowledge sources to mitigate the issues of the zero-shot mode while improving on `AIDT`. We used a dataset of 4600 curated JS packages from GitHub as our main knowledge source. The differences in the RAG architecture include support for more powerful search and ranking mechanisms, and the incorporation of LLM-based explanations for the outputs.

For evaluating the RAG architecture, we repeated our experiments against the results produced by humans and *vanilla* `AIDT` comparing them in terms of retrieval and ranking metrics. Furthermore, we extended the initial evaluation to include other LLMs, such as `Llama2` and `Cohere`[6]. We also performed a qualitative analysis of the generative LLM capabilities, involving package recommendation criteria such as key characteristics, pros and cons of each package. Using the RAG architecture led to a precision increase in the recommendations by putting JS packages relevant to the user's query at the top of the rankings and reaching a performance comparable to that of *vanilla* `AIDT`. Furthermore, the LLM approach was very good at providing justifications for its recommendations, which is a feature that *vanilla* `AIDT` cannot offer due to

[1] https://www.github.com
[2] https://www.npmjs.com/
[3] https://medium.com/@ericclemmons/javascript-fatigue-48d4011b6fc4
[4] Spanish acronym for Intelligent Assistant for Technology Decisions.
[5] `https://github.com/features/copilot`
[6] `https://cohere.com/blog/command-r`

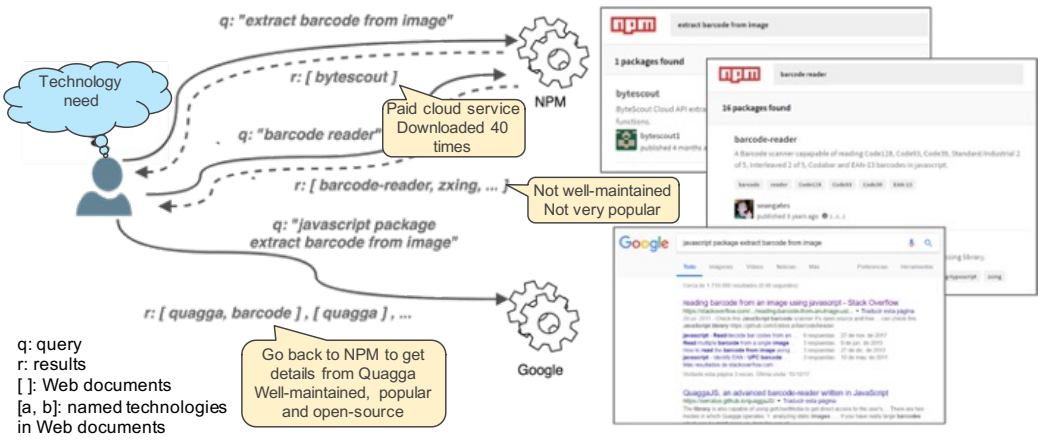


Figure 1: Example of JS package selection task using different search engines (AIDT).

its black-box characteristics.

The rest of the paper is organized into six sections as follows. Section 2 motivates the search and ranking of JS packages and briefly describes the workings of the original AIDT tool. Section 3 presents the LLM-based RAG architecture and its main capabilities. Section 4 outlines the study design, presenting the research questions and the experimental procedures. Section 5 reports the findings of our evaluation. Section 6 analyzes related works. Finally, Section 7 gives the conclusions and outlines future work.

2 The Package Selection Task

The selection of software technologies influences both the development process and the quality of the final product [13]. The successful application of a given technology, such as a JS package, means that its usage for a task produces a desired objective [14]. This also depends on contextual features, such as alignment between the developer's need and the chosen package, package maintenance support, or license type, among others. For developing AIDT, we departed from two ideas. First, the search and comparison of JS packages can take advantage of multiple information sources. Second, existing JS projects can provide useful information about criteria for assessing the relevance of a package. In the following, we present a scenario of how the technology selection works in practice.

2.1 Motivating example

Let us consider a JS developer who needs to extract a barcode from an image to automate the processing of barcodes from an image file, as illustrated in Figure 1. Initially, the developer goes to the NPM package repository and submits the query "*extract barcode from image*" to the search engine, which returns only the *bytescout*[7] package as output. *Bytescout* is a JS client for a cloud service. When reading about *bytescout*, the developer realizes that it is a paid service and that the JS client is not open-source. Also, when looking at the description, NPM reports that *bytescout* has been downloaded 40 times in the last month, which might indicate that it is not very popular in the JS community. Let us assume that these feature do not convince our developer, or that they are not aligned with the project needs. However, *bytescout* is the only technology returned by NPM. In this context, several options arise: (i) adopt the package despite disagreeing with its features, (ii) implement a solution for reading barcodes from scratch, (iii) submit a modified query to NPM to get more results, or (iv) rely on other information sources (e.g., Google) to find alternative technologies. Let us suppose that our developer picks the third option and re-phrases the query as "*barcode reader*", which makes NPM return 16 results this time. After inspecting each result, the developer is still unconvinced about using any of those technologies, since they do not seem very popular or have enough maintenance. The scenario exposes the limitations of JS-specific search engines, like NPM.

Let us assume that our developer goes for the fourth option instead and submits the query "*extract barcode from image javascript package*" to Google. This query returns a list of Web pages that the developer inspects to check whether some JS packages are mentioned. In doing so, our developer realizes that a package called *QuaggaJS*[8] is referenced in three results from the top-10 pages of the list. As the developer is not aware of this technology, she goes back to the NPM repository and finds that *QuaggaJS* is more popular

[7] https://bytescout.com/

[8] https://serratus.github.io/quaggaJS/

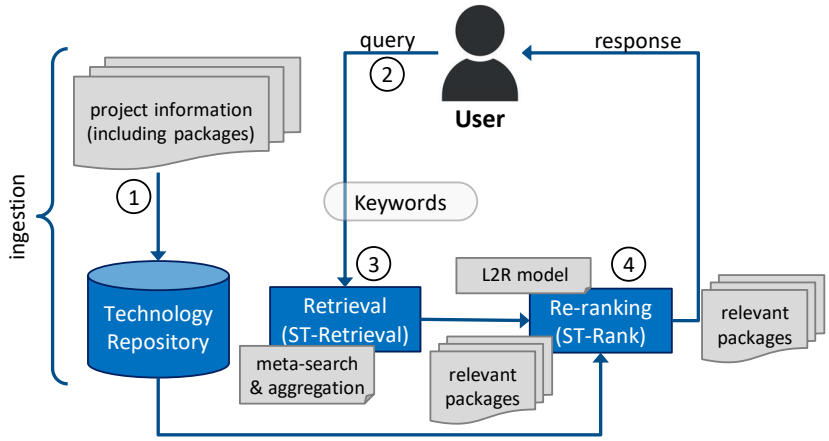


Figure 2: Overview of the recommendation workflow in *vanilla* `AIDT` (adapted from [3]).

Table 1: Borda Fuse aggregation example ([points] name) used by `AIDT` [3].

NPM	Google	Bing	Final list
[4] bytescout	[4] quagga	[4] quagga	[8] quagga
	[3] bcreader	[3] bc-js	[6] bytescout
	[2] bytescout	[2] *bwip-js*	[4] bcreader
	[1] *jaguar*	[1] bcreader	[3] bc-js
			[2] *bwip-js*
			[1] *jaguar*

than *bytescout*, it is open-source and well-maintained by the community. At this point, our developer can either pick *QuaggaJS* to fulfill the development need or keep looking for alternative packages. This scenario illustrates the challenge of using general-purpose search engines for retrieving JS packages, as well as the issues related to their comparison.

2.2 The `AIDT` tool

`AIDT` [3] is a recommender system for the JS domain that works in two stages, implemented by two separate modules: *ST-Retrieval* and *ST-Rank*, as depicted in Figure 2. We refer to this version as *vanilla* `AIDT`.

2.2.1 ST-Retrieval

This module takes a developer's query (2) and returns a list of candidate JS technologies[9] matching the query. The query is written in natural language and specifies a technological requirement (e.g., "extract barcode from image"). The package retrieval is treated with a meta-search strategy [4], in which the original query is sent in parallel to several search engines, each returning an ordered list of items for the query (3). In its initial implementation, `AIDT` relied on multiple engines to broaden the scope of a query, namely: NPM, Google, and Bing[10]. These engines provide a keyword-based search mechanism. Each engine returns a set of Web pages (or documents) that might have references to zero or more JS packages. In our example, NPM returned one result (*bytescout*) matching a package name in the repository.

An ordered list of packages per search engine is created based on the named packages extracted. Table 1 shows an example of the JS packages obtained from the NPM, Google and Bing engines for the query "*extract barcode from image*". The individual lists are then combined into a single one using a ranking aggregation function (3). We rely on the Borda Fuse method for merging the lists [15]. In Borda Fuse, each search engine is considered a voter with a list of n-ordered candidates (i.e., the JS packages). For each list, the best first candidate receives n points, the second candidate receives $n-1$ points, and so on. The points awarded by the different voters are added, and the candidates are ranked in descending order according to the total points obtained. The last column of Table 1 exemplifies a Borda Fuse aggregation for our example, in which the most relevant packages from the individual lists (*quagga* and *bytescout*) ended up at the top of the final list.

