Preprints.org

Article

Not peer-reviewed version

An LLM-Based Methodology for RESTful Service Publication and Discovery

Antonios Smardas and Kyriakos Kritikos *

Posted Date: 3 June 2026

doi: 10.20944/preprints202606.0323.v1

Keywords: LLM; service discovery; service matchmaking; REST; methodology; embeddings; accuracy; evaluation

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC, OpenAlex.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

An LLM-Based Methodology for RESTful Service Publication and Discovery

Antonios Smardas and Kyriakos Kritikos *

Department of Information and Communication Systems Engineering, School of Engineering, University of the Aegean
* Correspondence: kkritikos@aegean.gr; Tel.: +30-6970047972

Abstract

Large Language Models (LLMs) promise to automatically solve various research and industry tasks, including service discovery. However, the main research question is in which ways service discovery can be supported such that its accuracy is increased. The article's answer to this question is twofold. First, it proposes a novel LLM-based methodology that semantically enriches the OpenAPI description of RESTful services. Second, on top of that methodology, it places novel service discovery algorithms: (a) hybrid ones that exploit both LLM-based embeddings and ontology-based annotations and (b) a configurable and innovative LLM-based service discovery algorithm. All these algorithms are then evaluated in terms of their accuracy against a specific dataset to investigate: (a) whether OpenAPI description enrichment caters for higher accuracy and (b) which of them delivers more accurate results. The main conclusion drawn is that not only is service discovery accuracy elevated, but also the LLM-based algorithm is far better, thus indicating that LLMs can directly tackle the service discovery problem rather than just provide assistance to it.

Keywords: LLM; service discovery; service matchmaking; REST; methodology; embeddings; accuracy; evaluation

1. Introduction

Service-Oriented Architecture (SOA) is a very successful computing paradigm, well-adopted by both academia and industry due to its various advantages, including software re-use, interoperability, standards compliance and vendor neutrality [1]. In this paradigm, the notion of service is a first-class citizen, i.e., a software artefact that can be remotely executed, enabling the implementation of well-defined business functions. In fact, a service can be self-describable, discoverable and composable. The latter means that whole applications and business processes (BPs) can be implemented through a services' composition.

Until now, SOA has been unable to fully deliver its promises as it relies on the well-known SOA triangle that has been broken. In particular, in this triangle, a (global) service registry offers service discovery facilities, enabling service clients to discover and bind to existing services that satisfy their requirements. However, any technology or effort related to service registries has failed. UDDI [2], a standard promoted by IBM and other organisations, was abandoned. ProgrammableWeb was another effort that ceased to exist. In this respect, only organisations of medium or large size occasionally rely on internal service registries to organise and enable the discovery of their own services.

Overall, the reasons for (global) service registry failure are numerous. First, independently of the registry kind, initially only lexical-based approaches [3], e.g., using common Information Retrieval (IR) techniques, were adopted to support service discovery in these registries, leading to low accuracy. Second, while machine learning (ML) techniques came to the rescue [4–7], they had their own issues: insufficient service datasets for training, high computational costs for model training and inference, especially for the more specialised techniques like Deep Learning (DL) ones, and inability to transfer knowledge into new domains.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

Third, while ontology- and logical-based service discovery techniques were pro-posed [8–11], showing high accuracy performance, especially in full IOPE (input-output-preconditions-effects) service matching [8,9], they were never adopted by the market for various reasons: (a) it is a hard task to derive PE information for services and requires major expertise; (b) even the annotation of service input and output (IO) required ontology-based skills that were lacking; (c) automatic service I/O annotation tools did not have the right accuracy level; (d) organisations were reluctant to make the respective investments, as they perceived little value to adopt logic- or ontology-based techniques [12].

As a result, nowadays, organisational service registries still use inaccurate service discovery techniques. This creates frustration to application developers, who need to manually inspect the matched services and select those that satisfy the application/BP requirements. Further, due to the absence of a global service registry, they need to manually find and select external services offered by third-party providers so as to realise the functionality of their applications and BPs. As such, this situation leads to increased effort and cost while delaying the application development, even if the latter relies on modern methodologies (e.g., agile combined with software re-use).

Fortunately, in the current era, new opportunities arise due to the recent developments in Artificial Intelligence (AI) and especially its Generative AI (GAI) field. In particular, Large Language Models (LLMs) are now available, which are powerful GAI models that excel in various transformation tasks. In fact, their abilities show great potential in many areas, including software engineering (SE) and BP management (BPM). For instance, in SE, LLMs can automatically produce UML diagrams from textual descriptions [13,14] or generate code that fulfils a particular functionality.

To this end, our work, conducted in the context of a PhD [15], has aimed to explore the LLM potential in service discovery with the ultimate goal to prepare the foreground to realise a successful global service registry. This exploration targeted the answering of the following research questions:

1. *Q1*. In which ways can LLMs assist or further increase the automation degree in service publication and discovery?
2. *Q2*. Which from these ways leads to better service discovery performance and accuracy?

By observing that increased semantics can benefit service discovery accuracy, our work has developed a research methodology targeting not only service discovery but also service publication via the enrichment of OpenAPI RESTful service specifications with both meaningful textual descriptions and ontology annotations. On top of that methodology, someone can place any service discovery algorithm to increase its accuracy. However, our goal was not just to reuse existing algorithms but to extend them or propose new ones. This is due to the fact that existing algorithms might not be capable of exploiting the degree of attained semantic enrichment. Further, we desired to investigate whether LLMs can only support service discovery or even completely realise it.

As a result, apart from our novel research methodology, we also devised and implemented novel algorithms, which meaningfully extend existing service matching techniques to properly consume the increased semantics in OpenAPI specifications. These include LLM-based embeddings-oriented algorithms that consider both the structure of the OpenAPI specifications and their ontological annotations. Further, we have developed a novel, LLM-based service discovery algorithm, flexible enough to exploit any state-of-the-art LLM and configurable to change the way different factors contribute to service matching and ranking, while incorporating explainability for service matches.

Our methodology has been realised at the technical level by a semantic registry for RESTful service publication and discovery. This registry follows SOA as it is organised via a service orchestration, while it offers a uniform RESTful API, which can be consumed by any kind of application development environment or application/service marketplace.

Further, our methodology, along with the proposed service discovery algorithms, have been experimentally evaluated by considering a specific real service dataset called Evo Master Benchmark

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

(EMB)[1], developed in the context of the Evo Master framework for (RESTful) API testing [16]. The evaluation results lead to the following findings:

- Our methodology leads to enriched OpenAPI specifications with high enrichment accuracy
- This enrichment enables increasing the accuracy of existing service discovery algorithms
- Our new algorithms lead to higher service discovery accuracy as they better exploit the semantics incorporated in the OpenAPI specifications
- Our LLM-based service discovery algorithm reaches high accuracy levels. This highlights that LLMs not just assist service discovery but fully realise it in the best possible way.

Based on these findings, both of our initial research questions have been answered. Our methodology revealed that LLMs can semantically enrich service specifications. Further, we have investigated different ways service discovery can be realised by using LLMs. Finally, all such ways were experimentally evaluated to nominate the winner.

The rest of this article is organised as follows. Section 2 reviews related work. Section 3 analyses our proposed methodology. Section 4 presents the architecture of our semantic service registry implementing this methodology and supplies some implementation details. Section 5 analyses all the implemented service discovery algorithms, including existing and new ones. Section 6 explains how the experimental evaluation was conducted and what its main results were. Finally, Section 7 concludes this article and draws further research directions.

2. Related Work

Initially, functional service matching focused on purely structural, syntactic service specifications and requests, such that the relevant approaches exploited well-known IR techniques to transform these specifications into more semantic forms in order to raise the matching accuracy. For instance, the approach in [3] focused on transforming these specifications and requests into word vectors, then applying traditional similarity measures like the cosine one to compute their similarity and finally filtering each service by enforcing a threshold over its similarity with the request. Such approaches suffered from the following issues, placing a barrier on their service matching accuracy: (a) word vectors like TF-IDF ones cannot address lexico-graphic and semantic/terminological inconsistencies and rely on the completeness and quality of the specifications themselves; (b) the filtering threshold is difficult to establish, might be domain-specific and is another major imprecision source.

The next approaches focused on producing more semantic service/request representation spaces. They can be separated based on the transformation technique used. Embedding-based approaches [17,18] transform word vectors to word embeddings vectors by relying on a specific embeddings model. The resulting embeddings vectors are more precise and semantic, able to address terminological issues. However, such approaches suffer from three main issues: (a) they still require establishing a threshold, which can influence matching accuracy; (b) they rely on the quality and suitability of the embeddings model, which might not cover all possible domains well; (c) the quality and completeness of the original word vectors can influence that of the embeddings.

On the other hand, topic model-based approaches [19] utilise Latent Dirichlet Analysis (LDA) to transform word vectors to topic distributions/maps. However, while the generated topics could relate to higher-level and more semantic notions to address the terminological problems, similar issues apply as in the case of embeddings-based approaches.

Driven by the aforementioned drawbacks, new approaches were proposed utilising more formal or semantic techniques, especially logic and ontology-based ones. These approaches suppose that service and request specifications are formally defined based on the same formalism or that they are at least precisely annotated semantically via ontologies. They can be separated into: I/O-based [10,11] and IOPE-based [8,9].

[1] https://github.com/EMResearch/EMB

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

The former attempt to match the service and request based on their I/O by adopting specific rules, which exploit the semantic relations between the I/O parameters, like subsumption ones. Their service matching accuracy has been proven to be higher compared to the previous ones. However, they suffer from the fact that the I/O parameters alone do not cover the service functionality but only the entities affected by it. Further, they rely on either the proper formal service/request description or the correct I/O-based annotation of these descriptions. However, these tasks are traditionally conducted manually by (domain) experts and can be error-prone, while also organisations lack the respective expertise to conduct them.

On the other hand, full IOPE-based approaches [8,9] seem to attain the highest matching precision. However, they suffer from three main issues: (a) service requesters might not precisely/completely express their needs; (b) specifying PE information requires even deeper knowledge and expertise than formal specification or semantic annotation of I/O; (c) the matching precision is the highest possible, but recall drops as it can be influenced by the previous two issues.

Finally, ML/AI techniques have been recently applied to service matching. This includes unsupervised ML techniques focusing on service clustering [4], supervised ML techniques on service categorisation and tagging [5], as well as deep learning (DL) techniques [6,7]. However, while simple ML techniques might be sufficient for their trained domains, they tend to overfit them such that they are not so accurate in new domains. On the other hand, DL techniques, while better in capturing patterns across different domains, are quite demanding in computational power, especially in terms of training. Further, their accuracy is still imperfect in new domains unless knowledge transfer techniques are put into practice. In any case, any kind of ML/DL technique can be good depending on the quality of its training data. Unfortunately, as has been evidenced in the literature, there is not any rich and well-representative service dataset fulfilling this purpose.

Compared to all previous work lines, our methodology does not rely on any kind of input requiring specialised expertise and manual human work. In fact, it automatically constructs its own input (OpenAPI [20] specifications) based on what is always available: the service source code. Further, it is independent of the respective specification formalism as it can be easily adapted to cover any well-known service description language. In addition, it does not rely on the availability of any dataset or the use of any technique to transform the user request into a more convenient form. In fact, it transforms both service specifications and requests into the same hybrid description space to enable their proper exploitation by any service discovery algorithm kind. Even pure lexical discovery algorithms can exploit this space as they can rely on the lexical representation of the specifications rather than their semantic one.

By considering our novel service discovery algorithms that complement our methodology, we can certainly indicate that the LLM-based one can deliver high accuracy levels, far better than those attained by our other algorithms and existing ones proposed by other researchers. Further, this LLM-based algorithm is simple and straightforward to realise, as it only generates a specific prompt to be exploited by the LLM. In addition, it is LLM-independent as it can be configured to use any LLM kind, including state-of-the-art ones. In addition, its flexibility is also attested by the ability to configure it with different weights over the factors that contribute to service matchmaking. Last but not least, the LLM-based algorithm supplies both lexical and semantic categories for matches, and more importantly it offers the respective rationale for the matching. In this respect, it can cover different developer requirements on matches classification while it caters for explainability in service discovery, something not exhibited by any other algorithm.

3. LLM-Based Service Discovery Methodology

3.1. Assumptions

Prior to devising our methodology, we have relied on the following assumptions, some which can be considered as its core virtues:

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

- We rely on the OpenAPI standard [20] as it is well adopted by the industry. However, in principle, our methodology is independent of the RESTful service description formalism as it can be adapted to support any existing formalism.
- We consider that existing OpenAPI specifications of RESTful services are inappropriate to be used for service discovery purposes for the following reasons: (a) the specifications can drift with respect to the service implementation; (b) in many cases, the specifications are poorly described as they are usually automatically generated by specific tools or libraries that cover the technical service interface. In fact, it has been observed that there is a lack of textual description for service operations and their I/O while also the service operation and parameter names or identifiers do not convey any kind of semantics as they take arbitrary forms. As such, there is a need to consider a single truth source that can enable producing rich OpenAPI specifications. We believe that the service source code can play that role, and LLMs can inspect it to produce a semantically enriched OpenAPI specification.
- While LLMs can produce semantically enriched OpenAPI specifications, such specifications just have more textual descriptions and more meaningful operation and I/O parameter names. However, service discovery practice has unveiled that structural textual description is not enough. Thus, ontological annotations must be incorporated from existing ontologies. Again, LLMs can support this latter task, and our work properly investigated this.
- We focus on the case that service requesters seek specific service operations. We believe that this is more realistic in the context of developing applications or BPs by also considering the very nature of RESTful services, which usually realise management operations over business-related entities.
- Based on the previous assumption, we regard that service requesters do not possess special skills, so they supply a short textual description of the desired service operation. While this description can be ambiguous, we consider it highly realistic (under this highly probable real-world scenario), and our work will attempt to mitigate the incurred ambiguity in various ways.
- Due to our focus on service operation matching, we consider that service matching is a side-effect of the former. In this respect, we match service operations against the service requests, and then we also supply the services that offer these matching operations. This is also related to the very nature of RESTful services, which do not map to specific overall I/O parameters, thus satisfying an overall functional capability. Thus, they are seen as merely collections of functional capabilities implemented by their operations. However, this does not signify that they are neglected in service operation matching. On the contrary, they are accounted for in various algorithms that we propose, as we believe that they can enhance discovery accuracy.
- As can be well understood, our focus is on functional service matchmaking. So, non-functional service matchmaking will be covered in our future work.

3.2. Methodology Core

Our research methodology towards accurate service discovery attempts to achieve the same hybrid representation space for both service operations and requests. Through this achievement, it can then enable to more properly conduct service discovery by adopting any kind of discovery algorithm, either lexical, semantic or hybrid.

In the following, we present this methodology in terms of its main phases. Then, we depict the hybrid representation space for service operations and requests it can achieve. Next, we analyse methods that implement all methodology phases except the last one. As the last one maps to service discovery, its realisation via particular service discovery algorithms is covered in Section 5.

Our methodology is depicted in Fig. 1. It comprises five sequentially executed phases. The first three phases cover the service publication in the service management cycle, as they are conducted during the publishing of RESTful services. On the other hand, the last two phases cover service discovery as they relate to the request's pre-processing and its matching with the already published services. All of these phases, except the third one, map to methods supported by LLMs. However,

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

please note that our methodology is LLM-agnostic; in fact, it could be supported by methods exploiting alternative AI techniques.

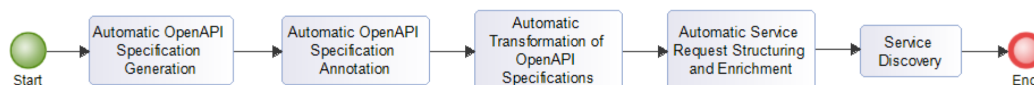

Figure 1. The proposed research methodology modelled as a business process.

Automatic OpenAPI Specification Generation. The first methodology phase covers the automatic OpenAPI specification generation for RESTful services based on their source code. In particular, by considering the service's source code as the single truth source, it attempts to automatically produce a semantically rich OpenAPI specification for that service with the following features: (a) proper names/identifiers for service/operations/parameters; (b) textual descriptions of the service, its operations and the operations' I/O parameters; (c) inclusion of types and examples of I/O parameter values; (d) proper hierarchical coverage of parameter types/schemata, including error ones.

Automatic OpenAPI Specification Annotation. The second methodology phase attempts to annotate the OpenAPI specifications by using ontologies, either domain-specific or general ones or both. This annotation covers mainly the service operations' I/O parameters and the actions realised by these operations. Please note that, contrary to semantic I/O based discovery approaches, the coverage of the service operation action prospects to increase service discovery precision as it "simulates" PE-based information. In particular, while I/O-based approaches rely on the semantic service (operation) signature, the consideration of service (operation) action can enable distinguishing between services (operations) with the same signature.

For instance, two service operations, which process the same input, like a patient's Medical Record Number (MRN) and produce a confirmation, are considered equivalent by semantic I/O-based approaches. However, one of the operations might check in the patient in the hospital while the other might check out this patient. Thus, considering the service action (CHECK-IN vs CHECK-OUT) could enable distinguishing between them and return only one in the context of a service request, which, e.g., requires a patient's check-in.

Automatic Transformation of OpenAPI Specifications. The third methodology phase focuses on transforming each OpenAPI service specification to a service description conforming to the adopted hybrid service/request representation model (see details in Section 3.3). In this way, eventually both service requests and operations will be uniformly represented, thus greatly facilitating service discovery. We expect that this phase is easy to implement in a deterministic way and thus does not require using LLMs or other AI techniques. This was actually the case for our service registry implementation.

Automatic Service Request Structuring and Enrichment. By considering that the service request corresponds to a short textual description of a desired service operation, the goal of the fourth methodology phase is first to structure the request in terms of its action, input and output sections and then to annotate all these sections by following the same annotation approach as for the OpenAPI specifications. As a result, the resulting service request's representation will be equivalent to that for service operations based on the adopted hybrid service representation model.

The service request structuring and annotation plus the annotation of service operations must be accurately conducted as any imprecision issue could impact service discovery accuracy. Thus, an accuracy greater than 90% (i.e., a very high accuracy level) could be considered as satisfactory.

Service Discovery. The last methodology phase concentrates on the functional matching between the service request and operations, provided that both conform to the same representation model, something guaranteed by the previous phases. Such a matching should end up in delivering service operation matches, which are categorised by at least one classification dimension (e.g., structural or semantic) and ordered based on their matching degree. Each service operation match should also point to its encompassing service. Further, a justification for including the service operation match should be

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

optionally supplied. Finally, the matching should be as precise as possible, as low accuracy values can frustrate the service requester. All these requirements signify the need to introduce a representation model concerning the functional service matching output, which is analysed in Section 3.3.

Overall, we believe that our methodology is valid, sensible and contiguous plus LLM / method / technique agnostic. Further, we strongly support that it can guarantee higher service discovery accuracy, if properly implemented by suitable methods realising its core phases. Finally, this methodology eventually produces very useful information in form of service discovery output, which can benefit the service requester as it can unveil the explainability of the results as well as supply their classification and ordering, catering to varied representation requirements posed by different requesters.

3.3. Service Representation Models

This subsection aims to analyse our service representation model that constitutes the backbone of our research methodology for two main reasons. First, it covers the representation of services, their operations and service requests. Second, it covers the representation of service matching results.

Fig. 2 depicts this model in a class diagram form. As can be seen, there are two central classes, covering two main entities, respectively, i.e., a service-related entity (*Artifact* class) and the operation match entity (*MatchResult* class). The first class covers any entity related to a service, including the service itself, its operations, the operations' I/O parameters and the service requests. This class incorporates important information for service discovery purposes, including: the entity's name, its textual description, its TFIDF vector and its embeddings-based vector, where the latter vectors enable using TFIDF and embeddings-based algorithms in service discovery. Further, this class is related to one or more (ontological) annotations. Each annotation is characterised by the name of the ontology element annotating the current artifact, the name of the ontology that includes that element and the element's complete URI. Please note that the use of multiple annotations would enable covering ontology heterogeneity issues and/or multiple information aspects relevant to the annotated artifact.

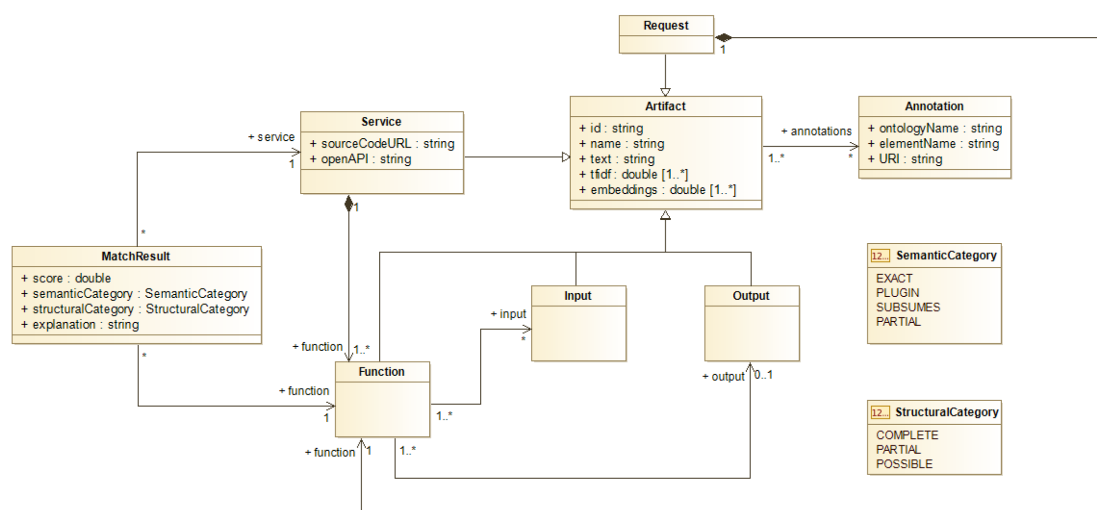

Figure 2. Class diagram of the service representation model.

