Cluster Computing
https://doi.org/10.1007/s10586-023-04237-x

RESTBERTa: a Transformer-based question answering approach for semantic search in Web API documentation

Sebastian Kotstein[1] · Christian Decker[1]

Received: 15 September 2023 / Revised: 12 November 2023 / Accepted: 11 December 2023

Abstract
To enable machines to process state-of-practice Web API documentation, we propose a Transformer model for the generic task of identifying a Web API element within a syntax structure that matches a natural language query. We solve this semantic-search task with Transformer-based question answering and demonstrate the applicability of our approach to two different tasks, namely the discovery of endpoints and the identification of parameters in payload schemas. With samples from 2321 OpenAPI documentation, we prepare different datasets and fine-tune pre-trained BERT models to these two tasks. We evaluate the generalizability and the robustness of our fine-tuned models. We achieve accuracies of 81.95% for the parameter-matching and 88.44% for the endpoint-discovery task.

Keywords Web API documentation · Semantic search · Endpoint discovery · Parameter matching · Question answering · BERT

1 Introduction

Protocols and formats like the Hypertext Transfer Protocol (HTTP), Uniform Resource Identifier (URI), and JavaScript Object Notation (JSON) are the technical foundation for realizing modern Web Application Programming Interfaces (APIs) [1]. With these protocols and formats, Web API developers can rely on standardized concepts with well-defined semantics, like HTTP verbs and status codes, but also extend their Web APIs with individual URIs, data schemas, and parameters [2]. This maximizes flexibility when designing Web APIs and allows developers to abstract nearly every application through suitable interfaces. However, it also complicates the integration of Web APIs from a client perspective. Integrators, regardless of whether they are human or machine agents, are confronted with different Web API designs with individual syntax and semantics [3]. For a successful integration, they must understand these application-specific syntax and semantics [4].

Web API documentation are a common way to describe interface characteristics, especially to share syntax and semantic descriptions with integrators. The syntax can be described in structured specifications and schemas, which are, to some extent, understandable for both humans and machines. However, it remains a challenge to express semantics in a structured way so that machines can reason from descriptions.

At the beginning of the 21st century, automated service composition was a promising field of research [5], and researchers proposed numerous rich description formats and language models to describe the semantics of Web APIs, mainly WSDL/SOAP-based Web services, in a structured and machine-understandable way. However, these proposed formats and models, like OWL-S [6], WSMO [7], and SA-WSDL [8], were not widely accepted in practice, because of their complexity and the required expert knowledge for their creation [9, 10].

Instead, more recent formats like OpenAPI, WADL, RAML, and API Blueprint became state-of-practice for documenting Web APIs [11] as they are easy to create and distribute in the form of online and offline documents. They are human- and machine-readable since they rely on

✉ Sebastian Kotstein
sebastian.kotstein@reutlingen-university.de

Christian Decker
christian.decker@reutlingen-university.de

[1] Herman Hollerith Zentrum, Reutlingen University, Danziger Straße 6, Böblingen 71034, Baden-Württemberg, Germany

Published online: 31 January 2024

structured formats such as JSON, YAML, and XML. However, only the syntax of a Web API and its elements is expressed through standardized key-value pairs in a structured and machine-understandable way. To explain the meaning and purpose, i.e., the semantics, of Web API elements, descriptions are embedded as natural language (NL) text snippets into the documentation but target human readers and not machines.

The shift from those rich description formats of the automated service composition domain toward these more lightweight formats suggests that providers have cut their efforts to make Web APIs machine-consumable, or at least to make their semantics machine-understandable. Instead of creating complex descriptions for machines, they offer more user-friendly formats with descriptions in NL, which are easy to create and address human readers.

The latest success in the field of natural language processing (NLP), however, leads us to believe that, by using NLP techniques, machines shall be able to understand current state-of-practice Web API documentation, even though they consist of structured syntax and unstructured NL descriptions for semantics. The ubiquitous Transformer architecture has proven to be effective and reliable on a variety of NLP tasks, e.g., text classification [12] and summarization [13], but is also applicable to other language domains, like programming languages [14]. In our previous work [15], we demonstrated that a Transformer model can support machines in reading and processing state-of-practice Web API documentation, consisting of structured syntax and unstructured NL descriptions for semantics. For our first prototype, we focused on a particular Web API integration task requiring the processing of Web API documentation, namely the matchmaking of output and input parameters [9, 10]. We defined this task as follows: Given an NL description of an output parameter and a documented syntax schema containing multiple, hierarchically organized input parameters, the model should choose the input parameter from the schema that matches the NL description best.

Technically, we fine-tuned a pre-trained CodeBERT [14] encoder model to the downstream task of question answering. Question answering, in a broader sense, is the task of finding an answer to a given question in a given paragraph. We formulated the original question-answering problem as a multiple-choice task to align it with the parameter-matching task: Given a semantic NL description of an output parameter, which is the question, and the syntax of the input schema, which is the paragraph, the model should choose the input parameter, i.e., the answer, in the schema that matches the description best.

During research and implementation, we noticed that this Transformer-based question-answering approach is not limited to parameter matching. It might also apply to other integration tasks requiring the processing, more precisely, the identification and extraction of syntax Web API elements from Web API documentation. For example, analogous to parameter matching, we can also model the integration task of endpoint discovery as a question-answering downstream task: Given an NL query describing the expected behavior of a target endpoint, the model should choose an endpoint from a list of endpoints that matches the NL query best.

This article is an extension of our previous work titled "Semantic Parameter Matching in Web APIs with Transformer-based Question Answering" [15]. In this article, we extend the approach not only to the integration task of endpoint discovery but also reconceptualize the entire approach by describing and investigating it from a more generic, i.e., task-independent, perspective. With this work, we want to establish an approach that serves as a framework for any Web API integration task relying on semantic search in Web API documentation, which is, to our understanding, the identification and extraction of a syntax Web API element matching a given NL query.

This article covers the concepts of fine-tuning a pre-trained BERT model to a specific Web API integration task, evaluating the performance of the fine-tuned model, and utilizing it as a search engine afterward. This includes the processing steps for the preparation of task-specific samples used for fine-tuning and evaluation, the conversion, i.e., tokenization, of samples into a BERT-compatible input format, and the interpretation of the model's output to obtain a chosen Web API element. Note that we developed these steps originally for the parameter-matching task and presented them in our previous work. Nonetheless, the steps for tokenization and output interpretation were mostly generic, and we could adopt them without significant modifications. The data preparation process, however, is individual for each integration task. In the course of this article, we have extended this process to prepare task-specific samples not only for parameter matching but also for endpoint discovery. Additionally, we have collected several metrics providing deeper insights into created samples of both tasks, which we will discuss later.

As such a framework consists not only of these reusable concepts and processing steps but also encompasses guidelines on preparing an optimal model for a specific task, we set particular focus on different fine-tuning strategies in this article. In detail, instead of fine-tuning only a CodeBERT base model to the parameter-matching task, as we did in our previous work, we test six fine-tuning strategies with combinations of two base models, namely CodeBERT [14] and RoBERTa [16], and different datasets for fine-tuning with samples from either one or both tasks. As a result, we create large models covering both the parameter-matching and the endpoint-discovery task and

separate models for each task. To determine an optimal strategy, we evaluate them in terms of generalizability and compare their performance.