[9]For simplicity, the words "package" and "technology" are used interchangeably as synonyms in the paper.
[10]https://www.bing.com

2.2.2 ST-Rank

Although *ST-Retrieval* can search through a large collection of Web resources, the first results of the ranking are not always relevant to the query. Along this line, the *ST-Rank* module permits to refine the retrieved items to generate a better ranking of JS packages (4). The (new) ranking is constructed by looking at package features and decisions made by other JS projects. This information is crawled beforehand from the NPM and GitHub repositories and stored in a technology repository (1). The rationale for incorporating these features into a ranking is that if a package T was selected in a project (over other available options), there should be a criterion that renders T more relevant (than the other options) that is derivable from the features. *ST-Rank* tries to learn this selection criterion through a data-driven strategy.

A JS package P is represented by a number of predefined features and its dependencies on other packages. To assemble the dataset, we collected more than 40 features from NPM and GitHub, including project stars, number of downloads, dependent projects, developers contributing to the project, subscribers, commits, files, or presence of tests, among others. Furthermore, we assess the popularity of a technology T by means of a metric so-called *CDSel* (Community Degree of Selection) [3], which models the relationship between the projects in which T was selected and the relevance of those projects. For example, in our repository, we obtained a *CDSel* value of 396.192 for *quagga*, 15.646 for *bytescout*, and 1.791 for *bcreader*; which would mean that *quagga* is selected more often than *bytescout* and *bcreader* in the repositories.

The technology repository serves as the basis for building an ML model to rank JS packages. The training dataset contains a set of instances, each capturing a pair of technologies and their associated features. Initially, a training ranking is computed for each technology according to its *CDSel* value. For instance, in our example, *quagga* will be ranked first since its *CDSel* value is higher than those for *bytescout* and *bcreader*. Then, each technology is mapped to a feature vector $[FT_{i1}, FT_{i2}, ..., FT_{in}]$ where FT is an individual feature and n is the total number of features. At last, for each pair T_i and T_j, a pair vector (i.e., a training instance) is created as the concatenation of the feature vectors for T_i and T_j. If T_i is more relevant than T_j, then the label 1 is assigned to the pair, or 0 otherwise. Based on the training dataset, we apply a *learning-to-rank* (L2R) technique [5] that works on the instances as if it were a binary supervised classification. The classification model is implemented with GBRank [16], which is a popular gradient-boosting algorithm for L2R. Once built, the ML model can predict the order for any JS package pair, and the resulting pairs are consolidated and finally presented to the developer.

3 LLMs and Retrieval Augmented Generation

Over the last few years, LLM technologies [6] have improved the state of the art in several Natural Language Processing (NLP) tasks by pre-training on large-scale text corpora and fine-tuning to follow human instructions. In particular, LLMs have demonstrated strong zero-shot and few-shot generalization capabilities. The former refers to the ability to perform a task without having seen any (related) training examples; while the latter refers to being able to perform a task with a minimal number of examples [17]. LLMs can be seen as a paradigm shift in research that facilitates in-context learning by simply constructing natural language prompts or instructions [6, 17], which promote applications across various domains [10].

Related to our work, LLMs have enabled the development of new kinds of recommender systems based on user instructions [10]. In this context, tools like ChatGPT [9] provide new means for information seeking, for example, by allowing users to interact with its underlying LLM, ask questions and retrieve information. Recommendations can be formulated as prompt-based tasks, where information about users (e.g., developers) and items (e.g., JS packages) gets integrated into personalized prompts as model inputs [10]. For instance, Figure 3 shows a possible prompt and response for the query scenario of Figure 1 using a zero-shot strategy. Note that the LLM, unlike the *vanilla* `AIDT`, can give additional information in the response. In general, it has been argued that LLMs can provide more natural and explainable recommendations, helping to reduce the cold-start problem and providing cross-domain recommendations [10]. Nonetheless, given that LLMs have not been naturally trained to make recommendations or generate rankings, the quality of the results might suffer [18]. Recent works attempting to employ LLMs to deal with recommendation tasks have reported mixed results [18]. On the downside, LLMs have shown limitations for handling domain-specific or highly-specialized queries, which often manifests through the well-known problem of hallucinations [19].

3.1 RAG Architecture

Although LLMs excel in generating dialogues, they often face challenges in integrating factual information due to their limited mechanisms for incorporating knowledge [12,20]. This issue may hinder their performance in tasks that rely on substantial domain knowledge access [17]. Recent advancements in augmenting LLMs

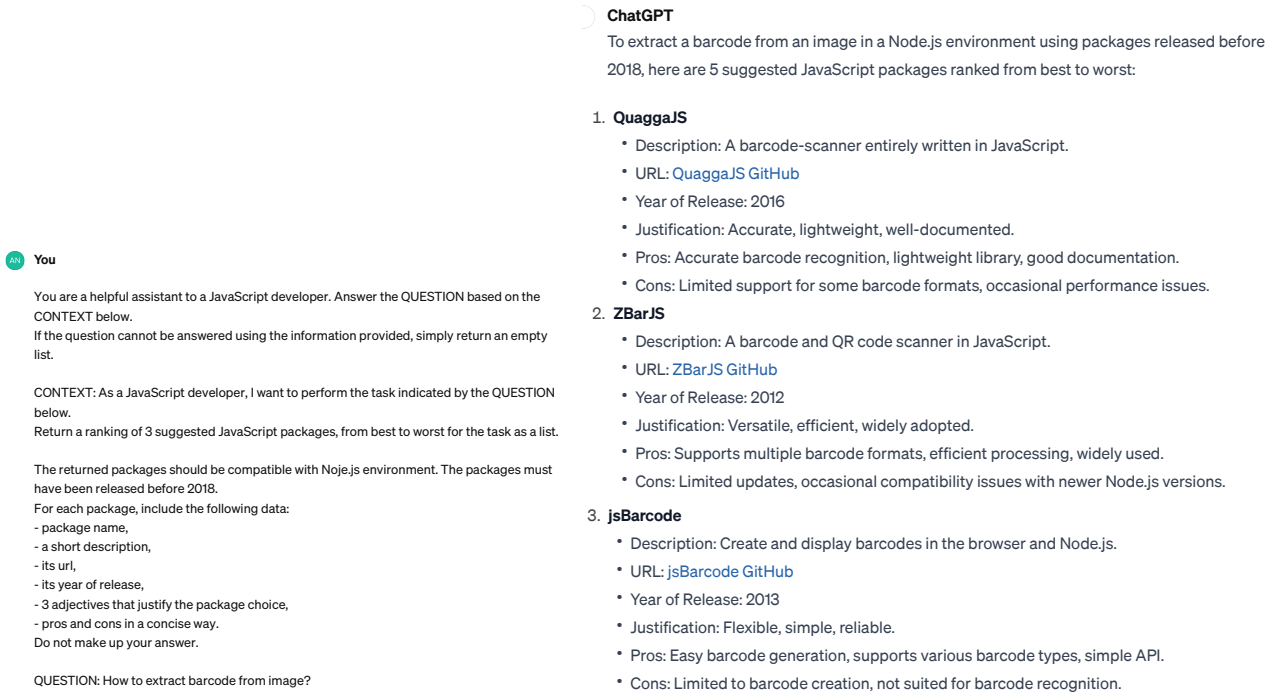


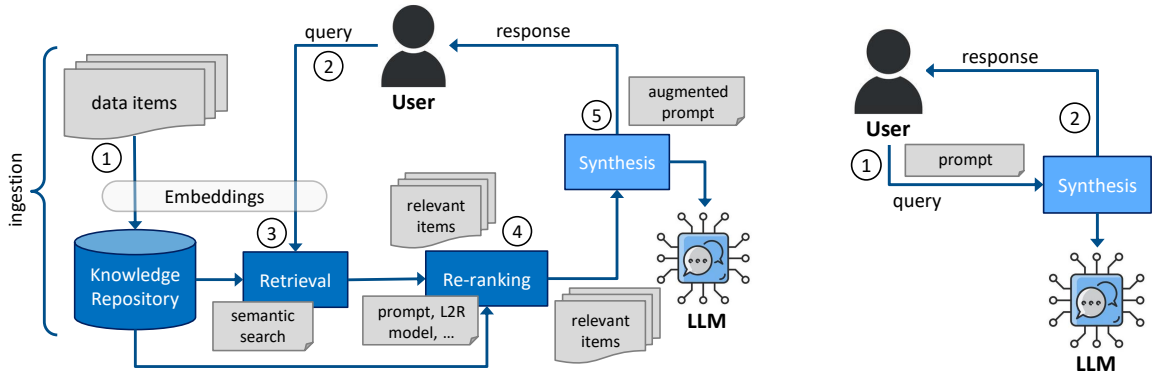
Figure 3: Example of JS package selection task with `ChatGPT` using a zero-shot strategy.

with auxiliary information are primarily categorized into two types [21]: generation-augmented and retrieval-augmented approaches. Generation-augmented methods instruct LLMs to generate a context tailored to the question, which serves as the basis for the final answer [17]. Retrieval Augmented Generation (RAG) methods, in turn, involve the integration of external (often domain-specific) data into the generative process of the LLM to enhance the accuracy and relevance of responses. For instance, in our motivating example, `ChatGPT` could be based (or grounded) on a repository of JS technologies to retrieve candidate packages for the developer's query.