A *Service* is an artifact kind that represents a RESTful service. This service has a specific URL, mapping to the internet address of its source code. Further, it has as a field its whole OpenAPI specification. A service comprises one or more *Function*s, which constitute another artifact kind.

A function represents a service operation. Such an operation can have multiple *Input*s and at most one *Output*. Inputs and outputs represent different artifact kinds.

A *Request* is an artifact kind that represents a service request, which relates to a specific function. This means that both service requests and operations are eventually represented by the same class/concept, which guarantees their uniform representation.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

Moving to the discovery output side, we observe that each service operation match is represented by a *MatchResult*, characterised by various fields, including the overall match score, its lexical and semantic category, and its textual justification. Further, each match is associated with a specific function and service. In this respect, we well cover all required information aspects relevant for service operation matches as designated by our methodology's last phase. As a result, a service discovery algorithm needs just to return an ordered list of *MatchResult*s, ranked based on their categories and score.

We end the service representation model analysis by explicating the semantics of the lexical and semantic categories. Three main categories are involved in the lexico-graphic dimension:

- *COMPLETE*: maps to service operations that fully match all aspects of the service request.
- *PARTIAL*: includes service operations that might require some customisation to fully match the service request. This signifies that there can be some intent or parameter gaps.
- *POSSIBLE*: maps to service operations that topically match the service request in an incomplete manner. Thus, they are insufficient to completely satisfy the user request.

On the other hand, the next four categories exist in the semantic dimension, formally defined in [11].

- *PERFECT*: the service operation is semantically equivalent to the request.
- *PLUGIN*: the service operation produces a more specific output (a child concept) than the one demanded by the request.
- *SUBSUMES*: it is similar to *PLUGIN* but includes service operations where their output is a sub-concept of the request output, while each request input is the same or a sub-concept of a service operation input.
- *PARTIAL*: the service operation partially matches the request. Thus, it might need to be combined with other operations to completely fulfil it.

3.4. Automatic LLM-Based OpenAPI Specification Generation

To realise our methodology's first phase, we have devised an LLM-based method [21] that incrementally processes the service's source code and builds its OpenAPI specification. This method includes two main activities: (a) *service source code filtering* and (b) *OpenAPI specification incremental construction*.

3.4.1. Service Source Code Filtering

The first method activity deterministically attempts to filter out irrelevant files from the service's source code to guarantee that the second activity will rely only on relevant files that correlate with the OpenAPI specification's content. This filtering relies on a rule-based approach primarily applied to Java- or Kotlin-based RESTful services. Such a deterministic approach was selected as it can be efficient in service code filtering and avoids potential hallucinations (usually delivered by LLMs), which can reduce the filtering accuracy and negatively impact the second activity. We acknowledge that our approach works only for specific programming languages, but we intend to extend it to cover more, especially those widely used for RESTful service development, like Python, Go and JavaScript.

The rule-based approach was implemented in the form of an algorithm, depicted in the Listing 1, which initially processes the service source code structure to maintain only the pure source code directories, which have the relative path "src/main/java" or "src/main/kotlin".

Next, the algorithm attempts to pick up the most relevant files recursively from the maintained source code directories. Such relevant files are selected by considering the architecture of RESTful services, which are developed by well-known frameworks, like Spring Boot and Jersey. This means that we consider files that play the role of *Controllers/RestControllers* (Spring Boot) or *APIs/Paths* (Jersey). Controllers expose a RESTful service's interface, while APIs/Paths both expose such an interface and implement it.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

Listing 1. File filtering algorithm's Java-based pseudo-code

```
Set<File> fileFilterer(File rootDir){
  Set<File> result = new HashSet<File>();
  Set<File> dirs = filterDirs(rootDir);
  for (File dir: dirs){
    for (File f: dir.filesRecursive()){
      if (f.matches(codeExtIndicators && (serviceIndicators ||
      domainIndicators || errorIndicators))
      && !f.matches(excludeIndicators)) result.add(f);
    }
  }
  return result;
}
```

In the context of Spring Boot, we also consider additional files, such as *Services* and *Components*. Services implement the interface exposed by Controllers, while the same could be said for Components in some cases. While it could be argued that the service interface is enough for OpenAPI specification generation purposes, it is insufficient as it covers mainly the service signature and not its operation semantics. While the latter can only be understood by an LLM by inspecting *Services/Components*, such that it can then incorporate in the OpenAPI specification the textual description of the service operations.

Further, in both cases (Spring Boot & Jersey), we attempt to cover domain models and DTOs (Data Transfer Objects), with a preference for DTOs, which do not expose internal details about the involved domain classes (when present). Both domain models and DTOs represent informational entities exchanged between service requesters and services during service operation calls. As such, they directly correspond to the schema part of an OpenAPI specification and are essential for its generation.

Finally, similarly to domain models and DTOs, we maintain error or exception classes as these are usually returned to service requesters when a specific error is made, either at the client/requester or the service side.

The file filtering logic relies on checking per each file whether it has the right extension (see *codeExtIndicators*), it maps to a controller/path/service (see *serviceIndicators*), domain/dto (see *domainIndicators*) or error/exception class (see *errorIndicators*), and does not include improper content (such that it is considered irrelevant and must be discarded) (see *excludeIndicators*). In case this complex condition holds, the file is added to the set to be returned. Otherwise, it is filtered.

The *codeExtIndicators* check if a file has a ".java" or ".kt" extension, mapping to a Java or Kotlin (implementation) class. On the other hand, the *serviceIndicators* focus on file content and attempt to investigate if a specific annotation exists, mapping to a valid service implementation file. Valid annotations include "@RestController", "@Controller", "@Api", "@Path", "@Service" or "@Component".

The *domainIndicators* enable to select domain models and DTO classes based on the following alternative cases:

- the file name should end with "DTO" (case insensitive)
- should include in its content the "'data class" declaration – this is Kotlin-specific and utilised to denote immutable domain/data objects
- the file's content must include one from the next annotations: "@Entity", "@Document", "@ApiModel", "@Schema", or "@JsonInclude". These annotations relate to using specific data-oriented frameworks or technologies like the Object-Relational Mapping (ORM) (e.g., JPA). In particular, "@Entity" is a JPA annotation for domain entities, "@Document" represents MongoDB domain documents, "@Schema" is used for documentation purposes of domain models, and "@JsonInclude" is used to enforce Jackson serialisation control for domain/DTO objects.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

errorIndicators attempt to explore whether a specific file corresponds to an error or exception class based on the following alternative cases:

- the file is included in a directory named as "error" or "exception"
- the file's name ends with "Error" or "Exception"
- if the file's class implements or extends "Exception" or "Throwable", thus checking in this case the file's content

Finally, *excludeIndicators* play the role of a safety valve, as there is always a risk that a wrong file is not filtered due to the presence of an annotation in the file's content or its misleading name or placement. For instance, in some cases, files annotated with "@Component" do not implement a service interface but rather play a supportive role to such an implementation. In this respect, the following alternative cases are covered:

- the file's content incorporates one from the following annotations: "@SpringBootApplication", "@Configuration", "@Bean", "Provider" and "@Singleton". All these annotations are utilised in the configuration and dependency injection layer of an application, so they map to irrelevant kinds of classes in the context of OpenAPI specification generation.
- when the file's class extends module, converter, validation or (de)serializer classes (e.g., StdDeserializer). Such classes might be named with "DTO" substring or have the "@Component" annotation while they are irrelevant for OpenAPI specification generation purposes.

As shown in [21], our file filtering algorithm achieves a filtering accuracy above 90% in average, thus reaching a quite high and satisfactory accuracy level. The main sources of imprecision were mainly converter, validator and mapper classes that bypassed the *excludeIndicators* filter, as well as model classes that were not needed as they were wrapped by respective DTOs. Concerning the latter, we should note that essentially model classes should be neglected as DTOs must be used. However, it has been observed as a programmer practice that in some services, either only model classes were implemented or a mixture of model and DTO classes that were both used in service operation request/response bodies. In any case, the incurred imprecision is tolerable as irrelevant files do not influence the OpenAPI specification generation. However, they lead to unnecessary cost and extra generation time as they are processed by the LLM conducting the specification generation. Thus, we plan to enhance our filtering logic to overcome the identified issues and reach an ideal file filtering precision.

Concerning file filtering recall, the main causes of impression were: (a) the non-consideration of enumerations that signify the values that a specific service operation I/O parameter can take and (b) bad programmer practice in very few cases where either DTO classes were not named as such and did not incorporate any relevant annotation or the service interface was implemented with pure Java classes that did not have also any relevant annotation. In the context of the first cause, we believe that this does not impact service discovery, but mainly the richness of the OpenAPI specification, which might be handy in cases of RESTful service testing or usage. The second cause could impact though service discovery, as the LLM might have difficulties in understanding the exact service operations' functionality just by their names and signature and might not properly/completely represent the data type for a missed DTO class in the OpenAPI specification.

Based on the above rationale, we believe that the recall issues are subtle and might not be experienced in modern RESTful service implementations, especially if we consider that nowadays these implementations are driven or even generated by LLMs. Nevertheless, we will attempt to enhance our file filtering algorithm to include enumerations where appropriate, as well as to search and include not annotated service implementation classes and DTOs.

3.4.2. OpenAPI Specification Incremental Construction

This is the second method's activity, which is LLM-based and leads to the construction of a service's OpenAPI specification based on its filtered source code. This activity is algorithmically implemented via a loop that attempts to incrementally construct the service's OpenAPI specification

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

by processing each individual file maintained from its source code. In the first loop iteration, the LLM receives the first file and generates a first specification version. In each remaining loop iteration, the LLM is fed with the current specification version and the next file to process and produces the next specification version. Eventually, when the loop finishes, the LLM will have constructed the complete OpenAPI specification for the current service at hand.

In every loop iteration, the LLM is fed with a specific, carefully crafted prompt, with a stable part including the instructions for the OpenAPI specification generation/updating. The variable part depends on whether we are at the first loop iteration or not. In the first iteration, the LLM is asked to create the first specification version. In the next iterations, the LLM is asked to update or enhance the current specification version based on the given file's content, whenever the latter is considered relevant by the LLM (please remember that our file filtering activity has imperfect precision). Please note that the specification might have to be updated to become more precise or might be enhanced to include additional parts. That is why we deliberately indicate to the LLM to consider both cases.

The stable prompt part was constructed by following the *Persona* (i.e., role) and *Template* (i.e., output) prompt patterns [22]. Initially, it applies the Persona pattern by requesting from the LLM to act as an expert in modelling OpenAPI specifications who follows best practices for OpenAPI 3.0.0 compliance. At its final section, it requests the LLM, by applying the Template pattern, to generate or update the OpenAPI specification in strict JSON format and not include any additional text or explanations in its output. The remaining sections include a set of instructions that explain what has to be included in each part of the OpenAPI specification, and which are the validation constraints.

In a nutshell, these instructions demand from the LLM to:

- include in the specification's *Info* section detailed text for the "title" and "description" fields, along with the value of 1.0.0 for the "version" field. The main goal is to introduce an overview of the service's functionality that can facilitate its operations' discovery.
- incorporate in the specification's *Paths* section all the service's operations as derived from its source code. In each operation, the right HTTP method must be used, the "operationId" field's value should be equal to the name of the method that implements this operation, and a detailed textual description of the operation's functionality must be incorporated in the "summary" field. Further, it is instructed that all I/O parameters should have clear names, types and locations along with a detailed analysis of their semantics in the "description" field. In case of complex types, they need to properly reference the correct schemas at the specification's *Components* section. In addition, one example must be given for complex types in I/O parameters. Finally, no duplicate keys should be supplied in the operation path. The main goal of the above rules is twofold: (a) guarantee that the service operation signature fully matches that of the respective class method in the service's source code; (b) detail the semantics of each service operation and its I/O parameters based on the implemented behaviour in the respective class method. As such, a more precise discovery of the service operations can be facilitated.
- include in the *Components* section the schemas of all composite types mapping to the service operations' I/O parameters. The schemas must reflect the structure of the information exchanged as reflected in the service's source code (e.g., in terms of the involved domain models or DTOs). Such structured schema definitions can definitely assist in the semantic enhancement of I/O parameters in the OpenAPI specifications, as they can match ontology concepts.
- ensure that the generated specification conforms to OpenAPI standard version 3.0.0, without causing any validation error. In addition, POST operations should not include plain input parameters but a *requestBody* part. Finally, no duplicate information should be incorporated in the specification. All these rules enforce the syntactic and structural validity of the generated OpenAPI specification, guaranteeing its proper processing (e.g., by other methods that implement the proposed research methodology).

The aforementioned stable prompt is included in the package produced for reproducibility reasons for this article. This package is available at a specific link given at the end of this article.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

This method activity was evaluated in [21], focusing on the quality and precision of the generated OpenAPI specifications as well as on their potential impact on service discovery. Concerning the first aspect, the evaluation validated that the OpenAPI specifications have high quality and richness, while they precisely reflect the service implementation. In particular, it was derived that: (a) the service and operation documentation was complete in all of the generated specifications, signifying that the LLM properly understood both the service code and the supplied instructions; (b) no OpenAPI validation issue incurred; (c) the OpenAPI specification precisely covered the signature of all the implemented service operations with proper path structure; (d) in the context of complex I/O parameter types, the right schema element was generated and associated as the type of the respective parameter (in most of the cases – please remember the recall issue about DTO classes); (e) the service documentation was written in English even though the source code was based on a different spoken language (in terms of comments and code variable/operation/class/field names) – this actually indicates that LLMs can understand different spoken languages and properly produce the service documentation, bypassing language objects and making our service registry language-agnostic/independent.

As far as the second aspect is concerned, the evaluation relied on two different RESTful OpenAPI specification sets: (a) the original set from EMB benchmark, which included multiple service specifications with meaningless operation/parameter names and incomplete or missing service/operation documentations; (b) the generated OpenAPI specifications from the source code of the RESTful services in EMB. Each of these two sets was registered in one instance of our TFIDF service matcher (see Section 5.3 and then both matcher instances were assessed based on a query dataset that we constructed, which contained standard keywords partially included in the matching OpenAPI specifications and had as a source truth a single service operation per query. By considering the topmost result produced by both matcher instances, we discerned that the instance configured with the generated specification set was able to accurately produce far more correct service operation matches (92%) than the other instance (42%). Thus, service discovery accuracy was more than doubled when using the generated OpenAPI specifications, which highlights their (semantic) richness and the potential of our method to boost service discovery accuracy.

3.5. Automatic LLM-Based OpenAPI Specification Annotation

This method's goal is to exploit the LLMs' transformation power to annotate the RESTful services' OpenAPI specifications based on both the action and I/O parts of their operations. As LLMs cannot accurately handle complex tasks, we decided to split the overall method into two activities: (a) one devoted to I/O annotation and (b) one devoted to action annotation, which are analysed in the following subsections. The overall process realised in our method, mapping to the parallel execution of the method activities, is depicted in Fig 3.

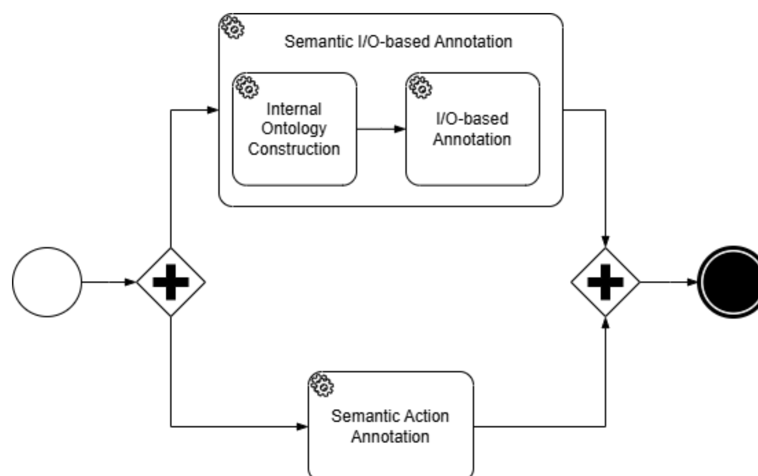

Figure 3. The LLM-based OpenAPI specification annotation method process.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

3.5.1. I/O Annotation of OpenAPI Specifications

Originally, we targeted using existing ontologies, both domain and generic, to realise this method activity. However, our evaluation [23] showed that the annotation accuracy mostly reached 60% in the context of Mistral's Large LLM. This accuracy level is obviously insufficient to increase service discovery accuracy. This inaccuracy was due to the fact that most of the time, the LLMs either provided a wrong annotation or created hallucinations under the pressure to suggest something relevant.

To this end, we settled for a more sophisticated approach with the following rationale: (a) existing ontologies might not cover their domain well; (b) ontologies might not exist for particular domains; (c) even if such ontologies exist, LLMs might not be able to discover them or be trained based on them. As such, we focused on producing our own internal ontology by relying on the richness of the schema definitions in the registered RESTful services' OpenAPI specifications. Our main logic is that even if the ontology might be incomplete, it: (a) will be able to cover all the domains related to the registered services – thus, this also addresses the above issue of non-existing or neglected ontologies; (b) if there is a request including a notion not present in our ontology, this can mean that potentially this request does not match any of the registered services – provided that our ontology is rich to cover notions hierarchies and not just a flat map of them. To further safeguard the latter, we also utilised a generic ontology, Schema.org, as a bridge to cover missing notions when these are not covered by our internal ontology. Such a bridge was realised by incorporating equivalence axioms in our ontology between concepts from our ontology and Schema.org. Thus, by relying on this bridge, we actually created a multi-ontology that is much richer and has a suitable hierarchy depth to cover any concept that might be related to a service request and might be missing in our internal ontology. Further, incorporating Schema.org enables to additionally cover multiple general and domain classes.

Consider, for example, the case of a *NewsArticle*. This is a kind of an *Article*, which in turn is a kind of a *CreativeWork*. This concept hierarchy is reflected in Schema.org. By bridging the concept of *News* in our internal ontology with the *NewsArticle* one, we are able to provide matching results for requests that demand the retrieval of creative works. In fact, the matching results could include even further operations, as they could cover the retrieval of *Book*s and *SoftwareSourceCode*. This is not an imaginative scenario but a real one that corresponds to the EMB dataset and the internal ontology that we constructed from this dataset.

Based on the above analysis, we have devised and implemented a sophisticated approach for I/O-based OpenAPI specification annotation [23] (i.e., the first method activity) that comprises two main tasks, where the first is devoted to constructing the internal ontology from the specifications and the second to annotating the specifications in terms of their I/O parameters by using this ontology and Schema.org as the backup. This approach is incarnated in the internal logic of the *Semantic I/O-based Annotation activity* as depicted in our method's process in Fig 3. As can be seen, the aforementioned activity is actually a sub-process that maps to the sequential execution of the approach's two tasks, which are analysed next.

First Task: Internal Ontology Creation

The first task of the semantic I/O-based Open API specification activity was incarnated in the form of an algorithm, incorporating the following three main operations:

- *I/O Parameter Extraction*: This operation, named as `parseOpenAPISpecification()`, extracts the main I/O parameters and their types from a service's OpenAPI specification.
- *Ontology Chunk Construction*: This operation, named as `computeIntegrateOntologyChunk()`, produces an ontology chunk out of an OpenAPI specification's extracted I/O parameters.
- *Ontology Chunks/Parts Consolidation*: This operation, named as `consolidateOntologyChunks()`, takes as input a set of ontology chunks or parts (e.g., N out of M, where M is the total number of OpenAPI specifications) and creates one overall ontology part out of them.

In the sequel, we first analyse this algorithm and then each of its main operations in separate paragraphs.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

Internal Ontology Construction Algorithm

Listing 2 depicts the pseudo-code of our internal ontology construction algorithm. The algorithm, for each OpenAPI specification given as input, first extracts the ontology parameters contained in it and creates an ontology chunk out of them. For each *incrementNumber* chunks generated, it then consolidates them into an ontology part. Finally, once all OpenAPI specifications are processed, it consolidates all merged ontology parts into a final ontology model, forming our internal ontology.

Listing 2. The internal ontology construction algorithm

```
constructOntology{Set<File> files, int incrementNumber}
    counter = 0;
    finalModel = ModelFactory.createDefaultModel();
    collectionModel = ModelFactory.createDefaultModel();
    for (File f: files){
      Collection<OntologyParameter> params= parseOpenAPISpecification(f);
      if (params != null){
        entered = computeIntegrateOntologyChunk(params, collectionModel);
        if (entered){
          if (++counter == incrementNumber){
            collectionModel = consolidateOntologyChunks(collectionModel,true);
            if (collectionModel != null) {
                    finalModel = finalModel.union(collectionModel);
            }
            counter = 0;
            collectionModel = ModelFactory.createDefaultModel();
          }
        }
      }
    }
    if (counter != 0){
      collectionModel = consolidateOntologyChunks(collectionModel,true);
      finalModel = finalModel.union(collectionModel);
    }
  finalModel = consolidateOntologyChunks(finalModel,true);
  return finalModel;
}
```

Suppose that there are M OpenAPI specifications and N is the number of chunks to merge each time (see *incrementNumber* parameter in the above pseudo-code). Then, the algorithm will produce and then consolidate into the final internal ontology $\left\lceil \frac{M}{N} \right\rceil$ ontology parts.

The ontology parts could have been consolidated into chunks (i.e., bigger parts) themselves before producing the final ontology. However, as the number of OpenAPI specifications is small in EMB, with the proper configuration of N, further chunking was not necessary. In real settings, though, the recursive chunking will be necessary as there can be a huge number of ontology parts, and all of these cannot fit the LLM context window size. As such, we plan to slightly extend our algorithm to accommodate this.