Moreover, we test the robustness of the best-performing models of each task and identify issues in NL queries and syntax that can lead to incorrect predictions. These findings should guide developers when using the models as a search engine and show how to prompt a fine-tuned model to obtain accurate results.

In summary, the contribution of this work is a novel approach for the semantic search in Web API documentation implemented on top of BERT. For this:

- We model semantic search as a question-answering downstream task.
- We present the steps to prepare a BERT model for a specific Web API integration task relying on semantic search. This includes preparing task-specific samples for fine-tuning and evaluation, which we extract from OpenAPI documentation of real-world Web APIs.
- We examine different fine-tuning strategies and demonstrate the applicability of our approach to two integration tasks, namely parameter matching and endpoint discovery.
- To identify limitations of our approach, we evaluate the best-performing fine-tuned models in terms of robustness.

We call our approach RESTBERTa, which stands for *Representational State Transfer on Bidirectional Encoder Representations from Transformers approach*. However, other than the name suggests, the approach is not limited to the semantic search in documentation of RESTful APIs. Instead, RESTBERTa can be applied to any Web API that exposes its functionality over URIs and HTTP and uses documents of hierarchically organized parameters for exchanging data, e.g., JSON or XML. To the best of our knowledge, RESTBERTa is the first approach that leverages Transformer-based question answering for the semantic search in Web API documentation.

We make the detailed evaluation results and the datasets that are required for fine-tuning and evaluation available on Zenodo.[1] Additionally, we publish the code used for fine-tuning and evaluation on GitHub.[2]

2 Background and related work

This section discusses the concepts of the original Transformer architecture, the BERT architecture, and the question-answering downstream task. We also describe the characteristics of the two models RoBERTa and CodeBERT, which we use as base models for fine-tuning. Afterward, we review related work that proposes approaches for the two tasks that we cover with our approach.

2.1 Transformers

In 2017, Vaswani et al. introduced the *Transformer* architecture [17], which set a new state of the art in the field of NLP. Similar to the former *Sequence-to-Sequence* (Seq2Seq) architecture for NLP tasks, like language translation and text summarization [18], the original Transformer architecture consists of an encoder, transforming an input text into a context vector, and a decoder, converting the context vector into an output text. While Seq2Seq models rely on recurrent neural networks (RNN), Transformers solely use stacks of point-wise, fully connected layers with a self-attention mechanism. This allows massively parallel processing and makes the utilization of Transformers fast and efficient on modern hardware with parallel computing capabilities, like GPUs. Transformers outperformed former models in various NLP tasks, e.g., text classification [12], text summarization [13], and language translation [17].

2.1.1 BERT

Although Transformers are faster than former NLP models, it is still computationally expensive to train a Transformer to a specific NL task from scratch [19]. Fortunately, Devlin et al. [20] introduced a new language model architecture pre-trained with unlabeled text and afterward fine-tuned with relatively little effort to a specific downstream task. They named this architecture BERT, which stands for *Bidirectional Encoder Representations from Transformers*. Compared to the original Transformer architecture [17], BERT only consists of an encoder, which takes the numerical representation of a text as input and calculates a context vector. This context, i.e., output, vector must be interpreted with respect to the individual pre-training or downstream task.

The original BERT model combines two self-supervised tasks in the pre-training procedure, namely *Masked Language Modeling* (MLM) and *Next Sentence Prediction* (NSP). By taking a pair of two sentences, i.e., sequences of tokens, as input, BERT has to predict whether the second sentence logically follows the first sentence, which is the NSP task, and, at the same time for the MLM task, tokens that have been removed from both sentences. In 50% of the training samples, the second sentence actually follows the first sentence. In the other 50%, the second sentence is chosen randomly. Additionally, 15% of all tokens have been randomly replaced with a special `[MASK]` or a

[1] https://doi.org/10.5281/zenodo.8349083.

[2] https://github.com/SebastianKotstein/RESTBERTa.

random token in both sentences. For the later explanation, it is essential to know that both sentences are separated by a special *separation token* `[SEP]`. Moreover, the whole sequence starts with the *classification token* `[CLS]` and ends with the *end of sequence token* `[EOS]`.

For fine-tuning BERT, a copy of the pre-trained base model is taken, which means the model is initialized with the pre-trained parameters. Additionally, the model is extended with a task-specific output layer. Then, both the parameters of the base model and the task-specific output layer are fine-tuned to a specific downstream task. In the following section, we will explain the question-answering downstream task, which is the foundation for our approach.

2.1.2 Question answering

Question answering (QA) is a downstream task in which a BERT model has to predict a span of text within a given paragraph containing the answer to a given question [20]. Therefore, this form of QA is also called *extractive question answering* since the model has to extract the answer from the paragraph [21] rather than generating an individual answer.

Similar to the two pre-training tasks, MLM and NSP, the model's input is the numerical representation of two sequences of tokens. The first sequence should contain the numerical representation of the tokens of the question. The second sequence is designated to the numerical representation of the tokens belonging to the paragraph. Technically, a tokenizer splits the original question and paragraph into their tokens and infixes the special `[SEP]` between both resulting sequences to separate their content. Additionally, the tokenizer adds the `[CLS]` as the first token to the whole sequence to indicate its start and appends the `[EOS]` token to the sequence. Then, a numerical representation is calculated for each token. For this calculation, BERT uses a combination of token, segment, and position embedding.

The model processes the input and predicts two vectors defining the span of the answer. The first vector gives the probability for each token that the respective token is the start of the answer. Accordingly, the second vector determines the probability for each token that the respective token is the end of the answer.

It is important to emphasize (1) that the size of the Transformer model, i.e., the number of inputs, which are 512 tokens in the case of BERT, limits the length of the input sequence. Moreover, (2) the answer must be within a contiguous span of text, meaning it cannot be distributed over multiple spans in the paragraph [22]. In Sect. 3.2 and Sect. 4.6, we will address these limitations and present workarounds as these constraints challenge the application of Transformer-based QA on both semantic-search tasks.

2.1.3 RoBERTa and CodeBERT

Since the introduction of BERT in 2018, several works have either proposed improved versions of the original BERT approach [16, 23] or used its architecture or one of its derivatives to train models for special language domains and NLP applications [24, 25].

Liu et al. [16], for instance, optimized with RoBERTa the original training approach of BERT. The authors identified several improvements by conducting training experiments with different hyperparameter settings and pre-training tasks. They proposed, amongst others, the following modifications to the original BERT pre-training strategy:

First, while Devlin et al. stated that NSP is very beneficial, especially for the QA downstream task [20], Liu et al. pre-trained RoBERTa with MLM but omitted NSP as they found that the latter task does not improve but even compromises the performance of downstream tasks.

Second, they used a larger byte-level *Byte Pair Encoding* (BPE) vocabulary, introduced by Radford et al. [26], and a corresponding BPE tokenizer. It consists of 50K subwords instead of the 30K character-level BPE vocabulary of BERT. Using bytes instead of characters as the base for sub-word units has the benefit that any input text can be encoded, and "unknown" tokens can be avoided, even with a modest vocabulary size of 50K.

Third, RoBERTa was pre-trained with 160 GB of uncompressed text crawled from books, news, Wikipedia articles, etc., which was roughly ten times more data compared with BERT.