The basic workflow for a RAG architecture is depicted in Figure 4a, which comprises four main functions: data ingestion, retrieval, re-ranking, and synthesis. In this scenario, the user submits a query (2), e.g., about a technology need for a JS task. Although trying to answer this query directly through a prompt for the LLM is possible, as in the zero-shot schema (Figure 4b), the LLM is usually constrained by its pre-training data and can lack sufficient knowledge (or context) to answer the query appropriately. RAG addresses this gap by departing from a knowledge base in which data (e.g., information about JS technologies) have been ingested beforehand (1) and retrieving a list of data items that are similar to the user's query (3). For the retrieval, semantic search is the standard technique for computing similarity between the query and the candidate data items, based on embeddings and cosine similarity. As an additional step, the retrieved data items can undergo a re-ranking process (4) to improve the identification of relevant data items (by moving them to the top of the list). Once the top-k relevant data items are retrieved, these items, along with the initial query, are merged into an enriched prompt (5), which enables the LLM to synthesize an informed response. Note the difference with the zero-shot workflow in Figure 4b, in which the user's query is directly submitted as part of the prompt to the LLM (1), which then generates a response to the users (2).

Coming back to the *vanilla* `AIDT`, its architecture shares some (but not all) of the RAG components. On one hand, the *ST-Retrieval* module fulfills the role of the retriever, while the *ST-Rank* performs the re-ranking function. On the other hand, the missing components are: the usage of semantic search for the retrieval and the synthesis of responses using an LLM. Also, the RAG architecture is not restricted to using an ML model for item re-ranking, and other variants can be supported. Thus, we argue that the RAG architecture enables new opportunities for improving the recommendation and selection of JS technologies.

Figure 5 shows how the prompt for `ChatGPT` differs when using a RAG strategy. The main difference with Figure 3 is that an initial list of candidate JS packages is provided for the LLM to work with. This exemplifies the grounding aspect of the RAG, because it reduces the range of options for the LLM and helps it keep the response focused.



(a) RAG workflow

(b) Zero-shot workflow

Figure 4: Differences between a RAG and a zero-shot strategy using LLMs.

4 Study Design

This work aims to assess the feasibility of using assistive tools, like `AIDT` and LLM-based techniques, to deal with the JS package selection task. In prior work [3], we performed an initial analysis towards that goal using a reference set of 16 common JS queries and making comparisons with rankings produced by JS developers. The analysis yielded interesting results with respect to the generated rankings; however, it was limited only to `ChatGPT` using a zero-shot strategy. In this paper, we extend the scope of the evaluation in two aspects: (i) we include two additional LLMs (`Cohere` and `Llama2`), and (ii) we incorporate the RAG strategy as an enhancement over zero-shot.

We address the following research questions:

- **RQ#1**: Are the rankings produced by the `AIDT` and `LLM` tools better than those produced by JS developers?
- **RQ#2**: Are there differences in the rankings of `AIDT` and the `LLMs` compared to the human rankings?
- **RQ#3**: Does the RAG strategy (using LLMs) perform better than the zero-shot one?
- **RQ#4**: Which selection criteria are considered by the JS developers and the `LLM` tools?

We performed four types of experiments to answer these questions, as depicted in Figure 6. The first two experiments are reported in [11], while the third and fourth experiments assess the zero-shot and RAG strategies for three LLMs (`ChatGPT`, `Cohere` and `Llama2`). A reproducibility kit for the experiments and the proposed RAG architecture is provided at `https://github.com/tommantonela/aidt_rag`. The experiments involved three phases: preparation, testing and analysis. The activities performed at each phase varied depending on the nature of the experiment. For all the experiments, we defined a baseline (or ground truth) consisting of set of queries along with their reference rankings (of JS packages). NPM was used as the de-facto JS repository. For this baseline, we asked two senior developers to record any queries in NPM that they would make in their projects for a period of two weeks. In this way, we obtained a reference set of 25 queries that represent a variety of technological needs, as listed in Table 2. After that, these senior developers provided two sets of JS packages, both relevant and non-relevant ones, for the reference queries. The reasons for including the packages in each set were thoroughly discussed until reaching consensus. Two of the authors participated in refining the results until the reference sets of relevant (hits) and non-relevant packages for each query were established.

During the testing phase, each experiment required executing the queries above and collecting the recommended packages. For answering **RQ#1**, **RQ#2** and **RQ#3**, the performance for the four experiments was evaluated using traditional metrics such as precision, recall, MAP and nDCG, based on the reference rankings. Precision measures the proportion of relevant packages among all retrieved packages, providing a snapshot of how accurately the system identifies relevant content. Recall, on the other hand, assesses the system's ability to retrieve all relevant packages. MAP evaluates the average precision across all relevant packages retrieved by the system. nDCG considers the positions of relevant packages in the ranking order, emphasizing the importance of ranking accuracy beyond mere relevance identification. Together, these metrics provided a comprehensive framework for assessing not only the relevance of retrieved packages but also the effectiveness of their ranking positions, offering valuable insights into the performance variations across the different experiments. A cut-off threshold $k = 5$ was defined to select the top-k recommended items[11].

[11] Results for other thresholds $k = 3$ and $k = 7$ can be found in the companion repository.

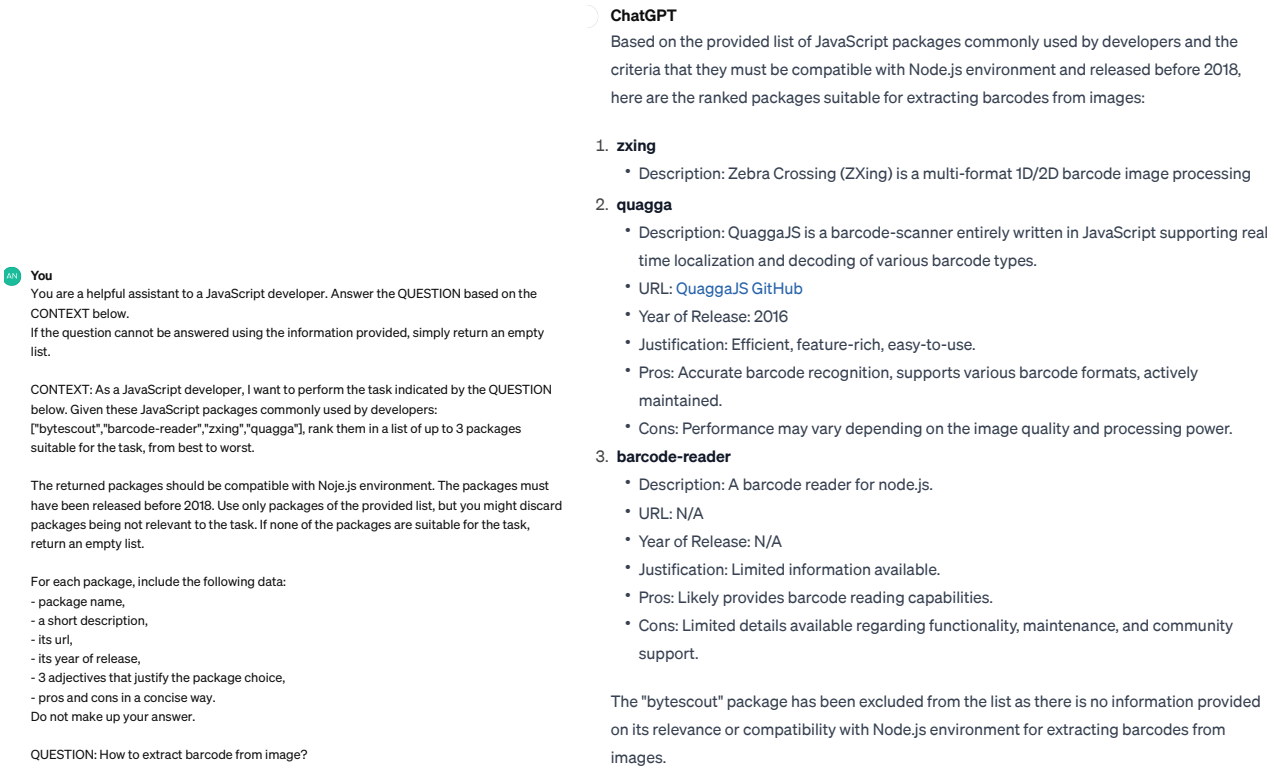


Figure 5: Example of JS package selection task with `ChatGPT` using a RAG strategy.