First Algorithm Operation: I/O Parameter Extraction

Before implementing the algorithm's first operation, we generated a specific structure called *OntologyParameter* to incorporate the necessary information from each I/O parameter that could facilitate its incarnation into an ontology chunk. This structure comprises the following fields:

- *name*: the I/O parameter's name
- *description*: its textual description
- *isComposite*: whether it maps to a composite type
- *simpleType*: its simple type (where applicable)

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

- *fatherParameter*: the parameter in which it is contained. This covers the case that the parameter is a field that is included in a composite type represented by another *OntologyParameter*
- *fatherClasses*: these are super-classes if the current parameter is composite and thus can be considered as a specific class. Parameter sub-classing relied on the use of the *allOf* element in the definition of the sub-class type

This parameter processing was conducted serially from the first towards the last I/O parameter in the OpenAPI specification by also consulting their types, when composite, from the specification's *Schema* part. The main processing logic was as follows:

- a simple-type parameter named equivalently to a field in a composite type is considered as a datatype property included in the composite type's class. So, for this parameter, the *fatherParameters* must be completed.
- a parameter that maps to a composite type is structurally represented by this type. Thus, we create an *OntologyParameter* for it, constructed from its composite type. The latter parameter structure will have the *isComposite* field as true and might have super-classes if the *allOf* element is included in its specification. Further, it might have a *fatherParameter*, if there is another composite type that has a field with the current type as its data type.

Second Algorithm Operation: Ontology Chunk Construction

Based on the first operation's main outcome, a list of *OntologyParameter*s is produced, which can be used to construct the respective ontology chunk by the second algorithm operation. The chunk construction covers the following cases:

- an ontology parameter with a simple type that does not have a father parameter is mapped to a global datatype property not encompassed in any concept. The `id` (i.e., the fragment part of this entity's URI) of the datatype property and its `rdfs:label` map to the parameter name, while the property's type (`rdfs:range`) is the parameter's XSD type (e.g., `xsd:string`). Further, the datatype has its `rdfs:comment` equal to the parameter's (textual) description. For example, the *identifier* parameter will have as `id` & `rdfs:label` "identifier", the `xsd:string` as `rdfs:range` and as `rdfs:comment` "the id of the entity".
- a parameter with a simple type that has a father parameter will become a datatype property of the father parameter's class. We construct the same RDFS (triple) statements for this parameter as in the previous case, but we add an extra `rdfs:range` statement, pointing to the father parameter's class id. For instance, "streetName" datatype property will have `:Address` as its `rdfs:domain` value (i.e. pointing to a concept representing an address).
- a parameter with a complex type will become an ontology concept (i.e., will map to rdf:type owl:Class statement). This concept will have as id and rdfs:label the parameter name and rdfs:comment the parameter's description. Further, in case it has super-classes, multiple rdfs:subClassOf statements will be added, each pointing to a different super-class. For example, the *EuropeanCountry* concept will have rdfd:subClassOf equal to `:Country`. Moreover, in case it has a father parameter, we need to create an object property (named as "has" plus the parameter name) connecting this parameter to its father one. For instance, if *Country* has *Address* as a father parameter, we will create an object property named as "hasCountry" with `rdfs:domain` equal to `:Address` and `rdfs:range` equal to `:Country`.

Third Algorithm Operation: Ontology Chunk / Part Consolidation

As evident from the internal ontology construction algorithm's presentation, both ontology chunk and ontology part consolidation rely on the same operation called `consolidateOntologyChunks()`, which we will now analyse. Before delving into this analysis, we must explicate the main reasons for this consolidation. First, chunks might contain similar, equivalent or overlapping elements that have to be merged into one in most of the cases. This is especially true when OpenAPI specifications map to similar domains or incorporate similar/equivalent generic notions. Second, the consolidation

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

is necessary to keep the ontology (part) size affordable, such that it can fit the LLM context window size. If this is not guaranteed, we might not be able to produce a complete final ontology. Third, the consolidation might enable making the ontology (part) richer, by: (a) introducing sub-class relations when appropriate, connecting concepts belonging to different chunks/parts or a concept belonging to a chunk with a novel concept created by the LLM itself – this enables the ontology hierarchy to become deeper; (b) bridging ontology part concepts with Schema.org ones.

The consolidation operation follows an LLM-based two-step strategy to separate the different functional tasks to be realised by an LLM: (a) first, core consolidation occurs to merge concepts where appropriate, and (b) second, ontology enrichment is performed. To this end, a specific, carefully crafted prompt was produced in both functional tasks/steps that was fed to the LLM so as to produce the respective consolidation result. Each of these prompts followed the Persona and Template prompt patterns. Both prompts are available in the article package linked at the end of this article.

In context of the core consolidation step, the LLM is instructed to act as an ontology engineer and consolidate the given ontology chunks/parts. Then, the prompt enumerates all relevant issues to be handled, such as duplicate/equivalent classes/properties. Finally, the prompt includes the next sections:

- *Equivalent Class/Property Merging*: Utilise one canonical URI per class/property and retain only one from classes/properties when they have the same or similar meaning and semantics. During class merging, retain the name that better matches the respective domain. During properties merging: (a) retain the most representative label and comment; (b) select the most general type for them (rdfs:range); (c) apply owl:unionOf directly in the rdfs:domain when the properties belong to different, non-merged classes.
- *Structural Integrity Maintenance*: The LLM is instructed not to assign the same resource to both 'owl:Class' and 'owl:ObjectProperty' / 'owl:DatatypeProperty'. Further, duplicate rdfs:domain and rdfs:range triples for a property must be removed. Finally, references to undefined resources must be eliminated.
- *Namespace Consistency Preservation*: The LLM must utilise the same namespace in all unified concepts and properties. Further, unused or duplicate prefixes must be removed or consolidated.
- *Output Format*: The LLM is instructed to generate only the consolidated ontology's specification in valid Turtle syntax with no explanations / commentary.

In the context of the ontology enrichment step, the LLM was instructed to act as a senior ontology engineer, with the main task to enrich the given (already consolidated) ontology part by enlarging the ontology hierarchy depth and bridging with Schema.org via adding owl:equivalentClass or owl:equivalentProperty statements to generate external ontology mappings. This is covered by the following prompt's instruction sections:

- *Class Hierarchy Enrichment*: generate a common super-class when multiple classes share structure or semantics and move in that class their common data/object properties. Further, if an object property signifies the containment of a specific class with a structure that is fully inherited and extended, model the contained class as a subclass of the containing using rdfs:subClassOf and remove the object property.
- *External Ontology Mappings Addition*: add an own:equivalentClass or owl:equivalentProperty statement when an element in Schema.org is semantically equivalent to a class or property in the consolidated ontology, respectively. Further, do not hallucinate.
- *Structural Integrity Finalisation*: guarantee that all rdfs:domain and rdfs:range statements are valid. Further, delete any residual duplicate or conflicting triples. In addition, utilise owl:equivalentClass or owl:equivalentProperty statements only when needed and only between two resources. Finally, break circular rdfs:subClassOf references by retaining only the most semantically valid rdfs:subClassOf links.
- *Namespace Consistency Preservation & Output Format*: these two sections are actually equivalent to those in the first step's prompt.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

The evaluation of the first method activity in [23] revealed that by using a state-of-the-art LLM (Claude Sonnet 4), the constructed internal ontology exhibits a good quality and richness level. Concerning the ontology richness, the inheritance depth was 5, the property-to-class ratio was 2.88, while the average breadth reached 0.72. Further, the ontology had correct equivalence axioms towards Schema.org and did not include any hallucination. The sole issue evident in the ontologies produced by all the involved LLMs was that there were very few invalid statements. To this end, we plan to extend our ontology construction activity to handle this issue by exploring both deterministic and LLM-based correction ways and selecting the one leading to better performance.

Second Method Activity: I/O Annotation

Once our internal ontology is constructed, it can be used to annotate the I/O parameters of all registered services' OpenAPI specifications by exploiting an LLM's facilities. However, it could be the case that due to the notions refactoring and consolidation, the LLM might have slight difficulties in identifying the right ontology element. Thus, to avoid LLM hallucination, we utilise Schema.org as a backup to annotate those parameters that do not seem to match our internal ontology. This is the main annotation logic that we follow for both RESTful services and their requests. This uniform (annotation) handling along with the ontology bridging and depth richness, enables overcoming various kinds of imperfect annotation cases. In this way, the matching of service requests with service operations can become possible even in such cases, thus enabling us to increase service discovery accuracy.

The ontology-based I/O parameter annotation of OpenAPI specifications relied on constructing a carefully crafted prompt, which conforms to the Persona and Template prompt patterns. This prompt first advises the LLM to play the role of an expert in ontology mapping and explains the mapping task to be performed. Then, it incorporates various instruction-oriented sections to support this mapping. Finally, it ends with the supply of the two main inputs, i.e., the internal ontology and the OpenAPI specification to annotate.

This prompt has also been incorporated into the article package linked at the end of the article. In the following, we provide a summary of its main instruction-oriented sections:

- *Ontology Element Selection*: map each I/O parameter to an element from the internal ontology. If no match is found, match the parameter with an element from Schema.org. When no match is found in any of the ontologies, set "UNKNOWN" as the mapped element.
- *Output Format*: return only a JSON array with no commentary or explanation. Each array member must be unique and include an identified mapping from the current service's I/O parameters to ontology elements. This mapping should include: (a) the name of the matched I/O parameter – for input parameters, we retain their name while for output parameters, the name of their data type/schema; (b) the full URI of the ontology element mapped to the parameter; (c) the name of the internal ontology or Schema.org as the source ontology including the mapped element.
- *Selection Criteria*: conduct the mapping by considering the semantics, data type and context (of the I/O parameters) within the OpenAPI specification. Output parameters must be mapped to ontology concepts while input parameters to data/object properties or ontology concepts, depending on the current usage/context. Finally, I/O parameters should be mapped only to valid/existing elements from the two ontologies considered.
- *Constraints*: do not annotate individual fields of schemas but only complete schemas. Do not utilise concepts/properties belonging to other ontologies than the considered. Further, do not hallucinate by using non-existing elements. Finally, do not provide duplicate array members.
- *Output Constraints*: produce only a syntactically and semantically valid JSON array. Further, ensure that all URIs in the array are valid and resolvable.

Our service parameter mapping/annotation approach is innovative as it combines the transformation power of LLMs with a clever two-level fallback mechanism to close semantic gaps. Indeed, as shown in [23], with the right LLM selection (Deep Seek V3), the annotation accuracy became perfect

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

with full coverage of all I/O parameters and no hallucinations. As such, a major prerequisite for having the ability to elevate the service discovery accuracy has actually been reached.

3.5.2. Action Annotation of OpenAPI Specifications

This method activity has equal importance to the first one, as the combined annotation of both I/O parameters and the operations themselves via actions can enable achieving higher service discovery precision. To realise this activity, we relied on the realistic assumption that it is hard to identify domain-specific action ontologies as they do not exist in most existing domains. As such, we decided to use the high-level action sub-ontology in Schema.org. This decision is valid as even high-level actions can enable discerning between service operations with the same signature. This was also justified and exemplified in Section 3.2. Further, Schema.org's action sub-ontology is quite rich, covering multiple actions that can abstractly match most RESTful service functions/methods. This was also evident by manually checking each OpenAPI specification from EMB – all service operations could be manually mapped to specific actions from that sub-ontology that well match their intended semantics.

Based on the above rationale, we have implemented a specific LLM-based approach to realise this method activity. This approach again relies on issuing a carefully crafted prompt to an LLM and getting back the respective (annotation) result. This prompt follows the Persona and Template prompt patterns. It initially advises the LLM to play the role of an expert in ontology mapping and explicates the actual task to perform. Then, it includes a set of instruction-oriented sections. Finally, it incorporates the OpenAPI specification to be annotated.

This prompt has been incorporated into the article package linked as the end of this article. In the following, we summarise the content of its instruction-oriented sections:

- *Ontology Element Selection*: The LLM is instructed to pick up the best possible action from Schema.org that best matches the service operation at hand (based on its description according to the main verbs used). In case no suitable match is found, the LLM must select the most general action, mapping to the URI: https://schema.org/Action. In addition, the section covers corner cases occurring in specific domains. Please see two such cases below:
  - In the transaction domain, if the verbs used in the operation description indicate the initiation, confirmation, or finalisation of a transaction or process, the most suitable action to select is https://schema.org/TradeAction or any suitable from its subtypes (like https://schema.org/PayAction).
  - In the mathematical/evaluation domain, if the operation verbs indicate computation, transformation, or numeric/text calculation, the best possible action is https://schema.org/SolveMathAction. Otherwise, if the verbs indicate evaluation, validation, or condition checking, the action to be selected is https://schema.org/AssessAction.

  Finally, in case no verb is found in the operation description, then:
  - If the description relates to diagnostics, error handling, health check, or status verification, the https://schema.org/AssessAction must be picked up.
  - If it relates to data fetching of retrieval, the https://schema.org/ReadAction must be selected.
- *Verb Extraction Rule*: This section indicates to the LLM which are the sources for identifying the right verb to consider in the service operations's specification. The primary source is indicated to be the operation identifier (first verb-like token within the identifier). While the secondary source is the start of the operations's textual description. In fact, the secondary source verb overrides the primary source one in case that the former is imperative and both conflict with each other. The LLM is also instructed to ignore nouns that include action words. Finally, a corner case is covered where the verb 'options' in the operation identifier (e.g., HTTP OPTIONS verb/method) should be mapped to https://schema.org/AssessAction as it indicates the necessity to probe or check the service's operations.
- *Selection Criteria*: the LLM is instructed to match based on the semantics of verbs by considering also the context (i.e., the operation's textual description). Further, only action-related valid terms

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

from Schema.org must be used. Finally, it is repeated that the best possible match should be supplied per each service operation relating to a subclass of https://schema.org/Action. If no match is found, then https://schema.org/Action should be the matched action.
- *Constraints*: the LLM is instructed not to utilise any element from a different ontology than Schema.org. Further, it is indicated that the LLM should not hallucinate, constructing action-related concepts that do not exist in Schema.org.
- *Output Constraints*: the LLM is instructed to always provide a full, valid and resolvable URI for the ontology concept mapped.
- *Output*: the LLM is dictated to only output a JSON array, including as entries the matched operation's identifier, and the full URI of its matching action from Schema.org.

To conclude, our semantic action annotation approach mapping to the method's second activity exploits the transformation power of LLMs to reach its main goal, i.e., map each service operation to a specific action from the Schema.org action sub-ontology. As indicated in Section 6.2, our approach attains high annotation accuracy levels, well acceptable for elevating service discovery accuracy.

3.6. Automatic Transformation of OpenAPI Specifications

While not stated in our previous method's analysis, all annotations identified by using LLMs are incorporated into the respective OpenAPI specification at hand at suitable places, which include the operation definition in the context of an action annotation, the input parameters definition when such parameters represent datatype properties and not classes, and the schema elements referenced in request/response bodies, as these map to ontology classes [23]. The annotations are inserted by exploiting the general extension mechanism in the OpenAPI standard and the extension pattern suggested in [24]. Based on this pattern, an *x-refersTo* element is inserted at the respective place with two attributes: *ontologyElement*, referring to the ontology element that annotates the respective OpenAPI specification's parameter/operation, and *sourceOntology*, mapping to the used ontology's URI.

Once OpenAPI specifications are annotated, the current method just attempts to process them to produce a set of *Service*s based on the service representation model presented in Section 3.3. As this transformation is more or less trivial, this article does not analyse it, but more details can be found in the next PhD thesis [15].

We should stress that for each *Artifact* (*Service*, *Function*, *Input*, *Output*) (in the generated representation of each service), we also produce TFIDF and LLM-based embeddings vectors. These vectors are exploited in the service discovery phase. To this end, in this sub-section, we will supply specific names to them and explain based on which information pieces they are constructed.

- *service vector*: it is produced based on the following information: the service's name, title, summary and (textual) description in the specification's *Info* section, the names and (textual) descriptions of all service's tags in the specification's *Tags* section and the information utilised to construct the vectors of all operations in the specification.
- *operation vector*: it is produced based on the following information: the operation name (mapping to the *operationId* field or, when it is absent, to a combination of the HTTP verb and relative path of the operation), the operation's textual description and summary, the operation's tags, and information related to the operation's I/O vectors.
- *operation input vector*: it is constructed per each operation's input parameter depending on the parameter's kind. In case of single (i.e., not entity/class-based) parameters, the parameters' name and description are only considered. On the other hand, for complex (i.e., class-based) parameters, we also consider the names of their properties/fields.
- *operation output vector*: only one such vector is constructed per operation. It is treated similarly to the case of input vectors, as an operation output can map either to a simple or a complex type.

The vector construction process is different depending on whether we construct a TFIDF or an LLM-based embeddings vector. In both cases, the initial input is a String that concatenates all

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

information pieces by introducing a simple space in between them. For LLM-based embeddings vectors, we do not conduct any kind of preprocessing, as this could modify the respective meaning or intent. On the other hand, for TFIDF vectors, a specific pre-processing is performed on the overall concatenated information:

- Inclusion of a space between adjacent lowercase and uppercase letters in a word. This enables splitting a CAMEL-case word into multiple sub-words
- Replace underscore characters with a space character to split snake-case words
- Replace non-letter characters with a space
- Lowercasing the remaining characters
- Splitting the overall string into multiple tokens/words based on white space characters
- Removing stop-word tokens
- Lemmatising the remaining tokens based on the Stanford Core NLP pipeline[2]
- Producing a final array of String-based tokens

The above list of vectors is not exhaustive; further vectors are defined in some functional service matchmaking algorithms that we propose in Section 5.

3.7. *Automatic LLM-Based Service Request Structuring and Enrichment*

To realise this phase of our research methodology, we have devised a method, incorporating two main activities, which are sequentially executed based on the process depicted in Fig 4. These two activities will now be analysed in the next two sub-sections, respectively.

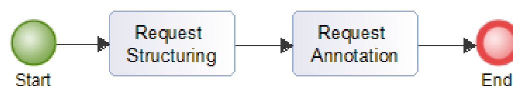

Figure 4. The request structuring and annotation process.

3.7.1. Automatic LLM-Based Service Request Structuring

In this first method activity, we attempt by using LLMs to structure accordingly a functional service request, coming in a single text form, into a *Function* such that both service operations and requests are equivalently represented. The use of LLMs is necessary as this structuring is not trivial: it is difficult to discern which is the verb determining the action to be performed by the needed service operation while it is not always evident which are the I/O parameters of such an operation (from the objects involved in the request).

This activity is realised by consecutively executing two main tasks: (a) LLM-based request structuring and (b) request vectors computation. While the first task maps to the main activity goal, the second is a side-effect of the first as, apart from structuring the service request, we need to compute all necessary vectors to support matching the request with the registered RESTful services.

These vectors' computation is similar to that followed for service operation vectors. The sole differences are: (a) the information about I/O parameters is limited to their names; (b) to distinguish between the service operation-related vectors, a different naming was applied, explicated as follows:

- *request vector*: it is computed similarly to an *operation vector*.
- *request input vector*: it is computed similarly to a *operation input vector*
- *request output vector*: it is computed similarly to a *operation output vector*

First Method Activity: LLM-based Request Structuring

As in the case of other method activities or tasks, we have realised this activity by constructing a carefully crafted prompt, which is issued over an LLM to retrieve the respective structuring result. This prompt follows the Persona and Template patterns. Further, this prompt is not zero-shot but

[2] https://stanfordnlp.github.io/CoreNLP/pipeline.html

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

includes specific examples to enable the LLM to properly understand the peculiarities of RESTful services (which are analogous to the HTTP verb used and thus the business entity-based management action implemented, like an update or delete of a business entity).

The prompt is also available in the article package linked at the end of the article. It starts by instructing the LLM to play the role of an expert in properly structuring textual requests for RESTful services and explicating the main task to be performed. Then, specific guidelines are supplied in respective prompt sections. Finally, the prompt ends by supplying the original service request text.

The main prompt sections are as follows:

- *Action-focused*: indicates to the LLM that when multiple verbs are involved in the textual request description, the one mapping to the request's intent must be selected. Further, detailed mappings are supplied from typical[3] and non-typical[4] verbs to specific "standard" actions from the action hierarchy in Schema.org. Each mapping includes a grouping of semantically-relevant verbs to a "standard" verb. For instance, "list", "find", "search" were mapped to "search" action.
- *I/O-focused*: dictates that single text must be returned per input/output and not structured/nested fields. Further, when multiple entities are involved (in the request), the one fetched or affected by the requested service operation should be the (single) output, while entities playing the role of subject, filter, scope, location or qualifier must be mapped to input parameters. When an action (implicitly) imposes identifying an entity to be processed or returned, an input parameter mapping to the identifier of that entity must be added. Finally, when the input is not specified or is considered irrelevant, the input parameter section should correspond to an empty array.
- *Fallback*: The fallback logic affects the handling of two main cases:
  1. when the request's textual description includes alternatives of entities or parameters, the most general or representative term must be returned
  2. when the request's textual description is ambiguous in terms of the designated action or I/O, the LLM should make the best reasonable guess for the request's extracted sections, instead of returning "unknown" or "unspecified".
- *Output format*: The output format actually corresponds to the *Function* entity in the service representation model (without the TFIDF and LLM-based embeddings vectors). Further, the LLM must not add commentary in the response, structured/nested fields in inputs & output sections, and not make up terms.
- *Examples*: Three examples were supplied covering three main management operations (update, delete and collection retrieval) that can be performed on business entities, respectively. They were termed necessary as some LLMs did not fully follow the previous instructions and, e.g., did not include as input the identifier of an entity to be deleted or did not include as output the entity being updated.