CodeBERT [14] is a model optimized for source-code-related NLP tasks. Relying on the optimizations of RoBERTa, Feng et al. pre-trained CodeBERT with bimodal samples consisting of NL and programming language (PL), i.e., code in Python, Java, JavaScript, PHP, Ruby, and Go, as well as unimodal samples containing only pure NL or pure PL. The objective of CodeBERT was to capture the semantic connection between both language domains and to offer a model that can be used for different PL- and NL-related tasks, such as code search and the generation of code documentation. As proposed by Liu et al. [16], the authors pre-trained CodeBERT using MLM but also *Replaced Token Detection* (RTD).

For pre-training, they used 2.1M bimodal samples, where each sample is an individual code function paired with documentation in NL, plus 6.4M unimodal code samples. The pre-training dataset had a size of approximately 20 GB.[3]

[3] According to the official repository of the dataset: https://github.com/github/CodeSearchNet.

As the language domains in CodeBERT are close to those in our application, we consider CodeBERT as a base model for our approach. Nevertheless, RoBERTa is also an interesting candidate since it was pre-trained with a larger dataset compared with CodeBERT.

2.2 Service and endpoint discovery

Although there is no unique definition of "semantic search" in the field of Web APIs and automated service composition, we observed that the term often goes hand in hand with the tasks of service and endpoint discovery. More specifically, semantic search, and especially service discovery, originates from the idea of making the semantics of WSDL/SOAP-based Web services machine-understandable by describing them in a structured way in rich description formats and with ontologies. Semantic-search engines should process ontology-based knowledge representations, understand the described semantics, and support integrators in identifying Web services that match a list of input keywords or even complex queries [27, 28]. With the shift from rich description formats and ontologies toward more lightweight formats, like OpenAPI, RAML, WADL, and API Blueprint, and the advent of Web APIs following the principles of Representational State Transfer (REST), it is, in our opinion, necessary to adjust the requirements for semantic search: Search engines must be able to process state-of-practice Web API documentation consisting of structured syntax and unstructured NL descriptions for semantics rather than ontology-based knowledge representations to identify APIs and their endpoints.

Indeed, the research focus has expanded from solutions addressing only service discovery in WSDL/SOAP-based Web services toward approaches for API and endpoint discovery in Web APIs over the last few years. Nevertheless, although Web APIs have displaced WSDL/SOAP-based Web services in practice, as well as related description formats and ontologies, there is still active research in the former area [29–31]. This development makes it difficult to capture the large body of work in this field of research. Therefore, we focus the presentation of related work on studies that either rely on a similar approach, in our case, the use of deep neural networks (DNN) with Transformers, or address the same domain and task, i.e., endpoint discovery in Web APIs.

In [32], Yang et al. proposed the DNN architecture ServeNet for automated Web service classification. Based on a given service name and description in NL, ServeNet predicts the categories of a Web service, e.g., "Weather", "Music", and "Payments". The architecture of ServeNet consists of convolutional neural network (CNN) layers and a bidirectional long short-term memory (BiLSTM) layer. It is mostly the same as in their former work [33], except that they use the pre-trained BERT uncased model as an embedding input layer instead of GloVe embeddings [34]. This input layer generates a sentence embedding for the service description and word embeddings for the service name. They trained ServeNet with 8733 samples extracted from ProgrammableWeb, where each sample represented a service and consisted of its name, description, and the primary category out of 50 available categories. For evaluation, they used another set of 2210 samples and the top-k metric as we did. ServeNet achieved an accuracy of 91.58% for $k = 5$, which means that in 91.58% of all evaluation samples, the correct category was under the five highest-ranked predictions, and 69.95% for $k = 1$.

Another Transformer-based approach for Web service classification and recommendation was proposed by Wang et al. [35]. However, unlike ServeNet [32], they pre-trained a BERT model using MLM, RTD, and Contrastive Learning from scratch. As training data, they crawled 128,536 raw documents, consisting of unstructured NL descriptions, Web API syntax, and exemplary source code, from ProgrammableWeb. Afterward, they fine-tuned the pre-trained base model, which they named ServiceBERT, to the task of Web API tagging, which is comparable to the objective of ServeNet, and a separate base model to a mashup-oriented API recommendation task. In both tasks, ServiceBERT outperformed former methods and approaches, including ServeNet. This emphasizes the effectiveness of pre-training a model with domain-specific data to this special language domain in Web services and Web APIs. ServiceBERT might be another interesting base-model candidate for our semantic search approach, but the model is not publicly available.

While [32] and [35] proposed solutions that should enable the discovery of relevant candidates on the service level, Liu et al. [36] introduced an approach for the discovery of endpoints in a set of Web APIs. Their proposed model takes an NL query, which describes the purpose of the target endpoint, as input and predicts endpoints matching the query. Technically, they treat endpoint discovery as a multi-class classification problem. The model, consisting of embedding input layers, BiLSTM layers, and multiple dense layers, has a static number of outputs. Each output represents a specific endpoint of a Web API. Consequently, the approach is limited to Web APIs and their endpoints that were known when the model was created. Unknown, e.g., recent, Web APIs and endpoints cannot be considered. Nevertheless, the approach yielded a remarkable accuracy of 91.13% for $k = 1$ and 97.42% for $k = 10$, with a model covering 1127 Web APIs and 9004 endpoints. In another experiment with a larger model serving 9040 Web APIs with 49,083 endpoints, they achieved an accuracy of 78.85% for $k = 1$ and 89.69% for $k = 10$.

Most of the work we reviewed focuses on discovery on Web service or Web API level [32, 35] rather than on the endpoint level. To the best of our knowledge, the processing of NL and syntax descriptions with Transformers for endpoint discovery has not been addressed yet. RESTBERTa is the first approach that processes NL and syntax descriptions to identify matching endpoints.

2.3 Semantic parameter matching

In our previous work [15], we reviewed studies from the field of Web service and Web API composition that solve parameter matching by grounding parameters in Web APIs in globally defined semantic concepts and models, e.g., in ontologies [9, 10] and knowledge bases (KB) [37, 38]. Although these solutions reduce the manual effort for annotating parameters with machine-understandable semantic descriptions, it remains a challenge to manually find and select a suitable ontology or KB covering the entire domain of the respective Web API beforehand. It is even more problematic if a suitable ontology or KB does not exist and must be manually created.

However, parameter matching is not only an exclusive topic of the Web API domain but is also addressed in the domain of data integration, albeit with a focus on matching database schemas and their attributes. This domain has proposed several approaches relying on DNNs, which are close to our solution:

The matching of two schemas and their attributes can be described in a similarity matrix [39]. Shraga et al. presented in [39] ADnEV, an algorithm that calibrates, i.e., improves, the similarity parameters in a given matrix in multiple iterations by using a self-evaluation mechanism. For calibration and self-evaluation, ADnEV relies on a DNN consisting of CNN layers to capture spatial patterns in similarity matrices and RNNs to capture the improvement of a matrix over multiple iterations. To train the DNN, the authors generated similarity matrices from schema pairs of three benchmarking datasets by using three state-of-the-art schema matchers.