Table 2: Set of reference queries (25) used in the experiments.

Queries		
check valid email address	download web videos	send sms
quick sort algorithm	filter adult content images	user authentication
extract barcode from image	convert data formats	download free music
convert typewritten image to text	sentiment analysis	third party authentication
convert text to speech	calculate word similarity	translate English to Spanish
credit card validation	health tracker	captcha authentication
detect text language	rank aggregation algorithms	mobile app framework
DOM manipulation utils	lightweight 3D graphic library	mathematical functions
	scraper	

Thus, all participants, `AIDT` and the LLMs worked with 5 packages for each query. Alongside relevance metrics, we performed paired (either ttest or Wilcoxon, depending on data normality) statistical tests (with $\alpha = 0.05$), and used Cohen's d to quantify the magnitude of effects to determine the significance of the observed differences.

In this experimental setting, the participants did not interact with the tools (e.g., NPM, `AIDT`, `ChatGPT`, `Cohere`, `Llama2`) directly, but rather the research team did it. This decision tried to reduce the effects of tool learning or user-experience (UX) aspects and focus on the performance of the task. For instance, the UX design of `AIDT` is not as intuitive as that of the LLM-based tools. For the RAG and zero-shot strategies, we also preferred to make the comparisons using predetermined, parametrized prompts, rather than letting participants write their own prompts in order to control for prompt variability. This experimental uniformity had the tradeoff of making the technology selection scenario somehow less realistic.

The experiments involving LLMs were implemented on top of the *Langchain* framework[12], which provides a common set of abstractions for constructing LLM-powered applications and integrates with different LLMs. In particular, this framework supports a common format for prompts, which internally gets translated to the details required by each LLM.

[12] `https://www.langchain.com/`

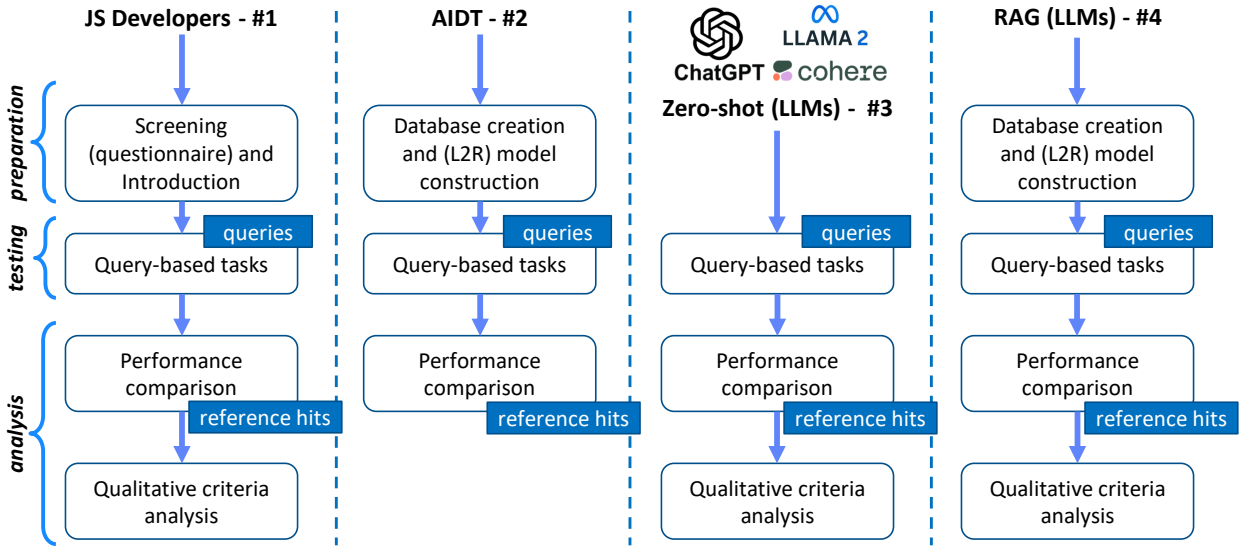


Figure 6: Experimental design for humans, `AIDT` and `LLMs` (zero-shot and RAG strategies).

4.1 Experiment #1: JS Developers

This experiment was carried out with 21 participants, who were asked to select and assess JS packages for a set of reference queries. These participants were students from a graduate university course (in Argentina) and most of them had at least 5 years of development experience. The participants were knowledgeable in object-oriented programming, Java and JS stack. During the preparation phase, we performed an initial screening to determine the participants' level of knowledge and expectations with respect to the study. In addition, each participant received a short introduction to the context of the JS package selection task.

The participants were (randomly) assigned 5 queries each. Internally, we ensured a coverage of all the queries in the set. The queries were phrased as simple as possible. For each assigned query, a participant had to select at least 5 JS packages. In the activity, participants had freedom to choose whatever search engine they judged convenient. They also had to indicate the reasons for selecting and ranking the packages. The maximum time allotted for the activity was 60 minutes. At the end, participants had to complete a post-mortem questionnaire about their level of satisfaction and opinions.

4.2 Experiment #2: `AIDT`

To configure and train the *ST-Retrieval* and *ST-Rank* modules, we initially downloaded the package registry from NPM and built a dataset of technologies up to a given date (September 2017). We ran *ST-Retrieval* 25 times on the reference set (once per query) and stored the aggregated lists of packages. When processing the results, we considered the first 20 documents from the lists of packages, as users searching the Web (e.g., using Google) are very likely to consider only the first pages. We ended up with a total of 2760 retrieved JS packages. The dataset was enriched with package features collected from GitHub. Furthermore, we relied on NPM to get each package's features.

For *ST-Rank*, we created a set with ≈ 250 rankings, each one having between $2-6$ packages. In total, more than 1000 training instances were obtained. To validate the rankings produced by the ML model, we split them into training and test sets with the usual $80-20\%$ partition rule of ML tasks. For the test set, we randomly selected 20% of the training rankings (and their corresponding training instances). Two senior developers verified these instances. The remaining 80% of the training instances constituted the training set for building the *GBRank* model. A k-fold cross-validation ($k = 5$) was performed to determine the best configuration of hyper-parameters for the model. More details about the construction of the `AIDT` pipeline can be found in [3]. It should be noticed that the ML model should be periodically re-trained with new queries and features from the open-source JS community to keep the recommendations up-to-date.

Since the modules of `AIDT` work as a black box regarding the predicted packages, it was not possible to perform a qualitative analysis of the outputs in light of **RQ#4**.

4.3 Experiment #3: `LLMs` using a zero-shot strategy

The concrete LLM models used for this experiment are summarized in Table 3. The prompts take the form of a chat conversation in which the user role (i.e., a JS developer) asks the model to perform a task

Table 3: Summary of the LLM models used in the experiments

	Model	Usage	Tokens	Training data
`GPT-3.5`	GPT-3.5-turbo	Traditional text completion tasks and chat interactions.	4,096 max	Up to January 2022
`Llama2`	Llama2-7b	Text completion and chat interactions	4,096 max	Up to July 2023
`Cohere`	Coral	Chat interactions	1,024 max	Retrained weekly

(i.e., a technology query), and the model plays the role of a JS assistant, as exemplified in Figure 3. We analyzed and refined different alternatives for the prompt instructions, according to well-known guidelines[13] and examples from the recommender systems literature [22].

For the testing phase, we designed a schema to describe a JS-related requirement and asked the LLM to recommend up to 10 packages for the target query. We additionally extended the prompts to get a justification of the recommendations. In line with **RQ#4**, we intended to understand the selection criteria suggested by the LLM for each item. To make results comparable and avoid missing packages, we included a restriction in the prompt requiring all recommended packages to have been published before 2018. For supporting comparisons with the explanations of JS developers, we asked the LLMs to provide three qualities for each retrieved package in the form of adjectives, and also a brief mention to pros and cons of the package.

4.4 Experiment #4: `LLMs` using a RAG strategy

This experiment is a variant of the previous one, in which the prompt was modified in such a way it can be grounded on the results of the semantic retrieval of the RAG. The prompt schema was exemplified in Figure 5. Note that, in this case the LLM can disregard some of the retrieved packages if they are judged irrelevant for the query.

In our instantiation of the RAG architecture for the task that `AIDT` performs, we reused the repository of JS technologies crawled in [3]. In addition to providing a textual description of each package, this database is enriched with different package features (e.g., popularity, number of downloads, number of contributors, lines of code, existence of test cases, etc.). The package descriptions are represented using a pre-trained embedding[14]. We measure the semantic similarity between the developer's query and each package through the cosine similarity between the package description embedding and the embedding of the query. Once a list of packages is retrieved, they can be optionally re-ordered using the *GBRank* model [5]. Alternatively, the LLM can be instructed to perform the re-ordering in conjunction with the explanation of the reasons (i.e., key characteristics, pros and cons) for selecting the packages.