As will be shown in Section 6.3, the request structuring accuracy reaches 96% (for two state-of-the-art LLMs), a quite high and satisfactory level, suitable to support the service discovery phase.

3.7.2. Automatic LLM-Based Service Request Annotation

This method activity aims to fully annotate a structured service request across all of its sections. This means that both the requested operation (in terms of its intended action) and its I/O parameters are annotated with ontology elements. This annotation relies on our internal ontology with Schema.org as a fallback concerning the I/O sections and the action sub-ontology of Schema.org concerning the action section. As such, the structured service request is annotated with the same means as those utilised for the registered services' OpenAPI specifications.

Further, the annotation logic is very similar and relies on the use of a carefully crafted prompt. The sole difference is now that the annotation is conducted at one shot instead of being split into

[3] they convey similar or equivalent semantics to standard REST verbs like create, update, read and delete
[4] correspond to specialised actions like search, download or pay

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

two different prompts and thus tasks. The main rationale for this is twofold: (a) the prompt's size is now smaller as the prompt now incorporates just a structured request instead of a full OpenAPI specification. This signifies that it will be easier for an LLM to follow more complex instructions than in the case of a very large prompt; (b) the action part of the request is already normalised, so it is easier to generate the action-related annotations. In fact, the (current) prompt now incorporates mappings from "standard"/normalised verbs into specific actions from the Schema.org action sub-ontology/hierarchy.

As the prompt is more or less a unification of the prompts utilised for the I/O and action annotation of OpenAPI specifications (with a simplified version for the action annotation prompt), which were detailed in Section 3.5, we do not need to analyse it. However, for completeness and reproducibility reasons, it is included in the article package linked at the end of the article.

As will be shown in Section 6.3, the overall annotation accuracy for structured service requests by following our method reaches 91% in the case of one state-of-the-art LLM (Sonnet 4). This high accuracy level is considered more than satisfactory for facilitating the service discovery phase and increasing its accuracy via the introduced semantics.

Overall, the way we have implemented our research methodology via LLM-based methods has enabled us to annotate with a very high accuracy both RESTful services and their requests. Further, it enabled us to equivalently represent both service operations and requests. Finally, this semantically-enhanced representation incorporates further ingredients, such as TFIDF and LLM-based embeddings vectors, that can facilitate the sole or combined use of different service matching techniques. Thus, in our view, all this preparatory work can greatly facilitate service discovery and elevate its accuracy. This will be empirically validated via our conducted evaluation, detailed in Section 6.

4. System Architecture & Implementation

4.1. System Architecture

Our methodology is backed up and complemented by a RESTful service registry system that takes a RESTful API form, which can be used to integrate the registry with development frameworks and service marketplaces. This system is organised based on the CSR (Controller-Service-Repository) architectural pattern, a variation of the well-known MVC (Model-View-Controller) pattern.

Fig. 5 depicts our system's overall architecture. As can be seen, it comprises ten (10) components, which will shortly be analysed below:

- *Controller*: it is the most central component, playing the role of the *Controller* in the CSR pattern. It exposes the service registry interface to the outer world. Further, it is responsible for receiving requests, delegating their processing to the right service at the service layer and returning the respective response. Finally, it validates the request's and returns appropriate error responses with correct HTTP status code when such a validation fails.
- *OpenAPI Service*: it handles the first three phases of our research methodology, as it covers the management of RESTful services and especially their OpenAPI specifications. To realise this management, this service communicates with the *Code Repository Service* to fetch a service's source code and filter it, and then with the *LLM Service* to produce the service's OpenAPI specification. Further, it communicates with the *LLM Service* to construct the internal ontology, with the *Ontology Service* to store this ontology and then back with the *LLM Service* to annotate the services' OpenAPI specifications. Finally, it communicates with the *RESTful Service Repository* to store the managed services (according to our service representation model) and their OpenAPI specifications.
- *Discovery Service*: it handles the last two phases of our research methodology. It communicates with the *LLM Service* to structure and annotate the incoming service discovery requests. It also interacts with the *Ontology Service* when executing semantic-based service matchmaking algorithms. This component is a wrapper of all service matchmaking algorithms we have implemented. Thus, it can be configured to execute any of them.
- *LLM Service*: it interacts with the *LLM Proxy* to issue LLM prompts or request the calculation of LLM-based embedding vectors.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

- *LLM Proxy*: interfaces with an external *LLM API*, which might be offered by a LLM provider or a LLM marketplace, so as to allow the *LLM Service* to flexibly utilise the LLM of its choice.
- *Code Repository Service*: it fetches via Git a service's source code based on its Git-based *Code Repository* URL and places it at an appropriate folder within the local file system.
- *Ontology Service*: it is responsible for ontology management in cooperation with the underlying *Knowledge Base*. Further, it supplies reasoning and semantic querying facilities over this *Knowledge Base*, which can be exploited by semantic functional service matchmaking algorithms. It can be imagined as a wrapper of the underlying *Knowledge Base* that exposes only the necessary interface, thus hiding low-level ontology management details.
- *RESTful Service Repository*: it plays the role of a *Repository*, which stores and updates service objects (according to our service representation model) in cooperation with the underlying *Database* by exploiting the ORM technology.
- *Database*: it is a relational DB enabling the transactional storage, updating and querying of information related to RESTful services, including their OpenAPI specification.
- *Knowledge Base*: it enables storing ontologies (e.g., internal and Schema.org) while also providing reasoning and querying operations over them.

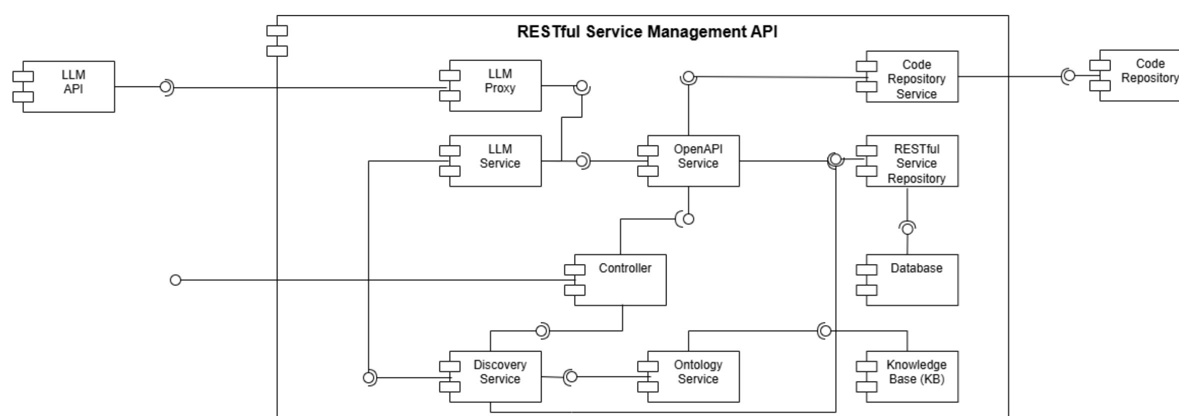

Figure 5. Component diagram depicting the service registry architecture.

Due to page restrictions and the fact that the system development is not at the core of this article's main contributions, we do not supply any other design artefact related to our service registry. However, the interested reader can find more details about our registry's design in [15]. In any case, we declare that this design conforms to our research methodology and its intended behaviour.

4.2. Implementation Details

Our RESTful API, offering service registry facilities, was developed in Java as a Spring Boot application. This application exploits classical ORM technology for interconnecting with the *Database* by relying on the use of Hibernate[5] and spring-boot-starter-data-jpa[6] libraries. The underlying relational database is PostgreSQL[7], supporting the efficient manipulation of both JSON data, such as OpenAPI specifications, and vector data.

The manipulation of OpenAPI specifications by the *OpenAPI Service* relies on the swagger-parser and the swagger-models libraries from the open-source Swagger Parser project[8]. While the *Code Repository Service* was realised by exploiting the Eclipse's JGit library[9].

[5] https://hibernate.org/
[6] https://docs.spring.io/spring-data/jpa/reference/jpa/getting-started.html
[7] https://www.postgresql.org/
[8] https://github.com/swagger-api/swagger-parser
[9] https://github.com/eclipse-jgit/jgit

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

The *LLM Service* was implemented by using OpenAI's Java library[10], while the *LLM Proxy* via liteLLM[11], which supports OpenAI's Java library. liteLLM was configured in turn to rely on the OpenRouter[12] platform API to access a wide range of free and proprietary LLMs.

Finally, concerning the *Discovery Service* and its underlying service matchmaking algorithms, we relied on two main libraries:

1. the well-known Smile library[13], so as to preprocess textual descriptions, produce TFIDF vectors out of them and compute their similarity based on the cosine similarity measure.
2. Apache's Jena library[14] so as to interact with the underlying *Knowledge Base* to query or reason over ontology-based data.

Further, the *Knowledge Base* was implemented via Apache Jena's*TDB2*[15] high-performance RDF store to realise the actual RDF/ontology query, reasoning and manipulation capabilities.

We conclude our API's implementation details by emphasising that this API has been dockerised. This means that a Dockerfile is included in its source code to enable producing its Docker image, corresponding to the API's core Spring Boot application part. Further, in the same codebase, we incorporated a docker-compose.yaml file to support our API's complete deployment in a physical or virtual machine via Docker Compose[16] by orchestrating three containers[17]. In this way, we have managed to make our API platform independent and take a cloud-native application form, which can be deployed and scaled in cloud environments.

5. Service Discovery Algorithms

5.1. Introduction

To support functional service matchmaking, we have implemented three families of functional service matchers: (a) TFIDF-based, (b) embeddings-based, and (c) LLM-based. All of these matchers implement the same Java interface called `Matcher`, which includes the operation: `public List<MatchResult> calculateSimilarityScore(String serviceRequest)`. This operation takes as input the functional service request in textual form and returns the matching service operations in the form of a list of *ServiceMatch*es (see service representation model in Section 3.3).

In each family of matchers, there is only one class that implements the `Matcher` interface, which realises all the service matchmaking algorithms/matchers in this family. The selection of the matcher to use is enabled at the class instantiation time (i.e., the constructor) by using an enumeration covering all family's matchers.

In the following, we will analyse each family of matchers in different subsections. Further, in a first subsection, we will explain a specific capability, offered to all our matchers to enable them to increase their service discovery accuracy.

5.2. Request Paraphrasing

A service request might be vague and not use consistent domain terminology, thus leading to reduced service discovery accuracy. To address this issue, we supply the capability of request paraphrasing to all our service matchers. Based on this capability, by producing variations that attempt to "correct" the original request, it is possible to produce more precise service matchmaking results that might better satisfy the intent the service requester might originally have had in his/her mind. In fact, the main goal of request paraphrasing is twofold: (a) allow the use of different words and

[10] https://github.com/openai/openai-java
[11] https://docs.litellm.ai/docs/
[12] https://openrouter.ai/
[13] https://haifengl.github.io/
[14] https://jena.apache.org/
[15] https://jena.apache.org/documentation/tdb/index.html
[16] https://docs.docker.com/compose/
[17] one concerning the API itself based on its image, one for liteLLM and one for PostgreSQL

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

phrases to express the same request, which could enable to increase the service matchmaking accuracy in lexicographic matchers like TFIDF and embedding-based ones; (b) increase the possibility of proper request annotation, especially in case that a full request annotation fails for any reason (e.g., input or output not annotated at all), such that semantic or hybrid matchers can also have their accuracy increased.

5.2.1. Request Paraphrasing & Results Merging

A `Matcher` equipped with paraphrasing should match both the original query and its paraphrases against the registered service operations and merge all results in the form of a single list of *ServiceMatch*es. As paraphrasing is considered a generic capability to be incorporated in any matcher and all matchers expose a single method for matching a request with the registered service operations, it has been decided that the respective operation that applies paraphrasing and merges the lists of results is implemented in the `Matcher` interface. In the following, we supply a Java-based pseudo-code that implements paraphrasing-enhanced service matchmaking.

Listing 3. Request paraphrasing enhanced service matchmaking

```
List<MatchResult> calculateSimilarityScore(String request,
boolean expandRequest, LLMService service){
  if (!expandRequest) return calculateSimilarityScore(request);
  else{
    List<MatchResult> results = calculateSimilarityScore(request);
    List<String> requestVersions = service.paraphrase(request);
    for (String requestVersion: requestVersions){
      List<MatchResult> newResults = calculateSimilarityScore(requestVersion);
      results.addAll(newResults);
    }
    return consolidateResults(results);
  }
}
```

As shown in Listing 3, the `Matcher` interface offers the implemented operation `calculateSimilarityScore()`, which takes as input both the original request and a boolean parameter that signifies whether the request should be paraphrased or not. If not, then `calculateSimilarityScore(request)` is called, which is implemented by the real matchers. Otherwise, the request paraphrases are produced, and then each paraphrase as well as the original request are matched against the registered services (operations) based on the abstract operation `calculateSimilarityScore(request)`. Each time a request or paraphrase is matched, the respective results are added to an overall list of results. Finally, the results are consolidated and then returned.

The result consolidation is performed by another operation of the `Matcher` interface called `consolidateResults()`, which takes as input the merged list of results and attempts to single out each unique match by considering its highest semantic/lexical category and score. Please note that, depending on the matcher, either one of the two possible categorisations (lexical vs. semantical) or both might be enabled. In the following, we will detail how request paraphrasing has been realised, as implemented in the `paraphrase(request)` operation. Please note that this operation is offered by the *LLMService* (i.e., a specific component in our system architecture – see Section 4), which is passed as a third parameter in the implemented `calculateSimilarityScore()` operation.

5.2.2. LLM-Based Request Paraphrasing

To support request paraphrasing, we have relied once again on the facilities of an LLM . To this end, we have constructed a carefully-crafted prompt that once again exploits the Personal and Template prompt patterns. This prompt has been incorporated in the article package linked at the end of this article.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

The prompt starts by instructing the LLM to undertake the role of a clever assistant that generates N paraphrases of the original request with the main objective of increasing the matchmaking accuracy without changing the intent, operation type or side effects of the requested service operation. Thus, the LLM should not introduce new actions, side-effects or assumptions. Further, in at least $N/2$ paraphrases, different lexicographically (but semantically equal) terms must be utilised to denote the same main entity of the original request. In addition, the LLM is encouraged to change the wording and sentence structure without generalising or specialising the original request. Finally, the LLM must only deliver the constructed paraphrases, one per line, and not include any commentary, explanation or label.

5.3. TFIDF-Based Matchers

This matcher family relies on using IR techniques to realise service matching. In particular, it first maps both services and their requests into a TFIDF vector space and then applies the well-known cosine similarity metric to compute the similarity between a service request and each service (operation) registered. Further, to appropriately categorise the service operations being matched, certain category-specific thresholds are applied over the computed cosine similarity between the service operationa and request. As a result, service operations become matches only when they belong to the three categories of the lexical dimension. All matchers in this family have request paraphrasing variants, a side effect of inheriting this capability from the `Matcher` interface.

In the following, we first explain how service categorisation is performed. Then, we analyse one by one the matchers belonging to this family.

5.3.1. Service Operation Categorisation

As our focus is on matching service operations rather than whole services, the categorisation concerns mainly service operations, while service filtering is merely a consequence of service operation categorisation. In particular, as long as a service includes operations that match the request and belong to one of the three lexical categories, it is kept in the result list as a *matching service*.

Before effective service (operation) matching can take place, there is a preparatory phase to calibrate the category-based thresholds. These thresholds are as follows:

- *COMPLETE_THRESHOLD*: maps to the threshold for the *COMPLETE* lexical category. If a similarity score is above this threshold, the respective service operation is classified under this category.
- *PARTIAL_THRESHOLD*: concerns the threshold for the *PARTIAL* lexical category. If a similarity score is above this threshold and below *COMPLETE_THRESHOLD*, the respective service operation is classified under this category.
- *POSSIBLE_THRESHOLD*: corresponds to the threshold for the *POSSIBLE* lexical category. If a similarity score is above this threshold and below *PARTIAL_THRESHOLD*, the respective service operation is classified under this category.

When a service operation is below the *POSSIBLE_THRESHOLD*, it is discarded. In addition, the following ordering relations must always hold under threshold calibration:

$$0.0 < POSSIBLE_THRESHOLD < PARTIAL_THRESHOLD < COMPLETE_THRESHOLD <= 1.0$$

The following calibration process is applied over all TFIDF and embeddings-based matchers. It relies on the main requirement that a request set exists in which, for each request, there is one service operation that fully matches it and one or more operations from the same service that partially match it. Thus, this request set matches a single service each time and only one of its operations fully.

Calibration is conducted by requiring that each matcher is configured to return the similarity of all service operations with each service request from the set. Then, we match each request from the set with all registered service operations, and we record as s_{best} the similarity of the operation that fully matches the request and as s_{second} the highest similarity of all operations that partially match the

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

request from the same service (that includes the fully matching operation). Finally, after recording all these similarities for all requests in the set in two vectors named as v_{best} and v_{second}, we calculate the value of the three category-based thresholds as follows:

- $COMPLETE_THRESHOLD = percentile(v_{best}, 5)$ – we consider the percentile of 10% of all values as a threshold such that we cover most of the best similarity values (90%).
- $PARTIAL_THRESHOLD = percentile(v_{second}, 80)$ – we consider the percentile of 80% of all values as the threshold, so we are more conservative as there is a need to have strong evidence that a specific match has a similarity value that signals a partial match.
- $POSSIBLE_THRESHOLD = percentile(v_{second}, 40)$ – this corresponds to a heuristic baseline that is satisfactory for our purposes. However, in the future, we will opt for a more sophisticated technique, which considers the overlap region between positive and negative distributions.

5.3.2. Core TFIDF Matcher

The *Core TFIDF* matcher focuses solely on the service operation vector space, such that it can enable the matching between service requests and operations. This vector space is constructed from all operation vectors of all services. This construction is rational as the focus of our service discovery phase is on service operations, and service requests have the same semantic granularity with operations.

As the process of matching service operations with requests has been already designated previously, we name the paraphrasing-based variant of this matcher as *Enhanced Core Matcher*. Further, we indicate that this matcher maps to the `SIMPLE` strategy in the algorithm-based enumeration incorporated in the implementation class of the `Matcher` interface for this family. This enumeration is called *TFIDFMatchingStrategy*. In addition, we stress that this matcher has excellent execution performance, especially in terms of its base variant, as it just computes vector similarities and applies thresholds over them. Thus, it does not rely on expensive calculation procedures or the exploitation of external LLMs. Its time complexity is $O(N)$ where N is the total service operation number. Thus, service matchmaking time linearly increases with the number of service operations.

The *Core TFIDF* matcher is expected to have a low level of service discovery accuracy, especially when the service OpenAPI specifications rely on a different terminology than that used in service requests. However, we expect that its paraphrasing-based variant might behave better due to the fact that the original request's variants might exhibit a higher probability of having common terms with the service specifications.

5.3.3. Structured TFIDF Matcher

By considering that both service operations and requests have three core sections, the rationale for this matcher is that it focuses first on the section level to compute the respective similarities and thus "match' service operations and requests per each section, and then combines these similarities into a single similarity value to globally match" them. Thus, first section-level signals are collected, and then an overall signal (i.e., similarity) is computed, utilised to filter and categorise each candidate service operation (according to the method designated in Section 5.3.1).

Considering two levels of similarity computation enables putting more significance on one section over another by using relative weights in the global level calculation. However, we make the matcher configurable to use weights that better reflect user preferences or the real situation in a particular domain of interest. In this respect, the overall similarity between a service operation and request is calculated as follows:

$$sim(req, op) = w_{action} * sim_{action} + w_{input} * sim_{input} + w_{output} * sim_{output}$$

where $w_{action}, w_{input}, w_{output}$ are the weights given to the action, input and output sections, respectively, and $w_{action} + w_{input} + w_{output} = 1$. The latter condition is necessary as the weights are relevant – could be computed based on the Analytic Hierarchy Process (AHP) [25] – and the overall similarity should still be in the range $[0.0, 1.0]$.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

In the following, we explain how the similarity is computed for each separate section:

- *action section*: we rely on information that purely covers the actual action realised by a service operation without incorporating any kind of input or output information. To this end, we compute a *partial operation vector* for the operation at hand, which is similar to the respective *operation vector* without including information from the respective *operation input* and *output vector*s. Thus, this new vector is computed from the service operation's name, its textual description and summary and its tags. At the service request side, we first structure the request so as to easily obtain its three main actions, and we take only its action section to compute its vector that we call *partial request vector*. Finally, we compute the cosine similarity of the *partial request* and *operation vector* and assign it to sim_{action}.
- *input section*: The computation here is more complicated as it takes into account four distinct cases:
  1. Both the request and operation do not have any input parameters, so their input similarity (sim_{input}) is 1.0.
  2. Both the request and operation have input parameters. In this case, we construct overall input vectors for them. The *overall operation input vector* is constructed by concatenating with a space as a separator the information concerning each operation's input parameter. While the *overall request input vector* is constructed by concatenating the names of the request input parameters (in the request input section), separating them with a space, too. Finally, we compute the cosine similarity between these two vectors and assign it to sim_{input}. This approach is better as it considers the overall similarity between all service operation and request input parameters. While a naive approach that constructs separate vectors per input parameter, "matching" them and then computing an overall similarity would fail as the probability of having input parameters unmatched due to uncommon terminology would increase.
  3. The operation does have input parameters, while the request does not. This is a rather problematic situation, as the requester will not be able to call the service operation because he/she expects no input parameter for the service. Further, this can be a signal of different intents between the service operation and request. As such, we consider that: $sim_{input} = \frac{1-penalty}{|I|}$ where *penalty* is the penalty value for the similarity score, configured to take the high value of 0.8, and $|I|$ is the number of the operation's input parameters. Thus, the overall similarity gets even smaller with the increase in the number of input parameters in the service operation.
  4. The operation does not have input parameters, but the request has. This is not a very problematic situation like the previous one. However, as it designates a potential intent mismatch, we supply a smaller penalty value, configured by default to 0.5, while the overall input similarity gets: $sim_{input} = \frac{1-penalty}{|I_R|}$ where $|I_R|$ is the number of the request's input parameters .
- *output section*: similarly to the case of input section similarity calculation, we discern between three cases:
  1. the service request and operation do not have an output, so sim_{output} is set to 1.0.
  2. the service request and operation do have an output. Then, we compute the cosine similarity between the *operation output vector* and the *request output vector* (see Sections 3.6 and 3.7.1) and we assign it to sim_{output}.
  3. one of them has an output, and the other does not. In this case, it holds that: $sim_{output} = 1 - penalty$ where *penalty* is configured to be high (0.8).