In [40], Zhang et al. proposed SMAT, a DNN model that uses an attention over attention (AOA) mechanism to obtain semantic mappings between attributes in source and target schemas. To determine semantic correlations between attributes, SMAT considers the syntax name of an attribute, its NL description, and the NL description of the table, i.e., schema, the attribute is contained within. Moreover, the authors treat the matchmaking of attributes in schemas as a binary classification problem. This means that SMAT takes the aforementioned features of two attributes as input and predicts whether they match. To capture the semantics of an attribute from text consisting of NL descriptions and a syntax name, they use a combination of BPE, GloVe embeddings [34], and a BiLSTM network to process input text. Zhang et al. used different schema-matching datasets with samples of different domains and applications, like healthcare, Web forms, and purchase orders for training and evaluation. They evaluated SMAT against five baseline models, including a fine-tuned BERT [20] uncased model and ADnEV [39], and used precision, recall, and F1 score for comparison. While SMAT outperformed all baseline models in terms of precision and F1 score, the recall score of BERT was comparable with that of SMAT.

In summary, the work of Zhang et al. [40] has some parallels to our approach, like the processing of syntax to construe semantics and the use of a DNN with an attention mechanism. However, it also differs in several aspects: SMAT processes syntax names and descriptions of both attributes to determine a semantic correlation between two attributes. RESTBERTa, however, tries to identify semantic correlations from NL queries describing source parameters to syntax names of target parameters but without considering the syntax names and NL descriptions of source and target parameters, respectively. Furthermore, RESTBERTa processes hierarchical syntax schemas with deep structures to construe semantics, while the schemas in databases, and thus attribute names, are relatively flat. Ultimately, SMAT relies on a network with BiLSTM and AOA and uses binary classification, whereas we use a Transformer encoder model with QA for matching parameters. Unfortunately, the performance of both approaches cannot be compared, since they use different metrics and datasets for their evaluation.

3 RESTBERTa: semantic search with question answering

In this section, we explain our understanding of the term "semantic search" when applied to state-of-practice Web API documentation. Furthermore, we present the concept of modeling semantic-search tasks as QA downstream tasks, which is the foundation of RESTBERTa.

3.1 Semantic search in Web API documentation

Semantic search applied to Web API documentation is the generic task of identifying a Web API element whose syntax is described in a documentation that semantically matches a given query describing the meaning and purpose of the target element in NL. Without providing any further semantic NL descriptions for the Web API elements in the documentation, the search engine has to construe the semantics of a Web API element from its syntax

description, e.g., from syntax names, and compare it with the NL query to determine a match.

At this point, how the search engine would construe semantics from syntax and how this comparison mechanism would look remains open. However, to construe semantics from syntax accurately, developers must use meaningful names and identifiers in syntax, e.g., for parameter names or path segments in URIs, which reflect the semantics of the respective element and its relation to other elements. Using meaningful names for software artifacts is a fundamental principle in software engineering [41]. This should be the case in every well-designed Web API.

Moreover, *to identify*, in this sense, means that the search engine has to extract one matching Web API element from the documentation. We consider the documentation to be a selection of elements. Depending on the specific search task, e.g., endpoint discovery or parameter matching, relevant elements might be endpoints or payload parameters. Thus, it is reasonable to limit the selection to relevant elements and present only a subset of the documentation to the search engine, e.g., a list of endpoints or a specific payload schema.

3.2 Semantic search as a question-answering downstream task

Our idea is to model semantic search in Web API documentation as a QA downstream task and fine-tune a pre-trained BERT model to this task to use it as a search engine.

Aligned with our definition of semantic search in the previous Sect. 3.1, the objective of the fine-tuned BERT model would be to extract the Web API element from a selection of elements whose syntactic appearance matches the semantics described in an NL query. For the application of QA to this task, we formulate the original problem of QA as a multiple-choice task: Given a selection of Web API elements and a query in NL, which describes the meaning and purpose of the target element, the model should extract the element from the selection that matches the query best. In detail, we feed the NL query and the selection of Web API elements in the form of a syntax structure into the BERT model and let the model extract the Web API element from the presented syntax structure.

From a technical perspective, applying QA to semantic search in Web documentation is not straightforward. We have to bridge the gap between the structure of Web API documentation, in particular how the syntax of Web API elements is described, and the characteristics of BERT. In the following, we discuss these characteristics requiring special treatment and present our solution.

BERT and its derivatives are mostly optimized for the processing of NL, which is commonly linear text. In Web API documentation, however, hierarchically organized tree structures for describing syntax are omnipresent. As illustrated in Fig. 1a, a payload schema of a JSON or XML document can be described in a tree structure consisting of nodes whose labels are the syntax names of the respective properties. Inner nodes represent either objects or arrays, and leaf nodes are parameters. Similarly, the endpoints of a Web API typically rely on a hierarchical URI model (see Fig. 1b), whose paths and operations can be described in a tree structure: While inner nodes represent path segments, leaf nodes are the operations in the form of HTTP verbs. In both cases, a Web API element, i.e., a parameter or an endpoint, is uniquely addressed by naming all nodes along the path from the root of the tree to the respective leaf.

A naïve approach would be to treat a tree structure, regardless of whether it is a payload schema or a URI model, as linear text and feed it together with the NL query, which is naturally linear text, as a sequence of tokens into the model. However, this would destroy not only the inherent hierarchical structure but also, we assume, impede the model from making correct predictions. Additionally, BERT can predict the start and end positions of an answer only if this answer is contained within a contiguous span of text (see Sect. 2.1.2). Treating a hierarchical structure just as linear text would fragment the answer, i.e., the path of a respective Web API element, and scatter it across the linear text.

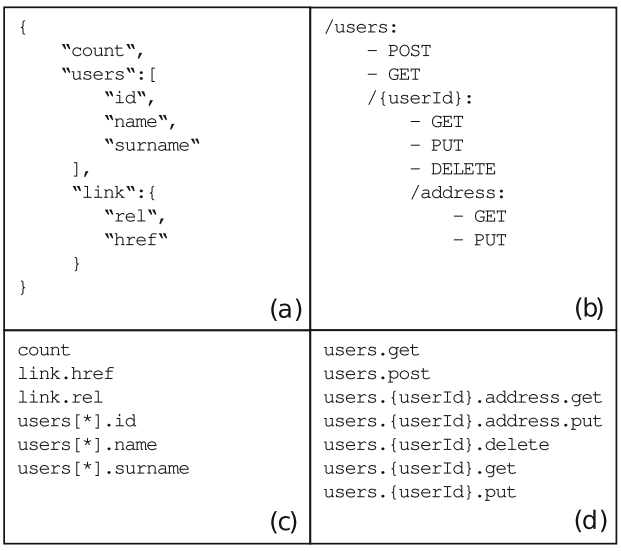


Fig. 1 Hierarchical payload schema (a) and URI model (b) serialized into lists of parameter (c) and endpoint (d) paths, respectively

Hence, we serialize a syntax tree structure by extracting all Web API elements and their paths. We transform each path, regardless of whether it addresses a parameter or an endpoint, into an XPath-like notation (see Figs. 1c and d). Additionally, we flag all properties that are arrays in parameter paths with a `[*]` tag. Likewise, we use curly brackets (`{...}`) to indicate URI parameters in endpoint paths. The result is a linear list of paths so that every possible answer forms a contiguous span of text, which satisfies the contiguous span constraint. The model has to choose an answer by naming the path of the targeting Web API element.