5 Evaluation and Findings

During the analysis phase, we evaluated the results of the four experiments. The analysis focused both on performance aspects (e.g., relevance of the recommended packages using metrics such as precision, recall, and nDCG) as well as on qualitative ones (e.g., differences between rankings, and selection criteria for the JS packages).

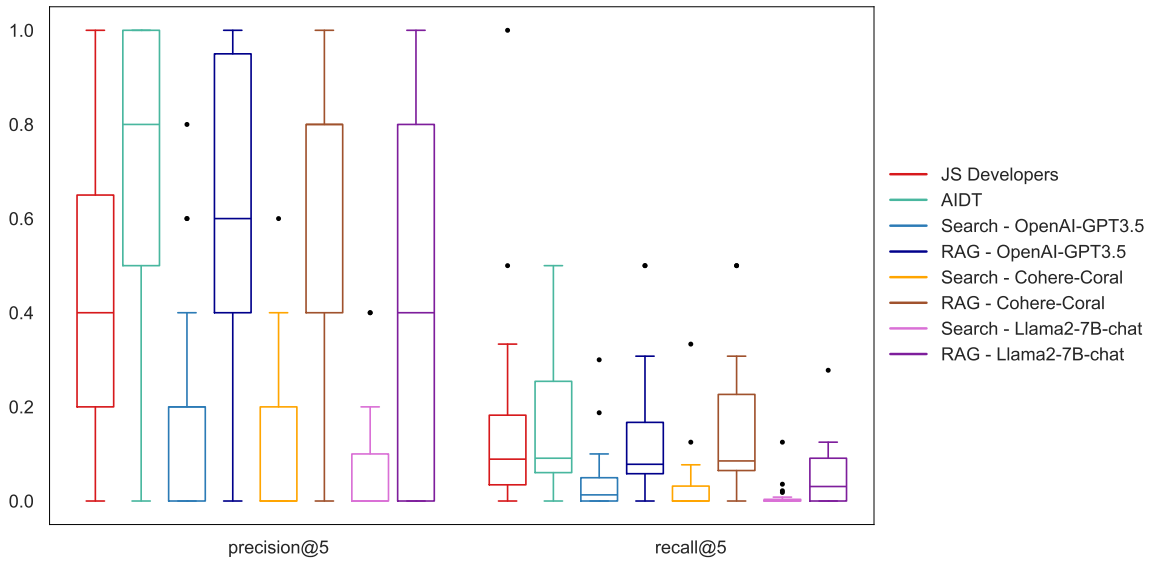
5.1 Performance

The distribution of results for the JS packages selected for all queries is summarized in Figure 7, including the experiments with the JS developers, `AIDT` and the LLM variants, respectively. Figure 7a shows the relevance of the returned packages (with respect to the baseline) in terms of precision and recall, while Figure 7b shows the quality of the package rankings using the MAP and nDCG metrics. In all cases, we considered the top-5 packages of each ranking.

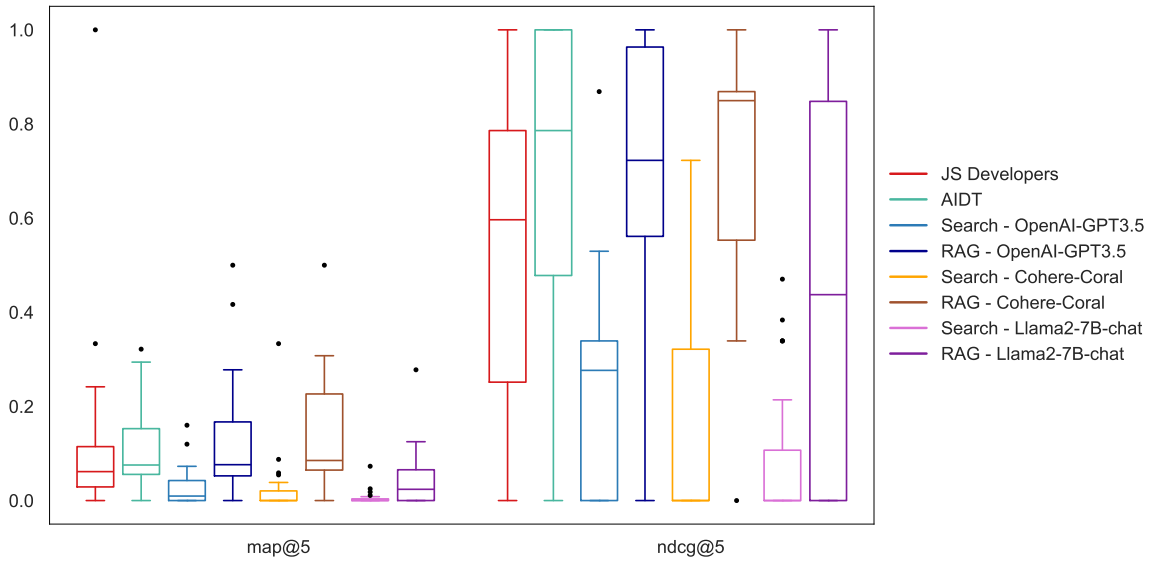
A noticeable effect is that `AIDT` achieved the best performance for all metrics, while the LLMs exhibited variations in their performance depending on whether the RAG architecture was in place. The zero-shot strategy had, in general, a lower performance than the human participants. However, the RAG strategy was almost as good as `AIDT` when using `ChatGPT` and `Cohere`, and lower than `AIDT` when using `Llama2`. This difference can be attributed to the grounding effect of the RAG for the prompts, which increases the chances for the LLM to return relevant packages at the top of the rankings. In particular, the differences between precision and recall for the participants, `AIDT` and the RAG approaches indicate that even when

[13] `https://www.promptingguide.ai`

[14] `https://huggingface.co/sentence-transformers/paraphrase-MiniLM-L6-v2`



(a) Relevance metrics



(b) Ranking metrics

Figure 7: Comparison of performance metrics.

non-relevant packages are recommended, the relevant ones are ranked high by all the alternatives, with the exception of the zero-shot variants. The high nDCG values confirm this trend.

For precision, we tested each alternative against the precision obtained by the JS developers, and found out that the differences in favor of `AIDT`, `ChatGPT` and `Cohere` were statistically significant ($p < 0.05$) with a large effect size. This trend also held for the nDCG values. However, the trend did not hold for `Llama2`, neither for precision nor nDCG. There were no statistically significant differences between `AIDT`, the LLM models and the JS developers, with respect to recall and MAP values. Although both MAP and nDCG assess the package order in the rankings, they treat decreasing ranks differently.

> We can answer **RQ#1** by saying that `AIDT` exhibited a superior performance compared to both the JS developers and the LLMs, particularly in terms of relevant packages. The LLMs seem to require contextual information to produce accurate recommendations, as promoted by the RAG architecture. The recommendations offered by `ChatGPT` and `Cohere` were quite close in performance to those of `AIDT`.

We observed that for some queries the `AIDT` and LLM alternatives did not retrieve any of the relevant packages. For example, developers did not select any of the relevant packages for "*lightweight 3D graphic library*", `AIDT` did not recommend any relevant package for "*translate English to Spanish*", "*calculate word*

similarity", "*convert typewritten image to text*", "*filter adult content images*" and "*lightweight 3D graphic library*", while `ChatGPT` was unable to recommend any relevant package for almost half of the queries. Interestingly, `ChatGPT` made recommendations of relevant packages for two of the queries in which `AIDT` failed. This situation might imply that not all queries are easy to satisfy. A manual inspection revealed that in some cases, only few packages were relevant (e.g., there was only one relevant package for the "*translate English to Spanish*" query), which hindered the achievement of that task. In some LLMs, like *GPT*, specific queries such as "*download free music*" or "*download web videos*" did not produce results, because the *ChatGPT* policy detected as requiring privacy-related information or potentially-illegal questions (e.g., music downloading).

5.2 Differences in rankings

When comparing the rankings from the JS developers and `AIDT`, we observed that `AIDT` achieved better results than the developers, with average improvements of 100% and 35% for precision and nDCG, respectively. We observed some exceptions for the "*mathematical functions*" and "*translate English to Spanish*" queries, for which the developers outperformed `AIDT`. Nonetheless, there was a noticeable deviation in the precision and nDCG values regarding the zero-shot LLM variants. The largest differences were obtained for the "*quick sort algorithm*" query. A manual inspection of the developers' rankings revealed non-existing packages (i.e., items that did not belong neither to the relevant nor to the non-relevant sets of the ground truth), whose names partially matched more than one real package. This fact made it difficult to distinguish which packages the developers referred to and, consequently, underestimated their performance. The packages missed by the LLMs (under a zero-shot strategy) can be related to the fact that the sources used for training the models could have included a bigger recommendation space than the one analyzed by the JS developers, the experts (for the baseline), or `AIDT`, leading to candidate packages that were unknown to the other parties. There is also a possibility that the LLMs might have returned inaccurate package names or even non-existing packages [18].