This matcher maps to the `STRUCTURED` member in the `TFIDFMatchingStrategy` annotation. We call its paraphrasing version *Enhanced Structured TFIDF* matcher.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

The execution performance of this matcher is worse than that of the *Core TFIDF* matcher as it needs to structure the request based on one LLM call while it has to perform 3 vector similarity calculations in the worst case. So, its time complexity is $O(3 * N + c)$ where c is the cost of one LLM call.

Due to the service request structuring, there is a removal of the respective noise in the request (by removing words that do not contribute to the request's intent). So, we believe that this matcher might have better service discovery accuracy than the *Core TFIDF* one. However, section matching might have a higher failure probability when the respective terminology is different between the service request and operation. Thus, this could negatively impact service discovery accuracy. To this end, it is not clear whether there can be accuracy benefits delivered by this matcher.

5.3.4. Ontology-Based TFIDF Matcher

This matcher is hybrid as it does not rely just on the main textual descriptions of service requests and operations, but also their ontology-based annotations. In this respect, as service requests and operations are annotated via the same ontologies, when they match but use different legicographic terms, some or all of their ontological annotations will be equivalent. As such, these common annotations can contribute to a higher similarity value, such that indeed a respective match can be inferred.

Based on the above observation, we enhance the *operation* and *request vectors* with the ontology-based terms based on which the operation and request have been annotated, respectively. We call these vectors *semi-semantic operation vector* and *semi-semantic request vector*.

We now supply a specific example of the following service request: "Obtain a specific manuscript". This request will be structured and then annotated. Based on the way structuring is conducted, the three main parts of the request will become: action → "read", input → "manuscript identifier", output → "manuscript". These will be then annotated with the following concepts from our internal ontology: "read" → "ReadAction", "manuscript identifier" → "identifier", "manuscript" → "Book". After removing the "Action" postfix[18] from the action annotation, we merge the original request with the identified annotations to obtain the final String based on which the semi-semantic request vector will be constructed: "Obtain a specific manuscript read identifier book".

Once this construction takes place for both the registered service operations and the service request, the matching process is equivalent to the one in the *Core TFIDF* matcher. However, after examining this matcher's results, we observed that it produces many irrelevant matches. In fact, the imrecision source is that there is no way to discern the actual action requested, such that when one service matches the request, many of its underlying operations match it with different action semantics/intents. This is well expected as the action section is under-represented compared to the combination of the input and output sections. Thus, the I/O sections contribute more to the final similarity result.

For example, by continuing our previous example, the aforementioned request would match most of the operations of a book management service, i.e., operations that modify or delete a book rather than return it.

To resolve this issue, we focus on the action section and especially its ontology-based annotation so as to apply a second-level filtering over the produced service (operation) matchmaking results. This means that even if a service operation has a similarity that is higher than the *COMPLETE_THRESHOLD* or the *PARTIAL_THRESHOLD*, it will be filtered if its action annotation is incompatible with the action annotation of the request. In this respect, we filter the service operations based on the actual action/function that they perform.

The focus of this filtering was on actions that create, update or delete entities. To this end, we have constructed specific rules that indicate which actions are compatible with these three clusters of actions. For instance, a "toggle" action is compatible with the update cluster while a "destroy" action is compatible with the delete cluster.

[18] in Schema.org, all action concepts have this postfix in their names

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

By continuing the previous example, via this filtering, the service operations that create, update or delete books will be filtered, even if they are very similar to the request. While operations that retrieve/read books will be maintained, which are those that must be returned. Thus, service discovery precision will be higher than that of the other matchers. This is one of the main benefits of relying on ontological annotations.

Overall, we believe that the consideration of ontological annotation will elevate service discovery accuracy both in terms of precision and recall. Recall will increase as the incorporation of ontological annotations in vectors can force them to contain more common terms, which will increase their similarity and thus the probability of being matched. On the other hand, action-based filtering will enable keeping precision at appropriate levels by removing results that do not match the request's intended action.

This matcher maps to the `ONTO` member in the `TIFDFMatchingStrategy` annotation. Further, we call its paraphrasing variant *Enhanced Ontology-based TFIDF* matcher.

Finally, concerning its time-based performance, this matcher requires the structuring and annotation of the service request via LLM calls. On the other hand, similarity computation is applied over semantically enhanced vectors per each service operation. So, if we regard that we have N service operations, the overall time complexity will be $O(N + 2 * c)$.

5.4. LLM-Based Embeddings Matchers

This matcher family utilises a different technique to compute a lexicographic similarity between the service requests and operations. In particular, instead of using TFIDF vectors, which require request/operation pre-processing and their eventual mapping to the TFIDF vector space, they rely on LLM-based embeddings vectors, which are produced via LLMs by relying on the original content of service requests and operations (i.e., the one we collect based on our research methodology)[19]. This is because the original content tends to better capture the contextual semantics and latent functional intent, while its pro-processing can lead to the removal of important information, reducing the precision in the aforementioned capturing.

We expect that the switch to this technique can increase service discovery accuracy compared to the previous one and to previous state-of-the-art approaches that rely on static embeddings. This is because LLM-based embeddings are more efficient than static embeddings (e.g., encoded in world vectors) and superior to TFIDF spaces as they better capture contextual semantics and latent functional intent, while they are robust to linguistic variations.

Similarly to TFIDF matchers, there is a single implementation class for the whole family, and the actual matcher to select relies on the use of a specific enumeration, given as input to the class's constructor. Further, there are paraphrasing variants for all family matchers. In the following, we analyse this family's matchers in separate subsections.

5.4.1. Core LLM-Based Embeddings Matcher

The service matchmaking in this matcher family relies on the cosine similarity computation between the LLM embeddings-based vector of the service request and operation, respectively and the application of the category-based thresholds. Where the latter thresholds are distinctly produced, through the aforementioned calibration process in Section 5.3.1, for each family's matcher.

However, by inspecting the results produced by this matcher, we observed a similar issue with respect to TFIDF matchers, as it tended to return irrelevant matches that do not match the request's intended action. This is due to the fact that the similarity of incompatible actions can be high by using LLM-based embeddings as these actions can be considered in general as relevant. As such, we decided to apply at a second level the same action-based filtering method as the one applied for the *Ontology-based TFIDF* matcher.

[19] We actually consider the name, textual description and summary of the service operation and not its whole content as it tends to be overwhelmed with I/O parameter information

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

In this way, we expect that this matcher will have better discovery accuracy than most TFIDF matchers, as it will tend to better capture the semantics and intent of all specifications (i.e., service requests and operations), plus remove irrelevant results through the application of action-level filtering.

The matcher's time-based performance will be similar to that of *Ontology-based TFIDF* matcher, but slightly worse. This is because apart from structuring and annotating the service request (time cost $2 * c$), there is a need to compute its LLM-based embeddings vector (time cost $1 * c$), while we assume that this vector has already been computed for all service operations, which should have already been annotated. Thus, the time complexity should be $O(N + 3 * c)$[20].

This matcher maps to the `SIMPLE` member in the `LLMEmbeddingsMatchingStrategy` enumeration. While we call its paraphrasing variant *Enhanced LLM-based Embeddings* matcher.

5.4.2. Onto LLM-Based Embeddings Matcher

While the consideration of action-based filtering makes the *Core LLM-based Embeddings* matcher slightly semantic, its similarity computation capabilities focus solely on the textual descriptions and neglect semantic annotations. As such, the goal of the current matcher (*Onto LLM-based Embeddings*) is to introduce annotations at the similarity computation level to make it more semantic.

However, contrary to the *Ontology-based TFIDF* matcher, the annotations are not incorporated directly in the vectors being compared, but a different approach is followed by considering a weighted sum of similarities. This is due to the fact that the direct use of annotations within a service operation/request's textual description will negatively impact the precision of its LLM-based embeddings vector, as the operation/request intent would be modified by using extra, annotation-related terms, while more emphasis will be put on the I/O sections as they usually incorporate more parts than the action one.

To this end, the similarity between a service operation and request is formally computed as follows:

$$sim(req, op) = w_{lex} * sim_{lex} + w_{sem} * sim_{sem}$$

where w_{lex}, w_{sem} are the relative weights given to the two different similarity components, the lexical (sim_{lex}) and semantic similarity (sim_{sem}) ones, respectively. The relative weights given to the aforementioned two similarity components should have a sum equal to 1.0. Further, they are configurable to take any value the user/programmer requires.

Both similarity components are computed by using the cosine similarity metric over LLM-based embeddings vectors. However, the lexical similarity is computed based on the original LLM-based embeddings vectors of the service operation and request. While the semantic similarity is computed over the original LLM-based embeddings vector of the operation and the semi-semantic vector of the request. The latter vector is constructed based only on the annotations of the structured service request, which are concatenated with an extra space as a separator, having the action annotation first, then the input annotation and finally the output annotation. As such, only the semantic component considers the annotations and only for the service request. This is a design choice driven by the following facts: (a) the service operation textual descriptions tend to be complete when produced by LLMs; (b) in such a case, the use of annotations is not required as LLM-based embeddings vectors capture well the contextual semantics and latent functional intent; (c) the service request might be vague such that it is more precisely described by its ontological annotations.

We expect that the accuracy of this matcher to be slightly higher than that of the *Core LLM-based Embeddings* matcher due to the consideration of further semantics. However, the matcher's time-based performance will be slightly lower as two instead of two similarities have to be computed per service operation, while also two LLM-based embeddings vectors have to be computed for the service request. In essence, the time complexity is $O(2 * N + 4 * c)$.

[20] in principle, vector computation is faster than request structuring and annotation, but here we assume that they all map to the same cost

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

This matcher maps to the `ONTO` member in the `LLMEmbeddingsMatchingStrategy` enumeration. While we call its paraphrasing variant *Enhanced Onto LLM-based Embeddings* matcher.

5.4.3. Structured LLM-Based Embeddings Matcher

This matcher is our novel, sophisticated contribution in functional service matchmaking, which is more structured as it focuses on section-related information, semantic by considering ontology-based annotations, and complex as it can apply alternative strategies for input (section) matching.

Vector Terminology

Before explaining the main matcher's logic, we introduce some additional vectors to establish the right terminology in our analysis:

- *partial operation vector*: constructed by considering the service operation's name, textual description and summary.
- *semi-semantic request vector*: constructed based on a String that concatenates the annotations in the structured request (action + input parameter annotations + output annotation).
- *semi-semantic operation input vector*: constructed based on a String that concatenates the annotations of all input parameters in the service operation.
- *semi-semantic request input vector*: constructed based on a String that concatenates the annotations of all input parameters in the request.
- *semi-semantic request output vector*: maps to an LLM-based embeddings vector, computed based on the request output's annotation.
- *semi-semantic operation action vector*: constructed from the action annotation of the service operation after removing the "Action" postfix
- *semi-semantic request action vector*: constructed from the action annotation of the request after removing the "Action" postfix.

Further, we have utilised the overall request/operation input vectors as originally introduced in the *Structured TFIDF* matcher.

Matcher's Logic

The overall similarity between a service request and operation is computed as the weighted sum of section-specific and context/domain similarity components. More formally:

$$sim(req, op) = w_{action} * sim_{action} + w_{input} * sim_{input} + w_{output} * sim_{output} + w_{context} * sim_{context} + w_{domain} * sim_{domain}$$

where $w_{action}, w_{input}, w_{output}, w_{context}, w_{domain}$ are the relative weights given to all these similarity components, signifying their relative importance. The sum of these weights must be equal to 1.0.

By default, the weights are configured to take the following values: $w_{action} = 0.3, w_{input} = 0.25, w_{output} = 0.25, w_{context} = 0.15, w_{domain} = 0.05$. This configuration reflects the relative importance of the similarity components as: (a) we supply a total weight of 0.8 to the most important components that "simulate" IOPE semantics; (b) contextual similarity is given more importance than domain similarity in the sense that context, along with action, completely cover PE semantics and similar/equivalent operations can be found in different domains.

In the following, we explain how the component similarities are computed:

- *action similarity*: computed via the cosine similarity between the *semi-semantic operation action vector* and the *semi-semantic request action vector*. Thus, we consider the action annotations in the service operation and request to compute it.
- *input similarity*: computed based on the input matching strategy configured. As this computation is more complicated, we detail it in the next paragraph.
- *output similarity*: the computation of this component is complicated, so we detail it in the second, next paragraph.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

- *context similarity*: the (service operation) context could be better covered via PE constraints. However, such constraints are not available. As such, we regard that the textual description of the service operation and of the request simulate the intended context. Thus, context similarity is computed by the cosine similarity between the *partial operation vector* and the *request vector*.
- *domain similarity*: as we do not have a description of the domain in the service operation and request, plus no relevant tags or annotations, we make the following assumption. A service covers multiple functionalities within a domain, so it includes sufficient information to cover it. Thus, the domain section for a service's operation can be the *service vector*. However, at the request part, we do not have any relevant information apart from the requested operation. Thus, the overall request document is considered the request's domain. Due to this under-representation, we decided to give a very small weight to this similarity component. Thus, eventually, domain similarity equals the cosine similarity between the *service vector* and the *request vector*.

Input Similarity

The computation of *input similarity* depends on the configured input matching strategy and the existence of input parameters in the service operation and request. To this end, the following four cases exist:

1. There are no input parameters in the service operation and request. Then, input similarity is 1.0.
2. The operation has input parameters, but the request does not. In this case, it holds that $sim_{input} = max(1 - penalty, max_input_sim(I_{op}, ssem_request_vector))$ where I_{op} is the operation's input parameter set and *penalty* is a specific configuration property to penalise the similarity due to the non-existence of input parameters in the request. It is originally configured to be equal to 0.8. The `max_input_sim()` is a function that computes the maximum similarity between each operation input parameter (i.e., the respective *operation input vector*) and the *semi-semantic request vector*. The main rationale is that we attempt to find whether there is a meaningful (LLM-based embeddings) similarity between any operation input parameter and the request's ontological description that could signify a respective correlation that might potentially unveil the ability of the request to somehow cover this input parameter. Further, we rely on the request's ontological description to remove any ambiguity kind.
3. The request has input parameters, but the operation does not. This case is symmetric with respect to the previous one. More formally, it holds that:

$$sim_{input} = max(1 - penalty, max_input_sim(I_{req}, partial_operation_vector))$$

   where I_{req} the request's input parameter set and `max_input_sim()` computes the maximum similarity between each request input parameter (i.e., its respective *request input vector*) and the *partial operation vector*. Again, we try to see in this way whether there is a semantic correlation between any request input parameter and the operation's partial description that could signify this parameter's coverage.
4. Both service operation and request have input parameters. In this case, the input similarity is computed as follows:

$$sim_{input} = w_{semantic} * sim_{input_semantic} + w_{lexical} * sim_{input_lexical}$$

   where $w_{semantic}, w_{lexical}$ are relative weights given to semantic and lexical-oriented component similarities, respectively. These weights are configurable, and their sum should equal to 1.0. The semantic input similarity is computed via the cosine similarity over the *semi-semantic operation input vector* and the *semi-semantic request input vector*. On the other hand, the lexical-oriented similarity's computation depends on the input matching strategy as follows:

   - *COMBINED_INPUT* (default): In this strategy, we regard each input parameter set (of service operation/request) as a unified (complex) parameter, and we attempt to compute these

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

parameters' overall similarity. As such, $sim_{input_lexical}$ is computed by the cosine similarity between the *overall operation input vector* and the *overall request input vector*.

- *DIFF_INPUT_AVERAGE*: In this strategy, we find the maximum similarity between each operation input parameter and all the request's input parameters. This means that we compute M similarities per each operation input parameter (where M is the number of request input parameters) and we identify the maximum, which is added to a specific variable. In the end, we compute the average over these maximum similarities by dividing this variable with the number of operation input parameters. Each individual similarity is computed by the cosine similarity between the *operation input vector* and the *request input vector* of the respective input parameters being matched. More formally: $sim_{input_lexical} = \frac{\sum_{i=1}^{N}(max_{sim}(v_i, I_{req}))}{N}$ where $max_{sim}(v_i, I_{req})$ is a function that computes the maximum lexical similarity between an *operation input vector* v_i and the request's input parameters (actually their request input vectors).
- *DIFF_INPUT_COVERAGE*: This strategy is similar to the previous one. However, instead of computing the maximum similarity between each operation input parameter and the request's input parameters, we explore whether one of these similarities is above a specific threshold called $INPUT_SIM$, configured by default to be equal to 0.7. If such a similarity is found, we consider the current operation's input parameter as matched/covered (by the request). Otherwise, if all similarities are below $INPUT_SIM$, we consider that the operation's input parameter as unmatched. By continuing this process, we count how many operation input parameters were covered, and then we divide the result by N (denotes the number of operation's input parameters) to compute an average value. More formally, $sim_{input_lexical} = \frac{\sum_{i=1}^{N}(matched_{sim}(v_i, I_{req}))}{N}$ where $matched_{sim}(v_i, I_{req})$ is a function that checks whether the similarity between the current operation input parameter (i.e., its vector v_i) and any request input parameter is above $INPUT_SIM$.

Output Similarity

The computation of sim_{output} is also complex and depends on the following three cases:

- Both the service operation and request do not have any output. In this case, output similarity is equal to 1.0.
- One out of the service operation and request does not have output, while the other does have. In this case, output similarity is computed as follows: $sim_{output} = 1 - penalty$ where $penalty$ was already introduced before and is by default equal to 0.8. Please note that penalisation is independent of the sub-case (whether the service operation or request does not have an output). This is because the lack of an output is a major signal of an (requested or offered) operation's intent, so it should lead to a major penalty when the other specification being matched does require or produce a specific output. For instance, consider the case of a service operation returning a specific news article and a requested operation deleting a news article. Both specifications will have the same input (the article's identifier), but the offered operation returns the news article, while the required operation does not return the deleted news article.
- Both the service operation and request have an output. In this case, output similarity is similarly computed as input similarity (in the respective similar case of input parameter existence) based on the following formula:

$$sim_{output} = w_{semantic} * sim_{output_semantic} + w_{lexical} * sim_{output_lexical}$$

The weights in the above formula are the same as in the similar case in input similarity computation. $sim_{output_semantic}$ corresponds to the semantic output similarity computed by the cosine similarity between the *semi-semantic operation vector* and the *semi-semantic output vector*. On the

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

other hand, $sim_{output_lexical}$ corresponds to the lexical output similarity that is computed by the cosine similarity between the *operation output vector* and the *request output vector*.

Performance and Accuracy Analysis

Performance-wise, due to this matcher's complexity, there is a need to compute multiple LLM-based embeddings vectors. By considering the worst case where the input matching strategy is *MAX_INPUT_AVERAGE* and the service request and operation have both input parameters and an output, there is a need to construct: the *partial output vector*, 2 *semi-semantic action vector*s, 2 *semi-semantic output vector*s and *M request input vector*s. Further, there is a need to compute 3 cosine similarities (for action, domain and context sections), 2 cosine similarities for the output section, and $K * M$[21] cosine similarities for the input section per each service operation. Finally, there is a need to structure and annotate the service request (cost $2 * c$). As such, time complexity becomes $O(N * (K * M + 5) + (M + 7) * c)$.

In terms of service matching accuracy, this matcher might have a better accuracy than the other family matchers, as it is more sophisticated and applies a two-level approach for similarity calculation with additional complexity in I/O sections, while it considers both action-specific and overall specification semantics in the form of respective annotations. However, it incorporates many configuration points (i.e., input matching strategy, weights, penalties), so its accuracy level can be influenced by the way it is eventually configured based on these points.

This matcher corresponds to the `STRUCTURED` member in the `LLMEmbeddingsMatchingStrategy` enumeration. While its paraphrasing variant is named *Enhanced Structured LLM-based Embeddings* matcher.

5.5. LLM-Based Matchers

Introduction

While the previous matcher families included hybrid matchers (i.e., lexicographic with semantic enhancements), all of them exploit ontology-based annotations in a lexicographic rather than a semantic manner, and each of them employs a different annotation exploitation degree. To this end, we expect that they can reach a moderate accuracy level. Our hopes lie in the semantic matcher family, which has the potential to further increase discovery accuracy, especially if it combines ontology-based subsumption with lexicographic techniques where appropriate (e.g., to bypass annotation accuracy and request ambiguity issues). Nevertheless, our second research question *Q2* is still effective, so there is a need to explore yet another potential matchmaking technique. In this case, we investigate if a pure LLM-based approach could enable reaching even higher service matching accuracy levels.

To this end, we have designed a specific LLM-based matching method, which is LLM-independent and does not require using semantic annotations. The second design choice was based on the observation that LLMs do employ semantics underneath their transformation procedures, thus they might be able to reach an even higher annotation accuracy themselves. Further, this method is capable of classifying service operation matches in both lexicographical and semantic categories, to suit different programmer/user needs and preferences, while it also offers the innovative feature of supplying the rationale for returning each operation match, thus catering for explainability.