As illustrated in Figs. 1c and d, we sort the list of paths in alphabetical order. This has the effect that Web API elements located under the same parent node are relatively close to each other in the resulting list of paths. It should support the model in construing hierarchical relations from the resulting linear text.

4 Data preparation and fine-tuning architecture

It is necessary to fine-tune a pre-trained base model to the modified QA downstream task described in Sect. 3.2 to use BERT as a search engine for the semantic search in Web API documentation.

Technically, fine-tuning has two effects: First, it aligns the base model to the characteristics of the QA downstream task. Second, by fine-tuning the base model with task- and domain-specific samples, we adjust its inherent language model to the modalities of NL and Web API syntax language and how they are semantically connected.

In Sect. 3.2, we have shown that both semantic-search tasks, namely parameter matching and endpoint discovery, can be theoretically addressed with the same QA approach. Both tasks rely on the search of Web API syntax elements following a hierarchical structure that can be serialized into a list of XPaths. However, although they share a similar syntax and are subject to the same language domain of Web APIs, the semantics of a parameter differs from that of an endpoint as both have a completely different meaning and purpose. Therefore, instead of fine-tuning one model for both semantic-search tasks with mixed samples, it is reasonable to fine-tune two separate models, i.e., one for each task, with task-specific samples. In this work, we want to test six different fine-tuning strategies and compare the performance of the resulting models.

Before we present the different fine-tuning strategies in Sect. 5, we clarify how we create samples for fine-tuning, as well as the later evaluation, and fine-tune a pre-trained base model to the modified QA downstream task with these samples.

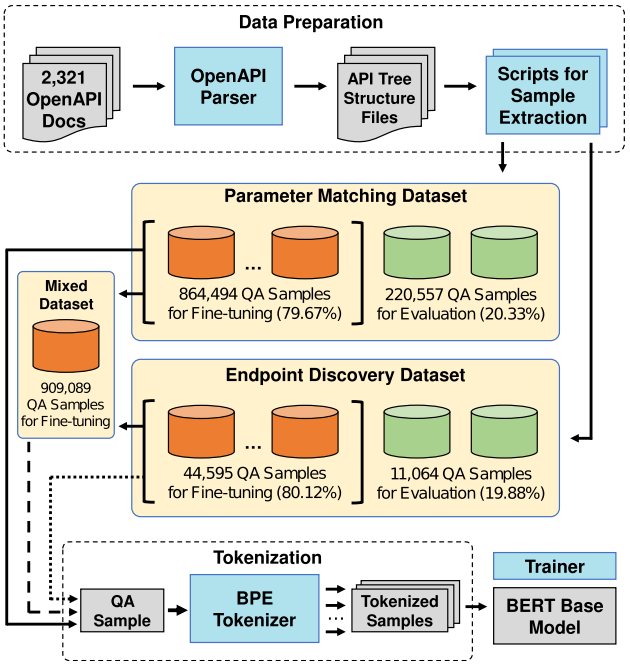


Fig. 2 Architecture for data preparation and for fine-tuning a pre-trained BERT base model to the QA downstream task with either task-specific or mixed QA samples

Figure 2 illustrates our architecture for data preparation and fine-tuning: It consists of a data preparation process, which creates three datasets with task-specific or mixed QA samples. Depending on the respective fine-tuning strategy (see Sect. 5), we will use the QA samples of one of these datasets to fine-tune a specific base model but also for the later evaluation. For fine-tuning, as well as inference, a tokenizer must convert the QA samples into a numerical representation before they can be fed into the BERT model.

This section starts with an introduction of the main characteristics of the OpenAPI format, whose documentation serve as data sources for QA samples for fine-tuning and evaluation. Afterward, we present the details of the data preparation process and provide a deeper insight into the created task-specific datasets, especially into collected metrics describing the samples. Finally, we present the details of the tokenization process.

4.1 OpenAPI format

As illustrated in Listing 1, the content of an OpenAPI documentation is a tree structure.

```
openapi: 3.0.3
paths:
 /groups/{groupId}/users:
  get:
   responses:
    '200':
     summary:
      Get users of group
     description:
      Returns all users of the group
      having the specified ID.
     content:
      application/json:
       schema:
        type: object
        properties:
         count:
          description: Number of users
          type: number
         users:
          type: array
          items:
           type: object
           properties:
            id:
             description: ID of a user
             type: string
            name:
             description:
             Name of a user
             type: string
         link:
          $ref: '#/schemas/linkObject'
```

The path object lists all paths the Web API exposes plus their operations labeled with the intended HTTP verb, e.g., `GET` or `POST`. The OpenAPI standard specifies two fields for describing the semantics of an operation in NL. As the field name suggests, the summary field is intended for a short description of the purpose of the operation. The description field should contain a detailed explanation of what the operation does and how it behaves. Throughout the paper, we use the term endpoint, which is the combination of a path and a respective operation.

As the OpenAPI standard allows the specification of individual response payloads for each possible status code, an operation may have one request but multiple response payloads. Moreover, it is possible to define multiple schemas for a specific request and response payload. Each schema defines a different resource representation format that the endpoint supports, e.g., JSON or XML, and is labeled with the intended media type, e.g., `application/json`.

The OpenAPI schema object specification extends the original JSON schema specification. It allows one to define the schema of documents having a hierarchical structure, such as JSON and XML, which are often used in request and response payloads of Web APIs [3].

A schema consists of nested properties representing arrays, objects, and attributes. Arrays and objects can have other properties as children. They are commonly the inner nodes in the resulting tree structure. Attributes, on the other hand, are leaves in the tree structure and are types of string, number, integer, or boolean. In the context of schemas and parameter matching, we use the term parameter as an alias for attribute throughout the paper. Like operations, each property can have a description field to explain its meaning and purpose in NL.

Schemas can be defined globally, even outside of the document, and referenced at multiple places, which avoids redundant schema definitions and increases their reusability in Web API documentation. For example, they can be reused in request or response payloads of different operations or as sub-schemas in larger schemas.

Relying on OpenAPI documentation as a data source for our QA samples has multiple advantages: First, all documents are in the same standardized format. This simplifies the extraction of samples and allows us to use uniform code for processing all documentation. Second, due to the popularity of the OpenAPI format, a large amount of publicly available documentation of real-world Web APIs exists in this format. Third, OpenAPI documentation are a potential source for samples for both semantic-search tasks as they list the endpoints of Web APIs and their payload schemas with parameters.

Nevertheless, OpenAPI documentation also bear the risk that they contain many NL descriptions with similar or even identical wording compared with the wording used in syntax. The author of the NL description was either the same person who also designed the syntax and thus tended to use the same words for both or was at least biased when writing the NL description after reading the syntax, or vice versa. Consequently, multiple QA samples might have identical wording in their NL descriptions and the Web API element paths. This could compromise the robustness of a fine-tuned model. Instead of capturing the underlying meaning of both description and syntax, it might lean toward seeking words given in the Web API element path and the NL description. Hence, the model could fail whenever synonyms are used. As part of our evaluation, we analyze the robustness of our fine-tuned models in terms of synonyms (see Sect. 6.3).