Regarding the rankings for the developers and the RAG variants, we noticed that developers achieved better results in 12 queries. The largest differences were observed for the "*mathematical functions*" query. As mentioned above, `ChatGPT` was prone to return no packages when identifying potentially legal issues. The other two LLMs were more permissive with respect to this aspect and did recommend a few packages. When comparing the rankings of `AIDT` and the RAG variants, the former outperformed the latter in 10 queries, while achieving the same performance in the remaining queries. On average, `AIDT` had an improvement over `ChatGPT` (RAG) of 35% and 15% in terms of precision and nDCG, respectively. The differences were higher in favor of `AIDT` for `Cohere` and `Llama2`.

The changes in the prompt used by the RAG strategy with respect to the zero-shot prompt can explain the better performance of the RAG strategy. The RAG prompt helps the LLM to narrow down the search space. This effect is remarkable in the precision and nDCG values of the LLMs, and smaller but still visible for the recall and MAP values. These results evidence the importance of prompt design and how contextual information can positively contribute to the target task.

Another observation of our experiments was the flexibility for altering the behavior of the LLM-based variants (with respect to the recommendation task), depending on how the prompts are crafted. Unlike the LLMs, the behavior of `AIDT` is not changeable once its internal ML is built.

> The analysis of the package rankings produced by the tool alternatives shed light on their performance, thus contributing to answering **RQ#2**. We observed quantitative differences in the rankings generated by the JS developers, `AIDT`, and the LLM variants. For `AIDT` and the LLMs, the differences seem to be caused by their internal workings.

> For the subset of experiments based on LLMs, we answer **RQ#3** by saying that using a RAG architecture produced more accurate rankings than using a zero-shot strategy. The main difference between the two strategies is that the RAG departs from a specific knowledge source (i.e., a curated dataset of JS technologies) that informs the generation of recommendations.

5.3 Justifications for the recommendations

As mentioned in Section 4, the JS developers recorded the main aspects they considered for selecting the packages. The prompts for the LLM variants asked for similar information for the returned packages, including also a summary of pros and cons of choosing each package. Figure 8 summarizes the most common criteria for the three LLMs and the two strategies (zero-shot and RAG). To facilitate the analysis, we unified these criteria into a set of common, most-frequent words. The circle size indicates the word frequency

(across all queries), while the color shows its source. Words shared by both the JS developers and the LLM under consideration are marked in green, although they were rare in our experiments. In most cases the words being used differed, but we noticed that they often referred to similar topics. Thus, we computed the semantic distance between the words within a set using FastText[15] embeddings and created 2D visualizations based on multi-dimensional scaling (MDS). MDS is a dimensionality reduction technique that constructs a representation that considers the distances between objects (i.e., words in our context). The interpretation is that words that are more similar (or have shorter distances) appear closer on the chart than words that are dissimilar (or have larger distances).

In general, the LLM justifications were more verbose than those of humans. Based on the semantic distance criterion, we observed that topics from the LLM (in blue) and the JS developers (in red) were more likely to appear closer when using a zero-shot strategy and fall apart when using a RAG strategy. Furthermore, for the zero-shot strategy, and despite the choice of the LLM, we found exact matches between the developers and the LLMs for the words "modern" and "popularity". We conjecture that this effect can be due to the (lack of) RAG grounding, although it needs further investigation. Certain topics (e.g., popularity, maintenance, quality, or usability) had a comparatively higher frequency for the developers than in the LLM responses. While this might indicate a human emphasis, it does not mean that the LLMs did not consider such criteria.

A further inspection of the developers' responses revealed that not every developer justified every package they chose. Instead, they tended to provide criteria for the first three packages (out of five). In most cases, criteria were expressed using one single word or expression, while only a few developers wrote longer phrases or even paragraphs. For most criteria, it was clear when the developers highlighted a positive aspect of a given package, although the expressions used were ambiguous in some cases. For example, this was the case of the "dependencies" topic, in which it was unclear whether developers were referring to packages having a low (i.e., a positive aspect) or high number (i.e., a negative aspect) of dependencies. Things got worse when a developer used that sole criterion for assessing multiple packages. In addition, other developers defined "best overall" as a criterion without much description of its meaning, which made the topic not comparable to others. We noticed that this subjectivity in the criteria was less apparent in the LLM responses, which might explain the higher topic diversity in all the charts (in blue). The selection criteria given by the LLMs appeared repeatedly across `GPT`, `Cohere` and `Llama2`, and most topics seemed to respond to positive characteristics of the packages. The causes for the low topic repetition (compared to the humans' topics) are unclear, and might be related to the defined prompts.

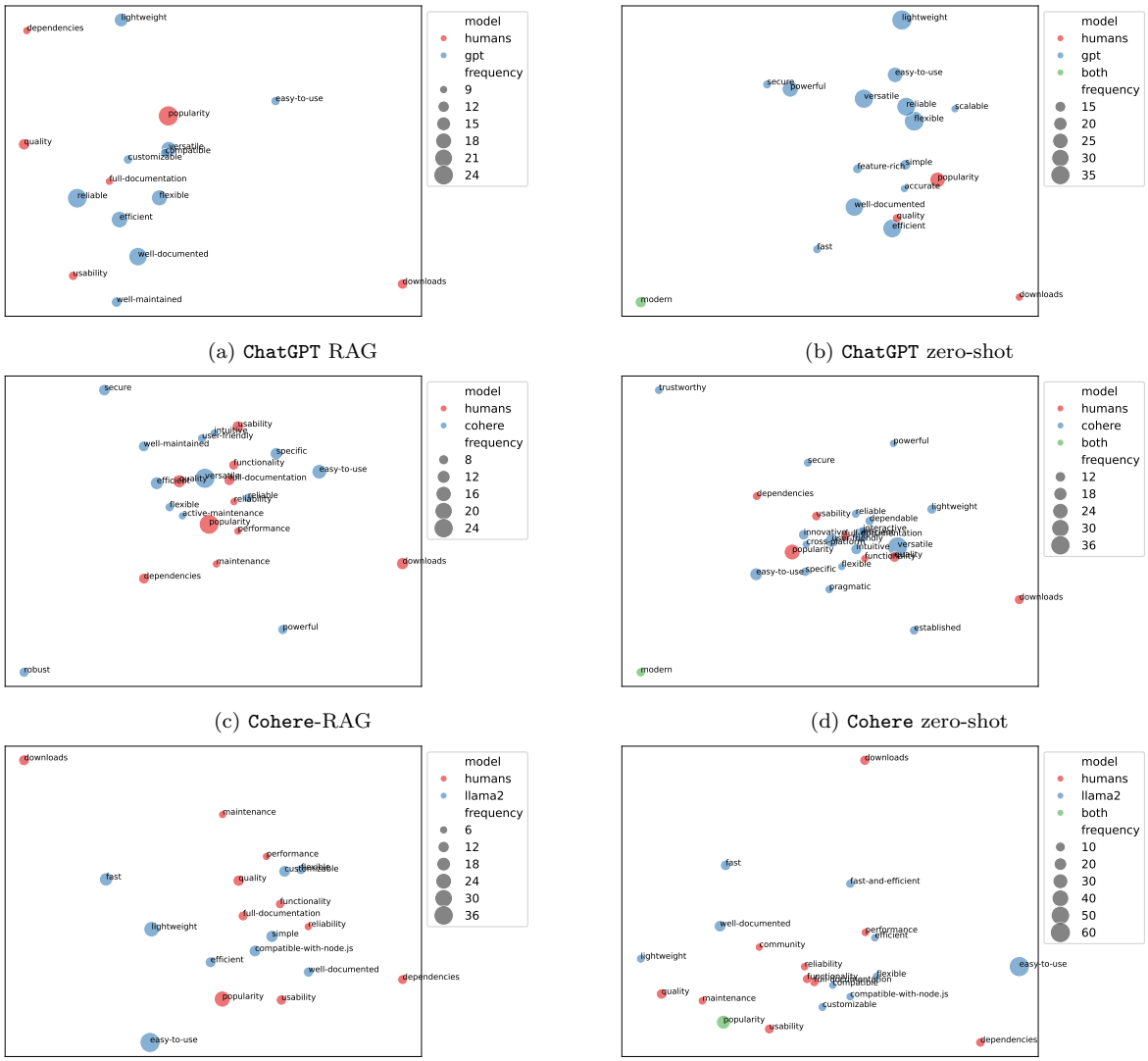
> Regarding **RQ#4**, the analysis of the selection criteria for the JS developers and the LLMs revealed a reasonable agreement on the semantic meaning of the topics used. We observed more uniformity (i.e., less ambiguity and repetition) in the topics given by the LLMs than in those expressed by humans. Furthermore, the topics returned by the RAG strategy seemed more diverse than those returned by the zero-shot strategy.