Prompt Design

The matching method wrapped by the respective matcher, which can be configured to use any LLM, relies on using a carefully crafted prompt that conforms to the Persona and Template patterns. This prompt was included in the article package that is linked at the end of this article.

The prompt starts by instructing the LLM to act as an intelligent API matcher with the task to match a plain request with the operations of RESTful services (for which their descriptions are given

[21] K is the number of operation input parameters and M is the number of request input parameters

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

as input) and return a ranked list of service (operation) matches. The prompt is then organised into the following five main sections:

1. *Inputs*: details the two main inputs given to the LLM, i.e., the textual service request and the description of all RESTful services. The description of the latter corresponds to a list of *service documents*, i.e., the information based on which the *service vector*s were constructed. This information indeed does not include ontology-based annotations.
2. *Matchmaking Logic*: This is the most detailed prompt section comprising the following six (6) main sub-sections, which explicate the core service matching logic:
   (a) *Confidence Scoring*: explicates that the service operation similarity (or confidence score) is computed as the weighted sum of semantic (based on request and operation name & description), intent (based on request and operation action/intent), parameter (based on request and operation I/O) and domain context (based on topic overlap, tags and keywords). More formally:

$$sim(req, op) = w_{semantic} * sim_{semantic} + w_{intent} * sim_{intent} + w_{parameter} * sim_{parameter} + w_{domain} * sim_{domain}$$

   where $w_{semantic}, w_{intent}, w_{parameter}, w_{domain}$ are the relative weights given to these similarity-based components, which should have a sum equal to 1.0. By default, these weights are configured as follows: $w_{semantic} = 0.45, w_{intent} = 0.25, w_{parameter} = 0.2, w_{domain} = 0.1$. Thus, the semantic similarity is the highest, followed by the intent and I/O similarities. This is a similar configuration to the one we utilised for the *Structured LLM-based Embeddings* matcher, giving the highest cumulative weight to IOPE-based similarity components.
   (b) *Match Types*: Here the prompt explicates the two different match categorisation dimensions and their categories' semantics. Please refer to Section 3.3 for a detailed analysis.
   (c) *Service Confidence*: While our focus is on supplying service operation matches, we regarded the interesting feature to also match the services themselves via a confidence score and rank them. While this is not reflected in the final output produced for this matcher in our implementation, we intend to properly implement this feature in all matchers so as to present two separate rankings, one corresponding to matched services and one to matched service operations. In this respect, via service ranking, the programmer will have the ability to check first those services that better match his/her request and then focus on which operations in these services can be used to implement his/her intended functionality.
   The calculation of the service confidence score or similarity is as follows:

$$sim(req, service) = w_{service} * sim_{service} + w_{method} * max_{op}\{sim(req, op)\}$$

   where $w_{service}, w_{method}$ are the relative weights given to the two similarity components, respectively, with a sum equal to 1.0. While $sim_{service}$ is the actual similarity of the service with the request, and $max_{op}\{sim(req, op)\}$ denotes the maximum from the similarities between the service's operation and request. Thus, a service's confidence score depends on its actual similarity with the request as well as the best similarity between the request and its operations. The default values for the aforementioned weights are: $w_{service} = 0.6, w_{method} = 0.4$, thus giving higher relative importance to the actual service-to-request similarity.
   (d) *Ranking*To support the future service-first ranking, we require the LLM to produce only this ranking. Then, we obviously transform it to the currently supported operation-first classification and ranking. This service-first ranking relies on the ranking of services at the outer level and then the ranking of the service operations at the inner level.
   (e) *Include Contributions*: it is requested to include per matched operation the values of the individual similarity components to have a clear view of their contribution degree towards

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

the overall operation similarity. This is another interesting matcher feature not included in the other matcher families and our service representation model. However, we intend to update the latter model to cover it.

(f) *Include Explanations*: it is requested to provide explanations of why a specific operation was matched and why it got the respective confidence score and was classified in the respective (structural and semantic) categories.

3. *Output*: it is prescribed to produce a JSON object with a specific format, including the original request and the set of matched services. Each matched service is featured by its rank, name and confidence score as well as its matched operations. Each matched operation in turn is featured by its rank, name, similarity score, structural category, semantic category, an explanation of its matching and the contributions of the similarity components to the overall operation similarity score.
4. *Additional Instructions*: Six extra instructions are supplied, the most important of which indicate that the LLM should not consider historical usage in the calculation of the confidence scores and all scores (overall and component) should be normalised in the range [0.0, 1.0].
5. *Task*: Here, the main task to be performed by the LLM is repeated. Further, the LLM is instructed not to add commentary or explanations and thus output only the prescribed JSON object.

Outlook

Interestingly, the service matching process is reduced to a single LLM call and a small output-processing operation. Thus, the time complexity is $O(c + N * logN)$ as there is a need to make this LLM call and then process the LLM output in order to produce the ranked list of service operations that match the request. Please note, though, that the single LLM call is an over-simplification over the EMB dataset. In case of a larger service dataset, we would need to split the dataset into chunks that fit the LLM window context size. Thus, in the general case, $\lceil N/T \rceil$ chunks will need to be produced and processed, where T is the number of service operations descriptions that fit the context window size, which will raise the time complexity to $O\left(c * \lceil \frac{N}{T} \rceil + N * logN\right)$.

Please note that the $N * logN$ part is due to the need to re-arrange the matchmaking results. If, however, the alternative results ranking is to be applied also to other matchers, this cost factor will be common among all matchers. Further, to be fair, if the output instructions were similar to the expected effective service matches format utilised in the other matchers, this cost factor would not hold. Thus, the time complexity would have been $O(c)$ for small service sets and $O\left(c * \lceil \frac{N}{T} \rceil\right)$ for larger ones.

Concerning the service discovery accuracy, we cannot make any assumption about whether this matcher's performance will be better than that of the others, as we do not know exactly how well similarities are computed by LLMs – except for the case of TDIDF matchers, where the performance will surely be better. However, we expect that the accuracy could reach moderate or even higher levels by considering the literature about LLM performance in other IR-related tasks.

In any case, due to the various innovative features introduced, we believe that our LLM-based matcher is novel and might have the potential to greatly advance the state-of-the-art. We believe that this and the *Structured LLM-based Embeddings* matcher constitute our major contributions to the functional service matchmaking area.

6. Experimental Evaluation

As our two main contributions are the research methodology and the novel functional service matching algorithms (mapping to the last methodology phase), we have evaluated them both, and the goal of this section is to shed light on how such an evaluation was conducted and what its main results were. However, the first methodology phase (Automatic LLM-based OpenAPI Specification Generation) has already been evaluated in [21] and the I/O annotation part of the second methodology phase (Automatic LLM-based OpenAPI Specification Annotation) has also been evaluated in [23]. Further, the third methodology phase is trivial, so it does not need any evaluation. To this end, this

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

section will focus on analysing the evaluation of action annotation in OpenAPI specifications, service request structuring and annotation, as well as the functional service matchmaking algorithms. As such, particular sub-sections are devoted to these evaluations, respectively. Besides, a first sub-section is used to introduce the exact evaluation setup and structure.

6.1. Evaluation Setup & Structure

Our evaluation is purely experimental and relies on the EMB dataset. This dataset was selected as it includes both the source code and original OpenAPI specifications of 19 RESTful services, so it covered well our research methodology's objectives and phases. While other RESTful service datasets exist, which are larger in some cases, they do not contain source code or OpenAPI specifications [26]. Further, they usually provide textual descriptions of services and not their actual operations.

We have added to the EMB dataset the source code and OpenAPI specification of another RESTful service, covering the management of books in a library. Thus, the enhanced EMB dataset includes 20 services, while the total number of service operations is 163, maping to around 8 operations per service.

Action Annotation Evaluation

Based on this EMB enhanced dataset and our implemented method in the first methodology phase [21], we have automatically generated 20 OpenAPI specifications that are richer than the original ones. Further, we have also manually annotated these specifications based on their I/O and action parts to create a ground truth over the two respective annotation activities. This manual annotation was conducted by exploiting the internal ontology automatically produced out of the 20 automatically generated API specifications and the Schema.org ontology. According to this manual annotation process, we were able to evaluate the accuracy of our LLM-based automatic I/O annotation work (i.e., our implemented first method activity for the service annotation phase) in [23]. Further, the results of this process constitute the basis for evaluating our LLM-based automatic action annotation work (i.e., our implemented second method activity for the service annotation phase) over OpenAPI specifications.

To this end, the evaluation of the latter work was conducted by first exploiting it to automatically produce action annotations in the generated OpenAPI specifications and then comparing these annotations with the ground truth ones. Per each OpenAPI specification, we finally recorded in an Excel file which action annotations were correct, which were partially correct (they correspond to an action that strongly correlates with the operation intended semantics), which were totally wrong, and which were actual hallucinations, while we also recorded the number of operations not annotated at all (LLM annotation failure). Via this recording, we were finally able to automatically compute, via Excel formulas, the following evaluation metrics that mainly focus on the accuracy of our LLM-based action annotation work:

- *annotation precision*: examines how precise the suggested annotations are. It is formally computed by dividing the number of correctly annotated operations ($correctMatches$) by the total number of annotated operations ($totalMatches$) as follows: $prec = \frac{correctMatches}{totalMatches} * 100$.
- *relaxed annotation precision*: examines how precise the suggested annotations are with a relaxed interpretation of precision, allowing the use of an action, which does not perfectly match but strongly correlates to the operation's intended semantics. More formally, this metric is computed as follows: $rprec = \frac{correctMatches+relevantMatches}{totalmatches} * 100$ where $relevantMatches$ is the number of relevant but not perfect matches. Please note that when the latter matches are considered, precision increases. Further, through this metric, we attempt to have a discrimination criterion in case some LLMs achieve equivalent annotation precision. Thus, we can discern the best among them based on its ability to attain a higher relaxed precision.
- *annotation recall*: examines the ratio between the number of correctly annotated service operations divided by the total number of service operations. More formally, it is defined as follows:

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

$rec = \frac{correctMatches}{totalOps} * 100$ where $totalOps$ is the total number of operations in the enhanced EMB dataset. In essence, there is a trade-off between precision and recall. So, an LLM is better when it is able to find the best possible balance between these two metrics.

- *relaxed annotation recall*: examines the ratio between the number of properly annotated service operations (i.e., with best and relevant matches) divided by the total number of service operations. More formally, it is defined as follows: $rrec = \frac{correctMatches+relevantMatches}{totalOps} * 100$. We expect that relaxed annotation recall is higher than annotation recall. Combined with relaxed annotation precision can enable exploring which LLM achieves the best possible balance between them.
- *annotation F1*: it is defined as the harmonic mean between annotation precision and recall. More formally: $f1 = \frac{2*prec*rec}{prec+rec}$. This is a better metric than precision and recall as it combines them both into a single accuracy value. Further, by applying the harmonic mean over these two metrics, it punishes their relative imbalance, signifying that it can attain high values only when both have high values, too.
- *relaxed annotation F1*: it is defined similarly to normal annotation F1 as the harmonic mean between relaxed annotation precision and recall: $rf1 = \frac{2*rprec*rrec}{rprec+rrec}$. With this metric, as it calculates a single (composite) accuracy score, we can explore in a better way which LLM achieves the best possible balanced between relaxed precision and recall.
- *operation coverage*: examines the proportion of service operations that have indeed been annotated. It is formally defined as :$opcov = \frac{totalMatches}{totalOps} * 100$. This metric is similar to a recall metric but instead of considering the number of correct matches, it accounts the number of total matches, either correct or not. The rationale of its usage is to explore how many of the service operations are mapped to ontology-based actions. Obviously, the more operations are covered, the better. However, how much better depends on the precision, i.e., whether a greater number of mapped operations is precisely annotated. Again, this metric can be utilised as a discriminator factor between LLMs to select an LLM that has both the highest F1 and operation coverage.
- *hallucination percentage*: this metric computes the percentage of hallucinations within all the matches returned by an LLM across all OpenAPI specifications. It is formally defined as: $hallperc = \frac{hallucinations}{totalMatches} * 100$ where $hallucinations$ is the total number of hallucinations returned by the LLM. A hallucination concerns the proposition by an LLM of a non-existing ontology element for annotating a service operation. Obviously, the lower is the value of this metric, the better. Its use is crucial as it signifies whether annotation precision issues relate to the suggestion of hallucinations. Further, it can be a discrimination criterion between LLMs, as it would be much better to select an LLM that does not hallucinate so much, if not at all.
- *service coverage*: relates to the proportion of services which we were indeed annotated. It is formally computed by dividing the number of outputs produced ($outputs$) with the number of services ($services$) in the (enhanced) EMB dataset as follows: $scov = \frac{outputs}{services} * 100$. This metric enables examining if an LLM can successfully produce the annotation output for each service without exhibiting any errors. Thus, the higher it is, the better. This is again another discriminator factor for LLMs in case they are equivalent in other important metrics, like annotation precision and recall.
- *syntax validity*: as the output from the LLM is a JSON-formatted array, we desire that this output is syntactically valid, such that we can successfully process to proceed with the actual annotation of the OpenAPI specification. As such, we need to explore the proportion of syntactically valid outputs ($validOutputs$) in terms of all outputs ($outputs$) returned by an LLM. More formally, this metric is defined as: $valid = \frac{validOutputs}{outputs} * 100$. Obviously, the higher its value is, the better. Again, this metric can be used as a discriminator factor between LLMs with similar performance on the most crucial metrics.

It must be highlighted that all the above metrics were computed for multiple LLMs and not just one. Thus, the above semi-manual evaluation process was repeated per each LLM considered. The

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

state-of-the-art LLMs that we considered at the moment of the evaluation were: OpenAI's GPT 4.1, Anthropic's Claude Sonnet 3.7 & 4, Deep Seek's V3 & R1 plus Mistral's Large.

Request Structuring and Annotation Evaluation

To evaluate our method realising this phase, we have designed EMBR, a request dataset. This dataset comprises carefully crafted textual functional service requests, which explicitly target the services in the enhanced EMB dataset and their operations.

Based on this EMBR dataset as well as our internal and Schema.org ontologies, we have manually structured and annotated its respective requests. As such, we have constructed a ground truth for evaluating our request structuring and annotation method.

Based on this ground truth, we evaluated our method as follows. Per each considered LLM and per each request in EMBR, we first apply our method's first activity to structure the request, and we mark down in an Excel file whether the correct action, input and output sections were produced for the request. Then, we apply our method's second activity, and we record per request section, how many annotations were correct, how many were relevant, how many were wrong, how many mapped to hallucinations and how many request elements were not covered (in terms of annotations). After all this manual work, we used Excel formulas to compute the performance of our method in the context of each exploited LLM in terms of the following metrics, each targeting either the method's structuring or annotation activity:

- *Action structuring precision*: the precision in correctly determining the right action of the functional service request. It is formally defined as follows: $actStrucPrec = \frac{correctAction}{allActions} * 100$ where $correctAction$ is the number of correct actions suggested and $allActions$ is the number of all suggested actions by the LLM.
- *Input structuring precision*: the precision in correctly determining the right input parameters of the functional service request. It is formally defined as follows: $inputStrucPrec = \frac{correctInputs}{correctInputs+wrongInputs} * 100$ where $correctInputs$ is the number of correct input parameters suggested, and $wrongInputs$ is the number of wrongly suggested input parameters.
- *Output structuring precision*: the precision in correctly determining the right output of the functional service request. It is formally defined as follows: $outputStructPrec = \frac{correctOutputs+noOutputs}{requests} * 100$ where $correctOutputs$ is the number of correctly suggested outputs, $noOutputs$ is the number of times the LLM correctly did not suggest any output, and $requests$ is the total number of requests in the EMBR dataset. Please note that, as a service request could signify the requirement not to create any output (e.g., when creating, updating or deleting a specific entity), a correct behaviour of an LLM is to not suggest an output in this case. This perfectly justifies the nominator in the above formula.
- *Structuring precision*: this is the overall metric covering the global precision in structuring service requests. It is formally defined as follows: $structPrec = \frac{correctActions+correctInputs+correctOutputs+noOutputs}{allActions+correctInputs+wrongInputs+requests} * 100$. As can be seen, we have a division between the sum of all correct section-specific suggestions and the sum of all section-specific suggestions, where the latter sum includes the number of all suggested actions, the number of requests in EMBR for the sake of outputs, as well as the number of correct and wrong input parameters (suggestions).
- *Action annotation precision*: it corresponds to the precision in correctly annotating the action section of a functional service request. It is formally defined as follows: $actionAnnotPrec = \frac{correctActionAnnots}{suggestedActionAnnots} * 100$ where $correctActionAnnots$ is the number of correctly annotated request actions and $suggestedActionAnnots$ is the number of action annotations suggested by an LLM. Please note that we do not consider an annotation as correct if it partially matches the intended service request action. Thus, we account only perfect matches.
- *relaxed action annotation precision*: it is similar to the previous metric, but it also considers as correct annotations those that partially match the request's intended action (i.e., they are strongly correlated with it but do not fully match it). It is formally defined as follows: $relaxedActionAnnotPrec =$

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

$\frac{correctActionAnnots+pCorrectActionAnnots}{suggestedActionAnnots} * 100$ where $pCorrectActionAnnots$ is the number of annotations that partially match the request's intended action.

- *Action annotation recall*: indicates the recall in action annotation. It is formally defined as follows: $actionAnnotRec = \frac{correctActionAnnots}{requests} * 100$.
- *Input annotation precision*: it corresponds to the precision in correctly annotating the input section of a functional service request. It is formally defined as follows: $inputAnnotPrec = \frac{correctInputAnnots}{suggestedInputAnnots} * 100$ where $correctInputAnnots$ is the number of correctly annotated request input parameters and $suggestedInputAnnots$ is the number of input parameter annotations suggested by an LLM. Please note that we do not consider an annotation as correct if it partially matches an intended service input parameter. Thus, we account only perfect matches.
- *Relaxed input annotation precision*: it is similar to the previous metric, but it also considers as correct annotations those that partially match a request's intended input parameter. It is formally defined as follows: $relaxedInputAnnotPrec = \frac{correctInputAnnots+pCorrectInputAnnots}{suggestedInputAnnots} * 100$ where $pCorrectInputAnnots$ is the number of annotations that partially match a request's intended input parameter.
- *Input annotation recall*: indicates the recall in input annotation, i.e., the ability to provide correct annotations for all input parameters of all requests (in EMBR). It is formally defined as follows: $inputAnnotRec = \frac{correctInputAnnots}{correctInputAnnots+pCorrectInputAnnots+wrongInputAnnots+hallInputAnnots+noInputAnnots} * 100$ where $wrongInputAnnots$ is the number of wrong input annotations, $hallInputAnnots$ is the number of hallucinated annotations for input parameters, and $noInputAnnots$ is the number of input parameters with no annotations suggested. Please note that the following holds: $suggestedInputAnnots = correctInputAnnots + pCorrectInputAnnots + wrongInputAnnots + noInputAnnots$. As such, the previous formula could be simplified in terms of its denominator as follows: $inputAnnotRec = \frac{correctInputAnnots}{suggestedInputAnnots+noInputAnnots} * 100$. However, we supplied its complicated form to stress the kinds of input annotation suggestions that can be delivered by an LLM.
- *Output annotation precision*: it corresponds to the precision in correctly annotating the output section of a functional service request. It is formally defined as follows: $outputAnnotPrec = \frac{correctOutputAnnots}{suggestedOutputAnnots} * 100$ where $correctOutputAnnots$ is the number of correctly annotated request outputs and $suggestedOutputAnnots$ is the number of output annotations suggested by an LLM. Please note that we do not consider an annotation as correct if it partially matches the intended service request output. Please also note that the suggested output annotations can include partially correct output annotations, wrong output annotations and hallucinated output annotations.
- *Relaxed output annotation precision*: it is similar to the previous metric, but it also considers as correct annotations those that partially match the request's intended output. It is formally defined as follows: $relaxedOutputAnnotPrec = \frac{correctOutputAnnots+pCorrectOutputAnnots}{suggestedOutputAnnots} * 100$ where $pCorrectOutputAnnots$ is the number of annotations that partially match the request's intended output.
- *Output annotation recall*: indicates the recall in output annotation, i.e., the ability to provide correct output annotations for all requests (in EMBR). It is formally defined as follows: $outputAnnotRec = \frac{correctOutputAnnots}{requests-noOutputs} * 100$ where $noOutputs$ is the number of requests with no outputs. Thus, the formula's denominator signifies the number of requests that do have an output.
- *Annotation precision*: indicates the overall precision in annotating all sections of a functional service request. It is formally defined as follows:

$$annotPrec = \frac{correctActionAnnots + correctInputAnnots + correctOutputAnnots}{suggestedActionAnnots + suggestedInputAnnots + suggestedOutputAnnots} * 100$$

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

- *Annotation recall*: indicates the overall recall in annotating all sections of a functional service request. It is formally defined as follows:

$$annotRec = \frac{correctActionAnnots + correctInputAnnots + correctOutputAnnots}{2*requests - noOutputs + noOutputAnnots + suggestedInputAnnots + noInputAnnots} * 100$$

  This metric covers the division between the sum of all section-specific correct annotations and the sum of twice the number of requests minus the requests with no outputs (this covers all actions and outputs to be annotated), plus the number of suggested input annotations and the number of input parameters not annotated.
- *Annotation F1*: signifies a composite and balanced metric of annotation accuracy, formally computed as the harmonic mean between annotation precision and recall: $\frac{2*annotPrec*annotRec}{annotPrec+annotRec}$.

Finally, it must be highlighted that we have assessed the performance of the same state-of-the-art LLMs as in the case of the evaluation of the LLM-based action annotation of OpenAPI specifications.