4.2 Data preparation

In an initial step, we downloaded 2358 OpenAPI documentation from APIs.guru,[4] which was the entire available

[4] https://apis.guru.

set on this platform as of October 2022. Then, we used a tool to parse the OpenAPI documentation and convert the parsed content into a canonical tree structure. In detail, the tool tries to resolve all schema references, including `allOf`, `oneOf`, and `anyOf`, and also attaches input and output schemas directly to operation nodes. We excluded 32 of these 2358 initially downloaded OpenAPI documentation due to syntax issues and another five due to their large size and schema complexity. The resulting 2321 tree structures were stored as files on disk.

In the second step, we extracted the task-specific samples from these 2321 tree structure files. For this, we implemented two scripts, one for each task:

The first script extracts the QA samples for the parameter-matching task. The script analyzes the request- and response-payload schemas of all endpoints of all tree structure files and extracts all parameters. For each parameter that has an NL description, it creates a QA sample consisting of the NL description, which is the question, the parameter in path notation, which is the answer, and the serialized schema, i.e., the list of parameters in path notation (see Fig. 1c), which is the paragraph.

The second script extracts the QA samples for the endpoint-discovery task. It iterates over all tree structure files and analyzes the documented endpoints. For each endpoint that has an NL description, which means that at least the description or the summary field is set, the script creates a QA sample. Each QA sample consists of the NL description, which is the question, the endpoint in path notation, which is the answer, and the list of endpoints of the respective Web API in path notation (see Fig. 1d), which is the paragraph. If both summary and description are set for an endpoint, the script compares their content lengths and chooses the NL description having more tokens.

Additionally, we implemented the following constraints for a valid QA sample in both scripts: The NL description must consist of at least three tokens. It must not exceed a maximum length of 96 tokens. Both scripts remove all URIs embedded in NL descriptions beforehand to avoid an exceedance of these 96 tokens. If an NL description consisting of multiple sentences still exceeds the limit of 96 tokens even after removing all URIs, the description is truncated by removing all trailing sentences that caused the exceedance.

These constraints should ensure that the descriptions contain enough semantic context. However, as the input size of BERT is limited to 512 tokens, they also should ensure enough space for the paragraph. For the same motivation, we limited the maximum depth of a path of a Web API element to eight nodes (without the root node) for both paths in the paragraph and the answer. For these and all subsequent length calculations, i.e., to determine the number of tokens, we used the byte-level BPE tokenizer proposed by the base model RoBERTa [16] (see Sect. 2.1.3).

4.3 Parameter-matching dataset

As a result, out of 2,502,078 extracted parameters, the first script created 1,085,051 valid QA samples for the parameter-matching task. 1,417,027 parameters, i.e., 56.63%, must be excluded either as their paths exceeded the maximum depth or their NL descriptions did not satisfy the constraints defined in Sect. 4.2 or both. In detail, in 159,781 cases, parameter paths were too deep. 1,362,649 parameters had missing (1,311,950), too short (50,246), or too long NL descriptions (453). The script created these 1,085,051 valid QA samples from the content of 1473 OpenAPI documentation and 54,611 request and response schemas. The remaining 848 documentation either did not contain at least one schema parameter that would satisfy all constraints or did not document request or response schemas with parameters.

To better understand the created QA samples, the script additionally collected the metrics listed in Table 1: The description, i.e., question, length in tokens ranged between three and 96 tokens, which were the lower and upper limits we set as constraints, with a mean value of 15.22 and a median of 10. Both values for mean and median suggest that rather compact than verbose descriptions were present in the created QA samples. The schemas, i.e., paragraphs, encompassed between one and 160,955 tokens, with a mean value of 12,721.25 and a median of 1196. Moreover, the number of parameters in a schema ranged between one and 6129, with a mean of 576.07 and a median of 105 parameters. At first glance, these high mean and median values suggest that using large schemas with hundreds of parameters is common practice in Web APIs. However, it is important to bear in mind that a schema consisting of n parameters is a potential source for n QA samples and, therefore, occurs up to n times in the dataset as a paragraph. Among the entire set of 1,085,051 QA samples, there were only 12,921 unique schemas.

Finally, the script evenly distributed the 1,085,051 valid QA samples into ten data chunks and stored the chunks on disk. We, furthermore, set the constraint that all QA samples originating from the same Web API were stored in the same chunk. As we used the samples of eight chunks with 864,494 samples (79.67%) for fine-tuning and the remaining two chunks with 220,557 samples (20.33%) for evaluation, we ensured that the model was fine-tuned with samples from specific Web APIs but confronted with unseen Web APIs and their samples during evaluation. This helped us to investigate model capabilities in terms of generalization. However, it resulted in a slightly

Table 1 List of collected metrics for the datasets of the parameter-matching and endpoint-discovery task

Metric	Parameter matching (PM)	Endpoint discovery (ED)
Total samples	1,085,051	55,659
Samples in fine-tuning set	864,494 (79.67%)	44,595 (80.12%)
Samples in 1st eval. set	110,877 (10.22%)	5536 (9.95%)
Samples in 2nd eval. set	109,680 (10.11%)	5528 (9.93%)
Length of question (in tokens)		
Min	3	3
Max	96	96
Mean	15.22	28.33
Median	10	14
Std. deviation	15.69	27.73
Length of paragraph (in tokens)		
Min	1	1
Max	160,955	13,262
Mean	12,721.25	2852.53
Median	1196	1301
Std. deviation	29,242.51	3431.92
# Web API elements in paragraph		
Min	1	1
Max	6129	930
Mean	576.07	197.88
Median	105	101
Std. deviation	1138.92	217.98

unbalanced distribution since the number of samples per Web API varied for each Web API.

4.4 Endpoint-discovery dataset

For the endpoint-discovery task, the second script created 55,659 valid QA samples. Similarly to the QA samples for the parameter-matching task, 10,051 out of 65,710 analyzed endpoints, i.e., 15.30%, must be excluded due to too deep paths (7021), missing (2323), too short (673), or too long NL descriptions (90). 2012 OpenAPI documentation served as a source for these valid QA samples. The remaining 309 documentation either did not contain at least one endpoint that would satisfy all constraints or did not document endpoints.

Due to the structure of Web APIs and the hierarchical relation between endpoints and schema parameters, the number of QA samples for the endpoint-discovery task is unsurprisingly magnitudes smaller than the number of QA samples for the parameter-matching task. Nevertheless, the share of parameters that must be excluded for the parameter-matching tasks is higher than that of excluded endpoints.

While most parameters were excluded due to missing NL descriptions, endpoints were mainly excluded because of the depth of their paths but rarely due to a missing NL description. These numbers suggest that authors of OpenAPI documentation spend more effort on the semantic description of endpoints but often neglect the description of parameters. Apart from this, these numbers reveal only a quantitative perspective. We cannot make any statement concerning the quality of descriptions.

The descriptions, i.e., questions, had between three and 96 tokens, with a mean value of 28.33 and a median of 14, which indicates that more verbose descriptions were used in these QA samples compared with the samples of the parameter-matching task. In contrast, the paragraphs contained fewer tokens and Web API elements compared with the samples of the other task: The paragraphs comprised between one and 13,262 tokens, with a mean value of 2852.53 and a median of 1301. Between one and 930 endpoints were contained within a paragraph, with a mean of 197.88 and a median of 101. We noticed 1889 unique paragraphs, i.e., lists of endpoints, among these 55,659 valid QA samples.