5.4 Discussion

Our study has several implications in terms of tool support and personalized assistance for JS developers. The proposed RAG architecture can improve developers' productivity by simplifying JS package evaluation and selection, and thus reducing technology fatigue. Although the final decision is up to the developer, tools like `AIDT` contribute to make more informed decisions, particularly when the recommended items are accompanied by explanations. Having explanations, based on the information being retrieved by the RAG and synthesized by the LLM, adds transparency and provides a rationale for the decision-making process, which is a clear improvement over traditional search engines. Along this line, our experiments showed a very good precision of the RAG architecture, surpassing the zero-shot strategy. Comparing the precision of the RAG-based retrieval against the human performance (for the same task) also yielded positive outcomes. Additional levels of personalization can be added to the current assistant, in order to consider goals and contextual information from the developer.

The inclusion of several LLMs shows that the RAG architecture can be adapted to different contexts, which is a typical requirement in a practical development setting. On the downside, our experiments showed performance variability in the LLMs (e.g., for `Llama2`), which indicates that the benefits of the RAG-based `AIDT` need to be weighted based on its underlying LLM and prompting for the target tasks. Although it is envisioned that LLMs might get better over time, establishing an evaluation benchmark for the tasks (to be automated) is crucial for a mainstream adoption of `AIDT` or similar tools. For instance, an issue with the `AIDT` explanations which we have not addressed yet is whether they are valid (e.g., they do not

[15] `https://fasttext.cc/`



(a) `ChatGPT` RAG

(b) `ChatGPT` zero-shot

(c) `Cohere`-RAG

(d) `Cohere` zero-shot

(e) `Llama2` RAG

(f) `Llama2` zero-shot

Figure 8: Main words used by the JS developers, and the RAG and zero-shot strategies (LLMs) to support the package recommendations. The layout of the words accounts for their semantic similarity.

contain hallucinations) or trustable for the developers. Evaluation benchmarks can help to mitigate this problem. The success of the RAG architecture also depends on its reliance on a comprehensive knowledge base (e.g., Github, or any other platform, even a corporate one) to ensure the quality of the retrieved items and posterior explanations. In our experiments, we constructed this base only once, but in practice it should be frequently updated and curated by a dedicated team.

5.5 Threats to Validity

A number of threats to internal, construct, and external validity were identified in our study, which we tried to mitigate whenever possible.

A first threat to construct validity, is related to the queries and technology searches used in the experiments. We intended to use queries and search criteria representative of real-world JS development. Having two senior developers providing the queries and checking the results, along with the authors' intervention to refine them, might have biased the chosen queries and packages (e.g, by the type of software projects they usually work on). Along this line, for `AIDT`, we collected a dataset from the NPM and Github repositories. Despite the low number of queries, 2760 JS packages were returned by the search engines and were manually analyzed. Since analyzing query results might take a substantial amount of time from experts, we preferred not to do a detailed query analysis in this work. To mitigate threats to external validity, we considered queries with different purposes in the experiments. However, other queries or query phrases for specific domains could have been used. Additional experimentation and surveys with JS developers are still necessary

in this regard.

Related to `AIDT`, the usage of *Borda Fuse* in *ST-Retrieval* to rank the package lists is a threat to internal validity, as this method might have biased the ranking of items and might have affected the outputs of *ST-Rank* as well. Applying alternative aggregation methods could have generated different package orderings.

Related to the LLM variants, the model could generate multiple responses for the same query. This variation can be related to how the prompt is provided, the training data, the grounding repository, or even an algorithmic bias. This constitutes a threat to construct validity, as the consistency of the recommendations or the package characterizations (topics) could have been impacted. A potential mitigation is to refine the initial query in iterations to ensure more consistent responses. Along this line, we showed how prompts can be refined for the RAG architecture to achieve a better performance (e.g., in terms of relevance or quality) in the results. Further improvements to the RAG architecture, such as a re-ranking mechanism of the retrieved results, need to be explored.

At last, there is an external validity threat regarding the generalizability of our findings. The experimentation with three different LLMs contributes to mitigate this threat. Nonetheless, since each LLM has its own training process, they could produce different results even when the same instructions were used in our prompts. Thus, additional evaluations and user studies are necessary to support our claims. We would like to replicate the experiments with JS developers having different seniority, and allowing them to interact with the tool and adjust its recommendations.

6 Related Work

Various techniques have been developed to help select software technologies [23]. Typically, these techniques involve compiling a list of technologies, comparing them, and presenting a ranking to decision-makers. Certain studies have focused on appraising pre-existing technologies but have disregarded searching and acquiring technologies from (Web) repositories. For instance, Ernst et al. [24] proposed a score-card that assists developers in selecting a particular component from a group of predefined candidate components. This score-card considers evaluation criteria such as performance, maintenance, and community support.

Software repositories [25] are one of the primary resources for finding technologies. However, current repositories have not been particularly successful in this regard, as their search engines often do not provide the desired outcomes. Several studies have attempted to enhance the search mechanisms provided by repositories. A few studies share similarities with our approach. Dolphin [26] considers open-source projects, which are ranked based on the extent of their impact (and how frequently they are mentioned) in forum communities, such as StackOverflow[16] or OSChina[17]. In this sense, Dolphin only considers open-source projects obtained from version control repositories.

LibFinder [27] employs multi-objective optimization to recommend Java libraries from GitHub and Maven[18] repositories based on source code. Nonetheless, search and recommendation are not guided by user queries. Instead, recommendations are made based on analyzing the source code, aiming at discovering libraries that could replace specific code fragments. Soliman et al. [28] developed an approach to retrieve architectural decisions and solution alternatives, employing StackOverflow as a repository of architecture knowledge. It is based on a correlation between text (queries) and a "de facto" ontology. Although interesting, the applicability of this approach to JS technologies is still to be demonstrated.

Chen et al. [29] proposed a recommendation technique that relies on a knowledge base extracted from curated Web resources (such as Q&A posts from StackOverflow). As in our approaches, queries are expressed in natural language, and, like in the RAG approach, similarity between the input and the candidate items is computed using embeddings. Li et al. [30] developed a related approach for searching JS code snippets implementing a particular feature. However, from a development standpoint, it should be noted that reusing snippets is not the same (nor has the same difficulty) as integrating JS packages. Similarly, Zhang et al. [31] aimed at recommending code snippets based on API descriptions and use cases created using ChatGPT.

Other works [32] have approached the ranking of technologies according to different criteria. Nonetheless, in most works, the ranking strategies are manually defined based on the features of the candidates. For example, Franch and Carvallo [33] developed a structured quality model for evaluating software packages. This model offers a taxonomy of quality characteristics and metrics for calculating its worth according to the domain at hand. Jadhav et al. [13] used an expert system to combine ranking strategies based on AHP. Instead of following a data-driven strategy, this approach requires experts to define the ranking rules. Finally, Grande et al. [23] conceptualized selection as a multi-objective optimization problem and solved it by means of genetic algorithms.

[16] https://stackoverflow.com/
[17] https://www.oschina.net/
[18] https://maven.apache.org/

Reports on using LLMs for software engineering tasks are relatively recent [8]. Assistive tools like ChatGPT can provide insights into how developers, users, and stakeholders interact through natural language, leading to enhancements in software development processes and results [34]. For example, ChatGPT could be used to identify test cases or test data, explain code fragments or models as a replacement for traditional documentation, or simulate user interactions with software systems to deal with user experience. In addition, ChatGPT has been shown to be able to perform on par of (novice) developers for simple coding tasks [35].

Ahmad et al. [36] studied the potential of ChatGPT to assist software architects. To this end, the authors presented a case study involving collaborations between architects and ChatGPT for the architectural analysis, synthesis, and evaluation of a microservices application. A preliminary evaluation showed that ChatGPT was able to imitate the architect's role to support an architecting process by processing user stories, articulating architectural requirements, specifying models, recommending tactics and patterns, and developing scenarios for architecture evaluation. Nonetheless, the experiment still needed a considerable dosage of human oversight and decision support. White et al. [37] also leveraged ChatGPT to try to automate common software engineering activities. The authors designed a catalog of prompt patterns covering requirements, system design and simulation, code quality, and refactoring tasks. Although this experience is potentially useful, the defined patterns have not been yet validated.

7 Conclusions

In this paper, we report on a series of experiments for the JS package selection task, in which we evaluated the results of a group of human developers, against those of the `AIDT` and three LLM alternatives. Both types of tools work as recommender systems for assisting developers in selecting, assessing and ranking relevant packages. Nonetheless, the tools have differences in their conception. While `AIDT` was explicitly designed for the task, LLMs are general-purpose, emerging models that can deal with this and other tasks. In particular, we exercised three LLMs using two strategies: zero-shot and RAG. A RAG architecture for the JS technology domain was proposed as an improvement of `AIDT`. We performed a comparison using a set of predefined queries for JS repositories, and then analyzed the rankings returned by each alternative. We also investigated whether the LLMs can argument about pros and cons of each recommended package, which is a limitation in the design of `AIDT`.