Functional Service Discovery Evaluation

Each request in EMBR is associated with all service operations from EMB that match it. This signifies that the ground truth of all EMBR requests is already known, thus facilitating service matching accuracy metrics computation. Each request in EMBR takes the form of a small sentence (which demands the supply of a specific operation), typically including a verb and specific entities to formulate the requested operation's intention. All such sentences have parts that have been named differently (i.e., via synonyms) to not exactly match the words in the description of the EMB services and their operations. This will severely impact TFIDF matchers' accuracy, as they rely on the existence of comment terms between service requests and operations to infer their matching. However, it is not expected to severely impact the accuracy performance of the other matcher families.

EMBR was cautiously separated into two subsets: (a) *single service requests*: comprises requests matching operations of a single service, and (b) *multi-service requests*: comprises requests that match operations from multiple services. This separation was conducted for multiple reasons. First, it covers two distinct cases in (functional) service matching, so it simulates realistic situations. Second, the first subset "presses" matchers over precision, as the return of operations from other services will lower precision, while the second over "recall", as the matchers need to cover the operations from multiple services and not just one. Third, the first subset can be used for the lexical category-based threshold calibration of matchers, like the TFIDF and LLM-based embeddings ones.

EMBR has been incorporated into the article package linked at the end of this manuscript. In the following, we provide some statistics over its content in Table 1.

Table 1. Statistics concerning the EMBR dataset.

EMBR Subset	Request Number	Avg./Min/Max Num of Matching Services	Avg./Min/Max Num of Matching Operations
Single-Service	15	1/1/1	1.75/1/4
Multi-Service	11	2.81/2/7	5.72/3/9

Based on EMBR and its request-to-service operation mappings, we have implemented an automatic evaluation framework, able to assess any of our matchers in terms of their matchmaking accuracy. Initially, the TFIDF and LLM-based embeddings matchers are calibrated based on the EMBR single-service subset. Next, per each matcher, this framework first calculates the raw precision, recall and F1 metrics for each EMBR request and then computes their average value. In addition, we should stress that each evaluation experiment is repeated 10 times per matcher. This means that the accuracy metric values are further averaged across these experiments. Further, these metrics are computed at both the service and operation levels as well as for the whole EMBR dataset and each of its subsets.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

The raw precision and recall metrics are computed by considering the ground truth and the returned results of a matcher according to the current EMBR request. In the context of the returned results, we only account specific categories as matching by considering proper academic practice. For the lexical dimension, the *COMPLETE* and *PARTIAL* categories are considered to incorporate proper matches, while the POSSIBLE category comprises partial matches that are thus ignored. For the semantic dimension, the relevant categories are the *EXACT*, *PLUGIN* and *SUBSUMES*.

6.2. Action Annotation Evaluation

The evaluation results produced when applying our LLM-based Automatic OpenAPI Specification Annotation work at the action level are supplied in Table 2.

Table 2. The evaluation results for our LLM-Based action annotation activity over OpenAPI specifications.

Metric	GPT 4.1	Claude Sonnet 3.7	Claude Sonnet 4	DS V3	DS R1	Mistral Large
annotation precision	95.3%	81.2%	91.2%	95.3%	97.2%	89.9%
annotation recall	95.3%	81.2%	91.2%	95.3%	95.3%	89.9%
annotation F1	95.3%	81.2%	91.2%	95.3%	96.2%	89.9%
relaxed annot. prec.	99.3%	93.2%	95.9%	97.9%	98.6%	97.9%
relaxed annot. recall	99.3%	93.2%	95.9%	97.9%	96.6%	97.9%
relaxed F1	99.3%	93.2%	95.9%	97.9%	97.6%	97.9%
operation coverage	100.0%	100.0%	100.0%	100.0%	97.9%	100.0%
halluc. perc.	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
service coverage	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
syntax validity	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%

By focusing on the most important accuracy metric, i.e., *annotation precision*, we can see that Deep Seek R1 is the best LLM, followed by GPT 4.1 and Deep Seek V3 (nominated as the best also for LLM-based I/O annotation). On the other hand, the worst LLM is Claude Sonnet 3.7, followed by Mistral Large. Overall, we could say that most of the LLMs exhibit an excellent annotation precision performance, surpassing 90%. Such a result can be justified by the fact that the current annotation activity is simpler than the external service I/O annotation one (see the evaluation results in [23] that map to a precision below 60%) and is restrained on a very small portion of just one external ontology. As such, there is no need to modify our work over this activity by, e.g., relying on the automatic construction of an internal function/action ontology.

Concerning *annotation recall*, most LLMs had the same performance as with annotation precision, except for Deep Seek R1. This result is in line with *operation coverage*, where we can observe that most LLMs matched all operations, while Deep Seek R1 failed to attain a perfect score due to the 3 operations that were missed. The current evaluation result is quite plausible and satisfactory, as all LLMs exhibit a metric score above 90%.

By combining annotation precision and recall, the *annotation F1* results are more than expected, as most LLMs had the same scores in both precision and recall. Only, Deep Seek R1 had a smaller score decrease in annotation recall due to the 3 missed operations. However, this did not prevent it from reaching the highest position, winning the annotation accuracy race. In second place, we find again GPT 4.1 and Deep Seek V3. While the worst LLM is once again Claude Sonnet 3.7. The normal annotation accuracy results are well acceptable as they enable us to have the right potential in increasing the accuracy in service discovery. That was actually the main objective, which is well achieved in our opinion.

By checking now the *relaxed annotation precision*, we can certainly see that it is increased for all LLMs. In fact, even the LLMs that were below the 90% bound have now surpassed it. Further, the increase in these LLMs was higher than that of those that already had an acceptable (normal) annotation precision score. This is another plausible evaluation result, highlighting that the current

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

annotation activity can attain high accuracy values and mostly supply either exact or close-matching actions to RESTful service operations.

We should also mark down that the order of the LLMs in terms of *relaxed annotation precision* has changed with respect to normal *annotation precision*. In particular, the best LLM is now GPT 4.1, followed by Deep Seek R1 and then Deep Seek V3 and Mistral Large. In addition, the difference between the first and third place is quite small (1.34%). We outline once again the leap in Mistral Large's performance, where this LLM moved from the second last place to the third one.

Similar results to *relaxed annotation precision* apply for *relaxed annotation recall*, with the sole exception of Deep Seek R1 (which missed 3 operations). This actually penalises Deep Seek R1 such that it moves from the second to the third place. This change not only applies to *relaxed annotation recall* but also *relaxed F1*. In this respect, the best LLM in terms of relaxed accuracy is GPT 4.1, followed by Deep Seek V3 and Mistral Large (now in second place) and then Deep Seek R1. While the worst LLM is still Claude Sonnet 3.7.

By considering the annotation accuracy results, it is difficult to easily nominate a winner. However, in our opinion, GPT 4.1 seems to have precedence as it is second in normal accuracy and first in relaxed accuracy. In the second place, we would put Deep Seek V3 due to its excellent performance as well as the fact that it does not miss any operation. Thus, while being a clear winner in normal accuracy, we would place Deep Seek R1 third as it misses 3 operations and is overtaken by the other two LLMs in relaxed accuracy.

We close the accuracy chapter by indicating that the *hallucination percentage* is 0.0% for all LLMs. This means that our prompt was excellently crafted, assisting the LLMs to always find a proper and valid action within the Schema.org ontology that could be mapped to each examined operation from the OpenAPI specifications processed. As hallucination is a major LLM issue, we are glad that this has been prevented in our current (annotation) activity.

We finalise the analysis by highlighting that all LLMs achieved a perfect performance for both the *service coverage* and *syntax validity* metrics. This means that our implemented activity is able to annotate any OpenAPI specification, regardless of its size, without any issues.

Overall, we are more than satisfied with the achieved results. The best LLMs have an annotation accuracy beyond 90%. This means that our activity has been properly designed and implemented. By also considering that the relaxed accuracy reached 99.3% (i.e., it is almost perfect), we believe that the LLM-based semantic service operation annotation has the potential to facilitate boosting the)(functional) service discovery accuracy.

6.3. Request Structuring and Annotation Evaluation

Table 3 depicts the evaluation results for our implemented method activity in terms of request structuring. As can be seen, all LLMs exhibited an ideal performance in terms of action structuring precision, highlighting that the LLMs obeyed our crystal clear instructions. However, this performance is differentiated in the context of the other two request sections' structuring.

Table 3. The evaluation results for our first method activity in terms of LLM-based request structuring.

Metric	GPT 4.1	Claude Sonnet 3.7	Claude Sonnet 4	DS V3	DS R1	Mistral Large
Action structuring precision	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Input structuring precision	89.28%	96.43%	88.88%	85.18%	92.00%	89.28%
Output structuring precision	100.0%	100.0%	100.0%	92.59%	88.88%	96.29%
Structuring precision	96.34%	98.78%	96.29%	92.59%	93.67%	95.12%

In particular, in terms of input structuring precision, Anthropic's Claude Sonnet 3.7 is the best LLM, followed by Deep Seek's R1, while the worst performance is exhibited by Deep Seek's V3. On the other hand, in terms of output structuring precision, three LLMs are the best, exhibiting an ideal

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

performance; in particular, OpenAI's GPT 4.1 and Anthropic's Claude Sonnet 3.7 & 4, while Deep Seek's R1 is the worst LLM.

Based on this differentiation in performance in terms of input and output request structuring, the best LLM in the context of the overall *structuring precision* is Anthropic's Claude Sonnet 3.7, followed by OpenAI's GPT 4.1 and Anthropic's Claude Sonnet 4. On the other hand, the worst LLM is Deep Seek's V3, followed by Deep Seek's R1. However, in overall, the *structuring precision* is quite satisfactory as it is above 90% in all LLMs.

The latter result signifies that, when budget restrictions apply, one could utilise an open-source LLM like Deep Seek's R1 to conduct the structuring. Otherwise, it is advised to select the best-performing LLM (Claude Sonnet 3.7), as higher structuring precision can further increase the chances of higher annotation accuracy and thus service matchmaking accuracy.

Table 4 depicts the evaluation results of our second method activity, the LLM-based request annotation one.

Table 4. The evaluation results for our second method activity in terms of LLM-based request annotation.

Metric	GPT 4.1	Claude Sonnet 3.7	Claude Sonnet 4	DS V3	DS R1	Mistral Large
Action annotation precision	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Relaxed action annotation precision	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
Action annotation recall	100.0%	100.0%	100.0%	92.59%	100.00%	100.00%
Input annotation precision	95.83%	85.18%	96.15%	82.61%	95.65%	88.46%
Relaxed input annotation precision	95.83%	92.59%	100.00%	86.95%	100.00%	100.00%
Input annotation recall	85.18%	85.18%	96.15%	79.16%	95.65%	88.46%
Output annotation precision	93.75%	79.16%	83.33%	80.00%	85.71%	82.61%
Relaxed output annotation precision	93.75%	87.50%	91.66%	90.00%	95.24%	100.00%
Output annotation recall	62.50%	79.16%	83.33%	76.19%	85.71%	79.16%
Annotation precision	97.01%	88.46%	93.50%	88.40%	94.36%	90.78%
Annotation recall	80.24%	85.18%	90.00%	78.20%	87.01%	86.25%
Annotation F1	81.76%	86.79%	91.72%	81.33%	90.54%	87.89%

Similarly to the case of request action structuring, the performance in terms of request action annotation is ideal for most LLMs with the exception of Deep Seek's V3, which was unable to produce an annotation for just one request action. However, the performance in the context of the other request sections is differentiated between the 6 state-of-the-art LLMs considered.

In the context of request input annotation precision, the best LLM is Anthropic's Claude Sonnet 4, closely followed by OpenAI's GPT 4.1 and Deep Seek's R1. For Sonnet 4 and R1, the performance becomes ideal, if we also consider partially matching input annotations. The same holds for Mistral Large, which, however, lags behind in terms of (full) input annotation precision. On the other hand, the worst LLM is Deep Seek's V3, followed by Anthropic's Claude Sonnet 3.7.

Concerning request input annotation recall, the results are similar to precision, with the exception of OpenAI's GPT4.1 having its performance deteriorated from 95,83% to 85,18%. The order of the best two LLMs is thus as follows: Anthropic's Claude Sonnet 4 is the best, closely followed by Deep Seek's R1. In fact, the recall performance for these LLMs is identical to the precision performance. This clearly indicates that: (a) these two LLMs have the best input annotation performance and (b) their non-ideal but excellent performance is due to partial matches. So, these LLMs do not exhibit wrong input annotations or hallucinated ones, and they do not fail in supplying annotation suggestions for any input parameter. On the other hand, Deep Seek's V3 is the worst LLM, also in input annotation recall, thus exhibiting the worst input annotation performance overall. This translates to one partially correct input annotation, 2 wrong annotations, one hallucinated one and one input parameter where no annotation suggestion was supplied. Further, the deterioration in OpenAI's LLM performance

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

translates to one wrong input annotation and three cases where no annotation suggestions were supplied.

Going further towards request output annotation, the results are slightly different at both the top and the bottom of the LLM classification in the context of output annotation precision. At the classification top, we have now OpenAI's GPT4.1 as the best, followed by Deep Seek's R1 and Anthropic's Claude Sonnet 4. At the bottom, we now have Anthropic's Claude Sonnet 3.7 as the worst LLM, followed by Deep Seek's V3.

When we also consider partial matches, the precision results are further mixed but have been improved in most LLMs. Surprisingly, Mistral Large exhibits ideal performance, followed by Deep Seek's R1 and OpenAI's GPT 4.1. However, OpenAI's GPT4.1 does not change its performance at all (no partial matches). Further, Anthropic's Claude Sonnet 3.7 is still the worst LLM, although its performance has been slightly improved.

On the other hand, the results concerning output annotation recall are similar to those of input annotation precision at the top of LLM classification: Deep Seek's R1 is the best LLM, closely followed by Anthropic's Claude Sonnet 4. Surprisingly, the worst LLM is now OpenAI's GPT4.1, which failed to provide output annotations for 8 requests in EMBR while it supplied one hallucinated output annotation, followed by the usual suspect of Deep Seek's V3 LLM (with only one output annotation missing, but two partially correct annotations and two hallucinated ones).

Overall, we should stress that the output recall performance in general dropped below 90%. This is due to the fact that there are imprecise or missing input annotations for all LLMs. For instance, even the best performing LLM, i.e., Deep Seek's R1, exhibits 2 partial matches and one hallucinated annotation. Further, this LLM failed in 2 cases, not providing any annotation for any section of the respective requests. In this sense, we considered that this was a temporal failure and is not always exhibited by this LLM. As such, we have neglected these two cases from this LLM's evaluation. Of course, the most correct action is to re-evaluate the LLM for these two cases and assess the respective results to update its output annotation performance.

Concerning overall output annotation accuracy, we consider that the best LLM is Deep Seek's R1, followed by Anthropic's Claude Sonnet 4. Thus, we have almost the same overall result as in the case of input annotation. On the other hand, the order at the bottom of the LLM classification has been modified, with OpenAI's GPT4.1 being the worst LLM, followed by Deep Seek's V3 and Anthropic's Claude Sonnet 3.7. This classification has been produced by silently considering the F1 measure for output annotation, applied on the precision and recall values of the considered LLMs. To be noted that the gap between the best and worst performing LLMs has been closed in output annotation with respect to that exhibited in input annotation.

Focusing now on the big picture of request annotation, we can observe that best LLM in terms of annotation precision is OpenAI's GPT4.1, followed by Deep Seek's R1 and Anthropic's Claude Sonnet 4. On the other hand, the worst LLMs are Deep Seek's V3 and Anthropic's Claude Sonnet 3.7.

Concerning overall annotation recall, the top in LLM classification has changed: the best LLM is Anthropic's Claude Sonnet 4, followed by Deep Seek's R1 and Mistral Large. On the other hand, the worst LLM is Deep Seek's V3, followed by OpenAI's GPT4.1.

By combining now annotation precision and recall in the context of the (annotation) F1 metric, we can discern Anthropic's Claude Sonnet 4 as the best LLM, followed by Deep Seek's R1. Both of these LLMs surpass the set threshold of 90%, so their performance can be considered quite satisfactory. On the other hand, the worst LLM is Deep Seek's V3, followed by Anthropic's Claude Sonnet 3.7. Thus, these two LLMs must be avoided in terms of request annotation.

6.4. Functional Service Discovery Evaluation

In the following, we supply the evaluation results of our matchers in separate subsections, each dedicated to a different matcher family. The evaluation focus is mainly over service operation matching accuracy and not service matching accuracy as service matching is a side-effect of service operation matching in our case.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

6.4.1. TFIDF Matchers

As TFIDF and LLM-based embeddings matchers are calibrated based on the first EMBR subset, it does not make sense to show their evaluation over this subset. The same holds for the whole EMBR dataset, as the number of requests in the first EMBR subset is greater than that in the second, and the evaluation results for that subset are generated under "ideal" conditions. In this sense, they will bias the final evaluation results for the overall EMBR dataset. Thus, the focus will be solely on the evaluation results concerning the second EMBR subset. On the other hand, the LLM-based matchers are not calibrated, so we will present and analyse their evaluation results over both the EMBR dataset and its two subsets.

Table 5 depicts the evaluation results for the TFIDF matcher family.

Table 5. The service matchmaking accuracy evaluation results for the TFIDF matchers concerning the EMBR multi-service subset.

Metric	Core TFIDF	Enhanced Core TFIDF	Structured TFIDF	Enhanced Structured TFIDF	Ontology-based TFIDF	Enhanced Ontology-based TFIDF
Avg. operation precision	0.31	0.25	0.41	0.47	0.44	0.32
Avg. operation recall	0.09	0.28	0.25	0.17	0.21	0.44
Avg. operation F1	0.1	0.19	0.18	0.13	0.19	0.32

As can be seen, we can first observe that the paraphrasing-based variants of most of the TFIDF matchers lead to a better matchmaking accuracy in terms of the F1 metric, with the exception of the *Enhanced Structured TFIDF* matcher. This is due to the fact that the paraphrasing variants tend to fix terminological issues in the original request, so they increase the probability of matching. Indeed, by observing the recall metric, we can see that recall is greatly increased, thus more correct results are returned. Further, while precision is obviously dropped due to the existence of wrong matches, this decrease is not significant, leading to a final increase in the F1 score.

Please note that while precision decrease is controlled by the *Ontology-based TFIDF* matcher and its variant, this is not the case for the *Core TFIDF* matcher and its variant. Thus, we are surprised that the precision did not sharply decrease for this latter matcher in the context of its variant. Maybe this could be justified by the fact that the core matcher does not return results for some queries. While its paraphrasing variant does lead to the production of results for many of these queries. In this respect, this variant's precision more or less conforms to the precision pattern of its core base matcher.

The exception of lower F1 score for the paraphrasing variant of the *Enhanced Structured TFIDF* matcher is due to the fact that recall drops instead of getting increased. Maybe this could be justified by the probability of introducing a wrong parameter via the LLM-based paraphrasing of the original request. This introduction can then influence the similarity in the input or output sections in the paraphrased requests, thus leading to lower recall. While this introduction does not severely impact the other family matchers as they are not section-sensitive.

By focusing on operation matching precision, we can observe that the *Enhanced Structured TFIDF matcher* is the winner, reaching a value of 0.47. This matcher is proceeded by the *Ontology-based TFIDF* and the *Structured TFIDF* matchers. This highlights the fact that the *Structured TFIDF* matcher and its variant have a good precision level, even if they do not apply ontology-based action filtering. This can be justified by the fact that the section-sensitivity of this matcher enables it to identify section incompatibilities between service operations and requests, thus enabling it to attain good precision levels. On the contrary, the lack of any kind of section-based filtering in the *Core TFIDF* matcher and its variant leads to the least possible precision values, as expected.

Concerning recall, we can observe that the best matcher is *Enhanced Ontology-based TFIDF*, followed by *Enhanced Core TFIDF*. This is inline with our previous justification of why the paraphrasing matcher variants got higher recall values. As expected, the use of both ontological annotations and

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

paraphrased request versions enables to increase further the probability of common terms in service requests and operations, thus leading to higher recall in the case of the *Enhanced Ontology-based TFIDF* matcher. On the other hand, the base matcher variants and especially the *Core TFIDF matcher* lead to low recall, something well expected based on the way the EMBR dataset has been constructed (requests have different terms than operations).

Based on the above analysis, it comes as no surprise that the best matcher in terms of F1 (overall accuracy) is the *Enhanced Ontology-based TFIDF*, followed by *Ontology-based TFIDF* and *Enhanced Core TFIDF* matchers. This well highlights the main merits of our research methodology, which leads to producing semantically rich service specifications and requests. Such merits are well exploited by our hybrid TFIDF matcher and its variant, thus leading to a better service matchmaking accuracy (increase of 0.2 in terms of F1 with respect to the worst value featured by the *Core TFIDF* matcher).

6.4.2. LLM-based Embeddings Matchers

Table 6 depicts the evaluation results for the LLM-based embeddings matcher family.

Table 6. The service matchmaking accuracy evaluation results for the LLM-based embeddings matchers concerning the EMBR multi-service subset.

Metric	Core LLM-based Embeddings	Enhanced Core LLM-based Embeddings	Onto LLM-based Embeddings	Enhanced Onto LLM-based Embeddings	Structured LLM-based Embeddings	Enhanced Structured LLM-based Embeddings
Avg. operation precision	0.45	0.55	0.46	0.54	0.6	0.48
Avg. operation recall	0.13	0.50	0.14	0.43	0.35	0.45
Avg. operation F1	0.13	0.40	0.13	0.34	0.25	0.33

As can be seen, the overall pattern of increased recall in the paraphrasing variants of the matchers also holds for this family. However, now we also observe that this pattern also transits to the precision level for two out of the three of such variants. This highlights that the LLM-based embeddings matchers are more robust to request paraphrasing and tend to better extract the intention of the paraphrased requests (which should be equal to the original's one). These two patterns absolutely justify the overall F1 increase in the paraphrasing variants of this family's matchers. In fact, F1 is more than tripled in the case of the core LLM-based embeddings matcher and almost tripled for the ontology-based LLM-based embeddings matcher.