Similar to the dataset for the parameter-matching task, the script distributed the 55,659 valid QA samples into ten data chunks. While eight chunks, containing 44,595 samples (80.12%), were dedicated to fine-tuning, we used the remaining two chunks, with 11,064 samples (19.88%), for evaluation.

4.5 Mixed dataset

In preparation for fine-tuning a base model to both tasks, we merged the 864,494 fine-tuning QA samples of the parameter-matching task and the 44,595 fine-tuning QA samples of the endpoint-discovery task into an additional dataset. This mixed dataset contained 909,089 QA samples.

4.6 Tokenization

Before we can feed the QA samples into a BERT model for fine-tuning and inference, they must be converted into a numerical representation. For this purpose, we used the byte-level BPE tokenizer mentioned in Sect. 2.1.3.

Due to BERT's input length limitation, the tokenizer must split a single QA sample into multiple tokenized samples if the input sequence, consisting of the question and the paragraph of the QA sample, has more than 512 tokens. The tokenizer splits only the paragraph and leaves the question unaltered. Consequently, the question is the same for each tokenized sample, but another fragment of the original paragraph is presented to the model. We set a *doc stride* of 128 so that the resulting fragments overlap by 128 tokens.

For instance, Fig. 3 lists five tokenized samples in their token representation resulting from splitting a QA sample with the question "The first name of a user" and the paragraph displayed in Fig. 1c. In this example, we set a maximum length of 16 tokens and a doc stride of two.

The tokenizer generates the numerical representation of the input token sequence for each tokenized sample. In detail, it outputs a vector of token identifiers used as model input. For fine-tuning, the tokenizer calculates the start and end index of the answer span on the token level for each tokenized sample. It adds these two indices as labels to the tokenized sample.

The splitting of paragraphs into multiple fragments may result in many tokenized samples that are unanswerable as the answer is no longer in the respective fragment. If a tokenized sample is unanswerable, the start and end index of the answer span is set to the `[CLS]` token (see Fig. 3).

Furthermore, the tokenizer may split the path of the Web API element being the correct answer into two halves so the resulting fragments contain only a portion of the original path at its boundaries. By setting a doc stride of 128, we can ensure that at least one of these tokenized samples contains this path entirely but only if the path has fewer than 128 tokens.

Nevertheless, the splitting of QA samples into multiple tokenized samples led to an unbalanced distribution of answerable and unanswerable samples for both task-specific fine-tuning datasets:

For the parameter-matching fine-tuning dataset, we noticed that 35,744,686 (97.26%) of all 36,749,955 resulting tokenized samples were unanswerable, whereas only 1,005,269 (2.74%) of the samples contained the answer within the presented fragment. 537,905 QA samples (62.22%) in this fine-tuning dataset exceeded the maximum token length of 512 and must be split. These numbers were unsurprising considering the large schemas consisting of hundreds of parameters and thousands of tokens (see Table 1).

Similarly, the splitting of the QA samples of the fine-tuning dataset of the endpoint-discovery task resulted in 324,957 (85.92%) tokenized samples being unanswerable and only 53,238 (14.08%) tokenized samples that were answerable. In this dataset, 32,463 QA samples (72.80%) had an input sequence with more than 512 tokens.

Using these unbalanced samples for fine-tuning, we assume that the resulting models would be highly biased toward unanswerable samples. This means they tend to classify any input as unanswerable during inference rather than predicting the correct answer. To avoid this, we set a maximum of three unanswerable tokenized samples that could result from tokenizing a single QA sample. We picked these unanswerable tokenized samples randomly for each fine-tuning dataset.

As a result, after processing all 864,494 QA samples of the fine-tuning dataset for the parameter-matching task, the tokenizer output 2,441,687 tokenized samples, of which 1,436,418 were unanswerable (58.83%), and 1,005,269 (41.17%) were answerable. Furthermore, the fragments of these tokenized samples contained between one and 126 parameter paths, with a mean of 30.32 and a median of 27 paths.

The processing of the 44,595 fine-tuning QA samples of the endpoint-discovery task resulted in 81,084 (60.37%) unanswerable and 53,238 (39.63%) answerable tokenized

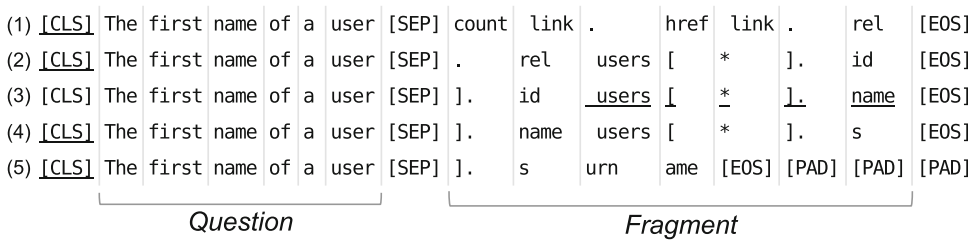


Fig. 3 Tokenized samples with max. length = 16 and doc stride = 2. Only the third sample contains the correct answer (underlined tokens) entirely, while for the other samples, the start and end positions are set to the [CLS] token

Table 2 Model configurations for fine-tuning

Fine-tuned model	Fine-tuning dataset	Base model
CB-PM	Parameter matching (Sect. 4.3)	CodeBERT [14]
RO-PM	Parameter matching (Sect. 4.3)	RoBERTa [16]
CB-ED	Endpoint discovery (Sect. 4.4)	CodeBERT [14]
RO-ED	Endpoint discovery (Sect. 4.4)	RoBERTa [16]
CB-PM+ED	Parameter matching + Endpoint discovery (Sect. 4.5)	CodeBERT [14]
RO-PM+ED	Parameter matching + Endpoint discovery (Sect. 4.5)	RoBERTa [16]

samples. The number of endpoint paths in the fragments of these tokenized samples ranged between one and 93, with a mean value of 33.27 and a median of 32.

As expected, the tokenized samples of both tasks contained fewer Web API elements in their fragments as in the paragraphs of the original QA samples (see Table 1) and the number of elements was also more balanced over all tokenized samples.

The tokenizer output 2,576,009 tokenized samples after processing the 909,089 fine-tuning QA samples of the mixed dataset, covering both tasks. 1,058,507 (41.09%) samples were answerable; 1,517,502 (58.91%) samples were unanswerable.

5 Experiments

In our previous work, we focused on parameter matching and fine-tuned a CodeBERT model to this task. With the extension of our approach to a second task, namely endpoint discovery, and by generalizing this approach, further fine-tuning strategies came into consideration. As mentioned at the beginning of Sect. 4, it is reasonable to fine-tune not only one large model for both tasks but two separate models for each task with task-specific samples. Furthermore, with CodeBERT, a pre-trained BERT model exists that is aligned to the language domain of PL and NL, which we estimate to be close to the language domain of our application. CodeBERT relies on the architecture of RoBERTa, but the latter was pre-trained with much more data, albeit mainly NL. Hence, in addition to CodeBERT, RoBERTa is also an interesting base-model candidate.