The results of the experiments were enticing. On one side, `AIDT` outperformed both the human developers and the LLMs, particularly in terms of precision and nDCG metrics. This might be due to the specialized knowledge of the JS domain captured by the ML model of `AIDT`. However, `AIDT` is unable to explain its rankings, which can compromise the developers' trust on the results. On the other hand, the LLMs showed a sub-optimal performance for the task, which seems to be in line with other experiments [8,18,36], but they were able to provide good arguments for its selection criteria. Regarding the zero-shot or RAG strategies, the latter achieved a better performance than the former, which was comparable to that of `AIDT`. For `AIDT` and two of the LLMs (`ChatGPT` and `Cohere`), their improvements over the results of JS developers were statistically significant. The results returned by the LLMs (and their performance, thereof) were affected by how queries were expressed in the prompt. Thus, there is a tradeoff between having a specialized model (like that of `AIDT`) versus a general-purpose one (like the LLMs). Based on our experience with the RAG architecture, we argue that LLMs can generate satisfactory package rankings for developers provided with an appropriate configuration (e.g., prompting, customized architecture). A related aspect of LLMs is that their recommendations might change according to the evolution of the available technologies, their usage and assessment by JS developers. This technology evolution can be seen as a concept drift scenario that an LLM-based recommender system should take into account.

Overall, although more evaluation is needed, our findings reveal a good opportunity for JS developers to rely on LLMs for the package selection task, as a less biased but still informative search engine. As future work, we plan to extend our study with subjects interacting directly with a chat interface, backed by a RAG architecture and a particular LLM, and also allowing newer JS packages as candidates for satisfying the queries. In addition, we will further investigate how to incorporate contextual information about the developer's need in the prompts, and how re-ranking mechanisms can help to improve the current rankings. A related line of research is an extension of `AIDT` to support explanations of its predictions [38], using the features of its underlying ML model. Another interesting work is the development of an LLM specialization [39] for the JS technology domain [29]. Finally, we envision that the RAG architecture for `AIDT` can be applied to technology repositories for other programming languages (e.g., Ruby, Python or Java, among others), such as the `library.io` platform.

Acknowledgments. The authors are grateful to the JS developers who participated in the experiments. This research is supported by project PICT-2021-00757, Argentina.

References

[1] H.-Y. Lin, P.-Y. Hsu, and G.-J. Sheen, "A fuzzy-based decision-making procedure for data warehouse system selection," *Expert systems with applications*, vol. 32, no. 3, pp. 939–953, 2007.

[2] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 351–361.

[3] H. C. Vazquez, J. Diaz-Pace, S. A. Vidal, and C. Marcos, "A recommender system for recovering relevant javascript packages from web repositories," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2023, pp. 175–185.

[4] J. A. Aslam and M. Montague, "Models for metasearch," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2001, pp. 276–284.

[5] H. Li, "Learning to rank for information retrieval and natural language processing," *Synthesis Lectures on Human Language Technologies*, vol. 4, no. 1, pp. 1–113, 2011.

[6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[7] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[8] B. Combemale, J. Gray, and B. Rumpe, "Chatgpt in software modeling," *Software and Systems Modeling*, May 2023. [Online]. Available: https://doi.org/10.1007/s10270-023-01106-4

[9] M. Abdullah, A. Madain, and Y. Jararweh, "Chatgpt: Fundamentals, applications and social impacts," in *2022 Ninth International Conference on Social Networks Analysis, Management and Security (SNAMS)*. IEEE, 2022, pp. 1–8.

[10] Y. Gao, T. Sheng, Y. Xiang, Y. Xiong, H. Wang, and J. Zhang, "Chat-rec: Towards interactive and explainable llms-augmented recommender system," *arXiv preprint arXiv:2303.14524*, 2023.

[11] H. C. Vazquez, J. A. Diaz-Pace, and A. Tommasel, "The javascript package selection task: A comparative experiment using chatgpt," in *2023 XLIX Latin American Computer Conference (CLEI)*, 2023, pp. 1–10.

[12] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[13] A. S. Jadhav and R. M. Sonar, "Framework for evaluation and selection of the software packages: A hybrid knowledge based system approach," *Journal of Systems and Software*, vol. 84, no. 8, pp. 1394–1407, 2011.

[14] A. Birk, "Modelling the application domains of software engineering technologies," in *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*. IEEE, 1997, pp. 291–292.

[15] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar, "Rank aggregation methods for the web," in *Proceedings of the 10th international conference on World Wide Web*. ACM, 2001, pp. 613–622.

[16] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun, "A general boosting method and its application to learning ranking functions for web search," in *Advances in neural information processing systems*, 2008, pp. 1697–1704.

[17] W. Yu, D. Iter, S. Wang, Y. Xu, M. Ju, S. Sanyal, C. Zhu, M. Zeng, and M. Jiang, "Generate rather than retrieve: Large language models are strong context generators," *arXiv preprint arXiv:2209.10063*, 2022.

[18] Y. Zhang, H. DING, Z. Shui, Y. Ma, J. Zou, A. Deoras, and H. Wang, "Language models as recommender systems: Evaluations and limitations," in *I (Still) Can't Believe It's Not Better! NeurIPS 2021 Workshop*, 2021.

[19] J. Albrecht, E. Kitanidis, and A. J. Fetterman, "Despite" super-human" performance, current llms are unsuited for decisions about ethics and safety," *arXiv preprint arXiv:2212.06295*, 2022.

[20] M. Kang, J. M. Kwak, J. Baek, and S. J. Hwang, "Knowledge graph-augmented language models for knowledge-grounded dialogue generation," *arXiv preprint arXiv:2305.18846*, 2023.

[21] H. Tan, F. Sun, W. Yang, Y. Wang, Q. Cao, and X. Cheng, "Blinded by generated contexts: How language models merge generated and retrieved contexts for open-domain qa?" *arXiv preprint arXiv:2401.11911*, 2024.

[22] Y. Hou, J. Zhang, Z. Lin, H. Lu, R. Xie, J. McAuley, and W. X. Zhao, "Large language models are zero-shot rankers for recommender systems," 2024.

[23] A. D. S. Grande, R. D. F. Rodrigues, and A. C. Dias-Neto, "A framework to support the selection of software technologies by search-based strategy," in *Tools with Artificial Intelligence (ICTAI), 2014 IEEE 26th International Conference on*. IEEE, 2014, pp. 979–983.

[24] N. Ernst, R. Kazman, and P. Bianco, "Component comparison, evaluation, and selection: A continuous approach," in *International Conference on Software Architecture Workshops*. IEEE, 2019.

[25] N. Clayton, R. Biddle, and E. Tempero, "A study of usability of web-based software repositories," in *Proceedings International Conference on Software Methods and Tools. SMT 2000*, 2000, pp. 51–58.

[26] Y. Zhan, G. Yin, T. Wang, C. Yang, Z. Li, and H. Wang, "Dolphin: A search engine for oss based on crowd discussions across communities," in *Software Engineering and Service Science (ICSESS), 2016 7th IEEE International Conference on*. IEEE, 2016, pp. 599–605.

[27] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[28] M. Soliman, M. Galster, and M. Riebisch, "Developing an ontology for architecture knowledge from developer communities," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 89–92.

[29] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions – incorporating relational and categorical knowledge into word embedding," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 338–348.

[30] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for javascript frameworks," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 690–701.

[31] K. Zhang, G. Li, J. Li, Z. Li, and Z. Jin, "Toolcoder: Teach code generation models to use apis with search tools," *arXiv preprint arXiv:2305.04032*, 2023.

[32] J. Klein and I. Gorton, "Design assistant for nosql technology selection," in *Proceedings of the 1st International Workshop on Future of Software Architecture Design Assistants*. ACM, 2015, pp. 7–12.

[33] X. Franch and J. P. Carvallo, "A quality-model-based approach for describing and evaluating software packages," in *Proceedings IEEE Joint International Conference on Requirements Engineering*. IEEE, 2002, pp. 104–111.

[34] M. A. Akbar, A. A. Khan, and P. Liang, "Ethical aspects of chatgpt in software engineering research," *arXiv preprint arXiv:2306.07557*, 2023.

[35] N. Nascimento, P. Alencar, and D. Cowan, "Comparing software developers with chatgpt: An empirical investigation," *arXiv preprint arXiv:2305.11837*, 2023.

[36] A. Ahmad, M. Waseem, P. Liang, M. Fahmideh, M. S. Aktar, and T. Mikkonen, "Towards human-bot collaborative software architecting with chatgpt," *ArXiv*, vol. abs/2302.14600, 2023.

[37] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," *arXiv preprint arXiv:2303.07839*, 2023.

[38] Y. Zhang and X. Chen, "Explainable recommendation: A survey and new perspectives," *Found. Trends Inf. Retr.*, vol. 14, no. 1, p. 1–101, mar 2020.

[39] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–10. [Online]. Available: https://doi.org/10.1145/3520312.3534862