By still focusing on F1, we can see that the *Enhanced Core LLM-based Embeddings* matcher reaches the highest F1 score, which is higher than that of the best TFIDF matcher, which happens to be hybrid. This could signify alone that potentially LLM-based embeddings matching does not require so much ontology-based annotations when equipped with request paraphrasing. However, it could also indicate that potentially there is much room for improvement for the more sophisticated hybrid matchers in the current family. For instance, the semi-semantic request vectors could be enhanced to become normal sentences to allow a better capturing of the request's intention. While apart from this improvement, there is a need to better explore the configuration space for the *Structured LLM-based Embeddings* matcher. As currently, we have examined only its default configuration. It must be also noted that we present the evaluation results for this matcher and its variant only for the first input matching strategy as the results for the other strategies are quite similar.

Overall, the paraphrasing variants of all matchers in this family have better F1 scores than all TFIDF matchers. This is a much expected result as LLM-based matching is considered as a better technique than TFIDF matching. However, the difference in F1 scores is not very significant, which correlates to the need to further improve our hybrid matchers in this family.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

By focusing now on operation matching precision, we can observe that it reaches moderate levels (at most 0.6) with *Structured LLM-based Embeddings* matcher as the best, followed by the paraphrasing variants of the core and onto LLM-based embeddings matchers. This can be considered a well-expected result as the *Structured LLM-based Embeddings* matcher conducts both action-based filtering and section-focused matching. It is also remarkable that the good precision value for this matcher correlates with its ability to provide matching results for almost all requests in the 2nd EMBR subset. On the other hand, the *Enhanced Onto LLM-based Embeddings* matcher fails to provide a matching result in half of the requests in this subset.

Concerning operation matching recall, we observe that it also reaches moderate levels (at most 0.5) with the top three matchers being the three paraphrasing variants of the core, structured and ontology-based matchers in this order. However, compared to precision, the recall results seem to be lower, something expected by considering that the 2nd EMBR subset puts more pressure over recall. The low recall values in the basic variants of the matchers is due to the fact that in some queries no results are supplied. This is more evident in the core and onto-based matcher, while this situation is better in the structured matcher. Potentially, this superiority of the structured matcher in this case is due to its section-oriented focus, which enables to calibrate similarity losses in one section over gains in others.

6.4.3. LLM-based Matchers

Our framework can evaluate our LLM-based matcher, regardless the LLM it exploits. As such, we were able to assess its individual accuracy performance across many state-of-the-art LLMs, actually those involved also in the evaluation of LLM-based action annotation and LLM-based request structuring and annotation. Table 7 depicts the respective evaluation results achieved, which we will now analyze.

Table 7. The service matchmaking accuracy evaluation results for the LLM-based matchers concerning the EMBR dataset.

Metric	GPT 4.1	Sonnet 3.7	Sonnet 4	DS V3	DS R1	Mistral Large
1st EMBR Subset – Single-Service Requests						
Avg. operation precision	0.76	0.77	0.85	0.9	0.83	0.83
Avg. operation recall	1	1	1	0.93	0.86	0.93
Avg. operation F1	0.84	0.85	0.89	0.91	0.84	0.86
2nd EMBR Subset – Multi-Service Requests						
Avg. operation precision	0.86	0.89	0.91	0.81	0.95	0.86
Avg. operation recall	0.68	0.83	0.91	0.68	0.60	0.70
Avg. operation F1	0.67	0.83	0.89	0.71	0.69	0.72
Overall EMBR Set						
Avg. operation precision	0.77	0.79	0.84	0.83	0.85	0.81
Avg. operation recall	0.86	0.93	0.96	0.83	0.75	0.83
Avg. operation F1	0.77	0.84	0.89	0.82	0.78	0.80

Before delving into the evaluation results analysis, we must stress that it is more than apparent that the LLM-based matchers are beyond the competition, as their accuracy reaches high levels. In some cases, these levels are ideal, especially in terms of recall. This signifies that the construction of LLM-based matchers constitutes the future in functional service matching. Further, this answers our second research question, highlighting that a prompt-based LLM service matcher is the best technique to use to reach high service matching accuracy levels. Further, this technique, with the exception of the TFIDF matcher family, leads to better time-based performance, based on our theoretical complexity analysis by also accounting that it does not require request structuring and annotation. However, we acknowledge that this needs to be proven also at the experimental level. As such, the LLM

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026

doi:10.20944/preprints202606.0323.v1

(prompt-based) service matching achieves the best possible performance-to-accuracy trade-off. Further, by considering our research methodology, the LLM-based matchers do not require incorporating ontological annotations. Thus, this can simplify the configuration of our service registry and can lead to reduced service registration/publication times, further improving our registry's performance.

1st Subset - Single-Service Requests

One major result that can be observed is that the recall for this subset reaches ideal levels. It is actually 1.0 in the context of 3 LLM, while it is close to 1.0 for 2 other LLMs. Thus, most LLMs achieve a close-to-ideal recall, with the exception of Deep Seek's R1.

On the other hand, precision is clearly lower and below 0.9 in most of the LLMs with the exception of Deep Seek's V3 reaching this threshold. This is an expected result by considering the design feature of the 1st subset: it puts pressure on precision as each request maps to just one service and has very vew operation matching results. Further, we also observe that well-known LLMs like OpenAI's GPT 4.1 and Anthropic's Claude Sonnet 3.7 lead to low precision values, going down to 0.76 in the case of GPT 4.1.

Concerning the overall accuracy, we observe Deep Seek's V3 as the best LLM, followed by Anthropic's Claude Sonnet 4, while the worst LLMs are Deep Seek's R1 & OpenAIs' GPT 4.1. The top-level order is justified as the top two LLMs are the best both in terms of precision and recall. With Deep Seek V3 achieving a better balance between these latter two metrics. On the other hand, the bottom-level order is justified by the bad recall performance, especially in case of GPT 4.1. While Deep Seek R1 has a more balanced performance in terms of precision and recall, but always below 0.9 and close to 0.85.

2nd Subset - Multi-Service Requests

By focusing first on precision, we can observe that the best LLM is Deep Seek's R1, followed by Anthropic's Claude Sonnet 4, while the worst LLMs are Deep Seek's V3 and Mistral's Large. Overall, the performance of Deep Seek R1 is extraordinary, reaching 0.95.

Concerning recall, we see that while Anthropic's Claude Sonnet 4 is the best LLM with a high recall value (0.91), the other LLMs have lower performance, which drops in the case of the worst LLM (Deep Seek's R1) to the moderate value of 0.60. This highlights the design feature of the second EMBR subset, which puts pressure over recall as there are multiple operations matching a request offered by multiple services.

Based on the above analysis, it is apparent that Anthropic's Claude Sonnet 4 is the best LLM in terms of overall accuracy. This is supported by the evaluation results on F1 as the F1 score for this LLM is 0.89. Further, the accuracy performance for this LLM is balanced (in terms of precision and recall). On the other hand, the worst LLM is OpenAI's GPT 4.1 with an F1 score of 0.67. This is really not expected from this high-class LLM but it is well justified by its bad performance on recall.

Overall Evaluation Results

By focusing first on precision, we can observe that best LLM is Deep Seek's R1, followed by Anthropic's Claude Sonnet 4, while the worst LLM is OpenAI's GPT 4.1, followed by Anthropic's Claude Sonnet 3.7. In this respect, the worst results seem to be in line with those for the 1st subset, while the best results seem to be aligned with those for the 2nd subset. In overall, the precision performance is below 0.86 for the whole EMBR dataset, which again conforms to the design feature of the first subset, which is greater in size that the second.

Concerning recall, we observe that the best LLM is Anthropic's Claude Sonnet 4, followed by Anthropic's Claude Sonnet 3.7, reaching very high, almost ideal recall levels. This is similar to the case of the 1st subset. However, compared to that subset performance, we can observe that all LLMs have a drop in recall, but, fortunately, this drop is smaller in the case of the Anthropic's LLMs. At the last places in the LLM ordering for this metric, the Deep Seek's LLMs and Mistral's Large can be

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

placed. This ordering seems again to be similar with that in the 1st subset, which is rational based on the greater size of this subset with respect to the other.

Overall, by focusing now on accurary and the F1 metric, we can see that the best LLM is Anthropic's Claude Sonnet 4, followed by Anthropic's Claude Sonnet 3.7. While the worst LLMs are OpenAPI's GPT 4.1 and Deep Seek's R1. Suprisingly, this ordering at the top and bottom reflects the one in the second and not the first EMBR subset.

Globally, the best LLM is clearly Anthropic's Claude Sonnet 4; this is reflected in the results concerning both the 2nd subset and the overall EMBR subset. At the second place, we can pick up different LLMs, depending on the respective case: when single-service requests are issued, we would select Deep Seek's V3; otherwise, in the context of multi-service requests, we would select Anthropic's Claude Sonnet 3.7.

6.5. Discussion

To conclude, the LLM-based matcher has a much better accuracy performance than that of the matchers in all other families, irrespectively of the LLM that it employs. This indicates that we can clearly answer our second research question by signifying that pure LLM-based service matching is the best technique to support functional service discovery as it leads to high accuracy results. Further, its time-based performance is good, potentially better than most of the other matchers, especially as it does not require the preprocessing of service requests, and the computation of LLM-based embeddings vectors via calls to external LLMs. Finally, this functional service discovery technique comes with extra features, which are not exhibited by other matchers in the literature, such as the explainability of the service (operation) matches, the classification of matches in both lexical and semantic categories, the supply of overall and component similarity values unveiling the contribution degree of the latter to the former, and the supply of hierarchically ordered service matchmaking results. As such, we can safely infer that this matcher advances the state-of-the-art in functional service discovery.

Concerning the first research question, we would like to provide the following core statements:

- Our research methodology increases the automation degree in service publication and discovery by incorporating specific LLM-based methods and their encompassing techniques.
- Service publication automation is increased as OpenAPI specifications can be automatically generated from the RESTful services' source code, while they are automatically annotated via the automatic production and use of internal ontologies and the complementary use of external ontologies, like Schema.org.
- Service discovery automation is increased via the automatic structuring and annotation of service requests, which can then be exploited by any service matching algorithm.
- The accuracy of all our methods has been experimentally validated both in our previous published work [21,23] and in the current article (in this section). The evaluation results signify that the automation degree achieved in the context of our research methodology and its implemented LLM-based methods does not sacrifice accuracy. On the contrary, the accuracy in all of the method activities or tasks is very high, making them suitable for use so as to increase (functional) service discovery accuracy.
- The main benefits of our research methodology can be observed by the experimental evaluation of our implemented matchers. As has been derived, there is an increase in service discovery accuracy when utilising the incorporated semantics in service specifications and requests.
- All the above prove the suitability and added-value of our research methodology, which surely increases the automation degree in both service discovery and publication while leads to an increase in service discovery accuracy.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

7. Conclusions & Future Work

7.1. Conclusions

This article has presented a novel research methodology, aiming to provide significant automation support to both RESTful service discovery and publication. This methodology has been realised via the use of innovative LLM-based methods, which enable to automatically produce OpenAPI specifications from the services source code as well as to annotate their operations based on both their I/O and action sections. Further, these methods enable to automatically structure and annotate service requests to transform them to the same semantic space with respect to the annotated service specifications as well as to match them based on different families of matchers. Finally, this methodology has been implemented in a semantic service registry, which incorporates all of its realised methods, and provides the facilities of service publication and discovery via the supply of a specific RESTful API.

In the context of functional service discovery, different matchmaking algorithms (a.k.a. matchers) have been implemented, classified under different families based on the core matchmaking technique that they employ. Many of them are hybrid as they can exploit the semantic annotations implanted in both service specifications and requests. We highlight that two of our matchers are innovative: (a) the *Structured LLM-Based Embeddings* matcher computes the similarity between service operations and requests in multiple levels (section-specific and global) and according to both the lexical and semantic aspects, while its paraphrasing variant enables to produce different versions of the same request to bypass request vagueness and incorrect ontology annotation issues; (b) the *LLM-Based* matcher can be configured to exploit any LLM to support functional service discovery, encompassing innovative features, such as service match explainability and classification under different lexical and semantic categories.

The evaluation of all our implemented matchers highlights that while request paraphrasing and the exploitation of ontology-based annotations increase service discovery accuracy in matchers that individually exploit conventional/classical matching techniques or their combinations, the LLM-based matcher can attain high accuracy levels without even exploiting these two additional capabilities offered by our research methodology. Thus, it seems that the future in functional service discovery goes to the exploitation of pure LLM service matching techniques. Provided that they rely on semantically rich information (i.e., precise textual descriptions) in the context of the service specifications that they are processing.

Our current and previous [21,23] experimental evaluation shows that the methods realising our methodology not only increase the automation degree in service publication and discovery but also achieve this without the expense of accuracy. In fact, the accuracy of these methods is high enough, something that benefits service discovery. Indeed, by observing the service matchmaking results attained by our hybrid matchers, i.e., those that exploit semantics, indicate that the incorporated annotations are precise enough to accelerate service discovery accuracy.

7.2. Future Work

We consider that our work is a first step towards our vision for a global, semantic service registry that is powered by LLMs. To further realise this vision, we plan to follow certain research directions. First, we intend to expand the evaluation of our research methodology and implemented matchers with the new EMB dataset version, which is larger, as well as with different datasets that we might construct from external sources. The evaluation is also planned to cover the time and scalability dimension so as to examine the time-based behaviour of our service matchers and registry. Second, we will devise new semantic-based service matchmaking algorithms that exploit AIO (Action-Input-Output) semantics to increase service discovery accuracy. We will also check whether different core matchmaking techniques can be combined to construct more powerful matchers. It remains to be seen whether these matchers will be able to reach the performance of the LLM-based matcher. Third, we will explore the extension of our methodology and registry towards supporting non-functional service publication and discovery. Fourth, we would like to go beyond RESTful services to support SOAP services as well as serverless

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026 doi:10.20944/preprints202606.0323.v1

functions (which can take the form of a RESTful service operation). Finally, we plan to move towards service composition in order to be able to answer service requests that cannot be fully matched by any of the existing registered services and their operations.

Supplementary Materials: All prompts utilised in our research methodology, the request paraphrasing and the *LLM-based Matcher*, as well as the EMBR dataset can be downloaded at: https://drive.google.com/file/d/1x0fZF-BtBKl33EC7eWj4Q-WACVTHhGVq/view?usp=sharing.

Author Contributions: A.S conducted the study, including conceptualization, methodology, software, validation, data curation, formal analysis, investigation, validation, visualization and writing, in the context of a PhD thesis. K.K supervised the research, contributed to the study design, and critically reviewed and edited the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All prompts utilised in our research methodology, the request paraphrasing and the *LLM-based Matcher*, as well as the EMBR dataset can be downloaded at: https://drive.google.com/file/d/1x0fZF-BtBKl33EC7eWj4Q-WACVTHhGVq/view?usp=sharing. The EMB dataset can be found in https://github.com/EMResearch/EMB.

Acknowledgments: This study was conducted as part of an PhD thesis completed by the first author and supervised by the second.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Georgakopoulos, D.; Papazoglou, M.P. *Service-Oriented Computing*; Cooperative Information Systems, MIT Press, 2008.
2. OASIS. UDDI Version 3.0.2. Standard, OASIS, 2004.
3. Dong, X.; Halevy, A.; Madhavan, J.; Nemes, E.; Zhang, J. Similarity search for web services. In Proceedings of the VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases, Toronto, Canada, 2004; pp. 372–383.
4. Rodriguez, J.M.; Zunino, A.; Mateos, C.; Segura, F.O.; Rodriguez, E. Improving REST Service Discovery with Unsupervised Learning Techniques. In Proceedings of the 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, Santa Catarina, Brazil, July 2015; pp. 97–104. https://doi.org/10.1109/CISIS.2015.14.
5. Lo, W.; Yin, J.; Wu, Z. Accelerated Sparse Learning on Tag Annotation for Web Service Discovery. In Proceedings of the 2015 IEEE International Conference on Web Services, New York, NY, USA, June 2015; pp. 265–272. https://doi.org/10.1109/ICWS.2015.44.
6. Zeng, K.; Paik, I. Semantic Service Clustering With Lightweight BERT-Based Service Embedding Using Invocation Sequences. *IEEE Access* **2021**, *9*, 54298–54309. https://doi.org/10.1109/ACCESS.2021.3069509.
7. Yang, Y.; Qamar, N.; Liu, P.; Grolinger, K.; Wang, W.; Li, Z.; Liao, Z. ServeNet: A Deep Neural Network for Web Services Classification. In Proceedings of the 2020 IEEE International Conference on Web Services (ICWS), Beijing, China, October 2020; pp. 168–175. https://doi.org/10.1109/ICWS49710.2020.00029.
8. Baryannis, G.; Kritikos, K.; Plexousakis, D. A specification-based QoS-aware design framework for service-based applications. *Service Oriented Computing and Applications* **2017**, *11*, 301–314. https://doi.org/10.1007/s11761-017-0210-4.
9. Bener, A.B.; Ozadali, V.; Ilhan, E.S. Semantic matchmaker with precondition and effect matching using SWRL. *Expert Systems with Applications* **2009**, *36*, 9371–9377. https://doi.org/10.1016/j.eswa.2009.01.010.
10. Plebani, P.; Pernici, B. URBE: Web Service Retrieval Based on Similarity Evaluation. *IEEE Transactions on Knowledge and Data Engineering* **2009**, *21*, 1629–1642.
11. Klusch, M.; Fries, B.; Sycara, K. Automated semantic web service discovery with OWLS-MX. In Proceedings of the AAMAS, Hakodate, Japan, 2006; pp. 915–922. https://doi.org/10.1145/1160633.1160796.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2026

doi:10.20944/preprints202606.0323.v1

12. Hogan, A. The Semantic Web: Two decades on. *Semantic Web* **2020**, *11*, 169–185. https://doi.org/10.3233/SW-190387.
13. Fan, A.; Gokkaya, B.; Harman, M.; Lyubarskiy, M.; Sengupta, S.; Yoo, S.; Zhang, J.M. Large Language Models for Software Engineering: Survey and Open Problems. In Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), Melbourne, Australia, May 2023; pp. 31–53. https://doi.org/10.1109/ICSE-FoSE59343.2023.00008.
14. Hou, X.; Zhao, Y.; Liu, Y.; Yang, Z.; Wang, K.; Li, L.; Luo, X.; Lo, D.; Grundy, J.; Wang, H. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* **2024**, *33*, 1–79. https://doi.org/10.1145/3695988.
15. Antonios Smardas. Semantic Service Discovery Supported by LLMs. PhD Thesis, University of the Aegean, 2026. PhD Thesis.
16. Arcuri, A. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology* **2019**, *28*, 1–37. https://doi.org/10.1145/3293455.
17. J. Obidallah, W.; Raahemi, B.; Rashideh, W. Multi-Layer Web Services Discovery Using Word Embedding and Clustering Techniques. *Data* **2022**, *7*, 57. https://doi.org/10.3390/data7050057.
18. Liu, F.; Deng, D.; Jiang, J.; Tang, Q. Event-Driven Semantic Service Discovery Based on Word Embeddings. *IEEE Access* **2018**, *6*, 61030–61038. https://doi.org/10.1109/ACCESS.2018.2876029.
19. Nabli, H.; Ben Djemaa, R.; Ben Amor, I.A. Efficient cloud service discovery approach based on LDA topic modeling. *Journal of Systems and Software* **2018**, *146*, 233–248. https://doi.org/10.1016/j.jss.2018.09.069.
20. OpenAPI Initiative. OpenAPI Specification v3.1.0. Standard, Linux Foundation, 2021.
21. Smardas, A.; Kritikos, K. Towards the Automatic Production of OpenAPI Specifications from Source Code. In Proceedings of the 2025 12th International Conference on Future Internet of Things and Cloud (FiCloud), Istanbul, Turkiye, August 2025; pp. 342–349. https://doi.org/10.1109/FiCloud66139.2025.00053.
22. White, J.; Fu, Q.; Hays, S.; Sandborn, M.; Olea, C.; Gilbert, H.; Elnashar, A.; Spencer-Smith, J.; Schmidt, D.C. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT, 2023. Version Number: 1, https://doi.org/10.48550/ARXIV.2302.11382.
23. Smardas, A.; Kritikos, K. Towards LLM-Assisted Automatic Semantic Annotation for RESTful Services. In *14th International Conference on Emerging Internet, Data & Web Technologies (EIDWT-2026)*; Springer Nature Switzerland: Cham, 2026. Series Title: Lecture Notes in Data Engineering and Communication Technologies (LNDECT).
24. Mainas, N.; Bouraimis, F.; Karavisileiou, A.; Petrakis, E.G.M. Annotated OpenAPI Descriptions and Ontology for REST Services. *International Journal on Artificial Intelligence Tools* **2023**, *32*, 2350017. https://doi.org/10.1142/S0218213023500173.
25. Saati, T. *The Analytic Hierarchy Process*; McGraw-Hill, 1980.
26. Liu, M.; Tu, Z.; Zhu, Y.; Xu, X.; Wang, Z.; Sheng, Q.Z. Data correction and evolution analysis of the ProgrammableWeb service ecosystem. *Journal of Systems and Software* **2021**, *182*, 111066. https://doi.org/10.1016/j.jss.2021.111066.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.