Eventually, we tested six different fine-tuning strategies. Each strategy was an independent fine-tuning process, using a specific combination of base model and fine-tuning samples. Each fine-tuning process resulted in an individually fine-tuned model. Table 2 lists these combinations of base models and fine-tuning samples, which we discuss in the following:

The first four combinations address the strategy of fine-tuning separate models for each task. We fine-tuned one instance of the CodeBERT base model and another of the RoBERTa base model with the 2,441,687 tokenized samples of the parameter-matching dataset to this task. We named the resulting fine-tuned models CB-PM and RO-PM. These model names, which we will use throughout the remaining paper, contain the code of the underlying base model, i.e., CB for CodeBERT and RO for RoBERTa, as well as the tokenized samples the models were fine-tuned with, i.e., PM for samples from the parameter-matching dataset, ED for samples from the endpoint-discovery dataset, and PM+ED for the dataset with mixed samples.

In the same way, we used the 134,322 tokenized samples of the endpoint-discovery dataset to fine-tune one instance of the CodeBERT base model and another of the RoBERTa base model to the endpoint-discovery task. We named the resulting models CB-ED and RO-ED using the same systematic as before.

To fine-tune two large models addressing both tasks, which was the fifth and sixth strategy and combination, we used the 2,576,009 tokenized samples of the mixed dataset. We fine-tuned one instance of the CodeBERT base model and another of the RoBERTa base model with these samples. The resulting models had the labels CB-PM+ED and RO-PM+ED.

In each process, we fine-tuned the respective base model with ten epochs and a batch size of 16 on a single GPU system.[5] The entire set of generated tokenized samples of the respective dataset (see Sect. 4.6) was used for each epoch. We saved a checkpoint of the model after each epoch of fine-tuning. One epoch of fine-tuning with the 134,322 tokenized samples of the endpoint-discovery task took about one hour. The 2,441,687 samples of the parameter-matching task, as well as the 2,576,009 samples of the merged dataset, required roughly one day of computing time for one epoch.

6 Evaluation

To determine an optimal strategy, we evaluated all six fine-tuned models in terms of generalizability and compared their performance. Generalizability is a quality attribute

[5] Nvidia Ampere GPU.

that describes the performance of a model when applied to unseen data [42], i.e., samples that were not presented to the model while pre-training or fine-tuning.

Afterward, we evaluated the robustness of the two models that yielded the best performance on the parameter-matching and endpoint-discovery task. Robustness is another quality attribute and is understood as the model's resilience to outliers and noisy data [42]. To some extent, we can ensure uniformity in the presented paragraphs, e.g., by always presenting Web API elements in path notation, but the question is subject to strong variations. In input queries, we must expect individual wording, e.g., synonyms, and different sentence structures, e.g., questions instead of descriptions. This may challenge the robustness of a fine-tuned model.

We withheld two data chunks in each task-specific dataset to evaluate generalizability and robustness. These two data chunks contained approximately 20% of all QA samples of the respective dataset. They did not only contain unseen QA samples of different Web APIs, i.e., samples we did not use for fine-tuning. They may also contain new combinations of Web API elements in paragraphs, i.e., payload schemas and list of endpoints, resulting from Web APIs that we completely excluded as sources for fine-tuning samples (see Sect. 4.3 and 4.4).

In Sect. 4.6, we have explained the tokenization process that splits a QA sample into multiple tokenized samples and inputs them into the model in their numerical representation. Before discussing the evaluation results in this section, we have to clarify how we interpret the model's output to obtain a chosen Web API element, i.e., a predicted answer. This output interpretation process is required for the later application of the model as a search engine (see Sect. 7) but also for the evaluation to determine whether a QA sample is predicted correctly.

6.1 Output interpretation

A BERT model that has been fine-tuned to the QA downstream task outputs two vectors when making a prediction for a tokenized sample. The first vector contains a logit for each token of the input sequence that indicates the probability that the respective token is the start of the span containing the answer. Similarly, the second vector contains a logit for each token stating the probability that the respective token is the end of the answer span. An input size of 512 tokens leads to 512 x 512 possible start-end combinations, i.e., answer spans. We can rank these spans by calculating a score for each start-end combination, which is the sum of the start and the end logit.

It would be relatively straightforward to interpret these results in pure NL applications. With only minor postprocessing, the QA system could directly present the highest-ranked span to the user. However, a search engine for our semantic-search tasks should extract a Web API element from the presented fragment by unambiguously naming its element path. As a predicted span may contain multiple element paths that are either entirely or only partially contained, we require additional logic to interpret the model's output.

We implemented this logic as part of a component named output interpreter. This output interpreter has the objective of removing predicted spans that are invalid or lead to ambiguous answers. This could happen if (1) the resulting span is reversed, i.e., the end position is before the start position. (2) The span contains tokens that do not belong to the fragment but to the question. And (3), although being syntactically correct, the span contains multiple Web API element paths that are entirely or only partially covered.

While (1) and (2) can also be an issue for pure NL applications, the issue of ambiguous answers (3) is unique to our application. To reduce a span to a single Web API element and, therefore, an unambiguous answer, we implemented a function that distinguishes the following cases: If the span contains multiple element paths, but one path is entirely covered within the span, the function chooses this entirely-covered element path as the answer. If the span contains, however, only partially covered paths, it chooses the element path with more covered characters. In any other case, especially if more than one element path is entirely covered, the span is marked invalid and removed from the results list. Furthermore, the model may predict a tokenized sample as unanswerable. In this case (4), both start and end positions point at the first token of the input sequence, which is the `[CLS]` token. Figure 4 presents four exemplary answer spans to which these four cases (1–4) apply.

As a result, the output interpreter creates a list of suggested Web API elements that apparently match the description in the question.

6.2 Generalizability

We tested the performance of all six fine-tuned models applied to unseen task-specific samples. Or in other words, we evaluated their generalizability with respect to the task they were fine-tuned to.

We used the 110,877 QA samples of the first of the two withheld chunks of the parameter-matching dataset to evaluate the performance of the models CB-PM, RO-PM, CB-PM+ED, and RO-PM+ED, which were 10.22% of all samples of this dataset. Similarly, we evaluated the performance of the models CB-ED, RO-ED, CB-PM+ED, and RO-PM+ED on the endpoint-discovery task with 5536 QA

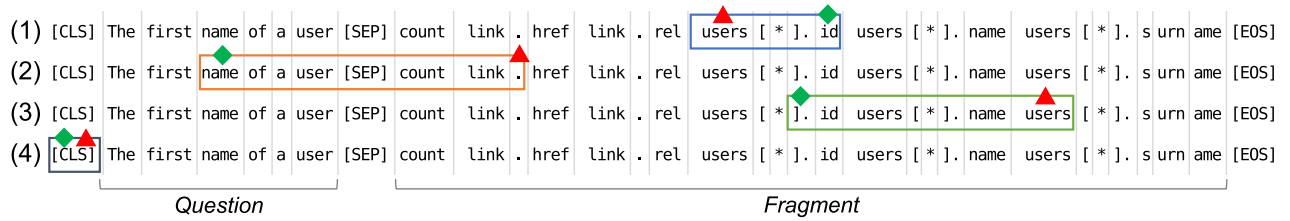


Fig. 4 presents four answer spans. The green diamond and the red triangle indicate the start and the end token of each span, respectively. The spans (1) and (2) are invalid as they are reversed and contain tokens of the question, respectively. Span (3) contains multiple Web API elements, but `users[*].name` is entirely covered; thus, (3) is valid. Span (4) points to the `[CLS]` token and is, therefore, valid

samples (9.95%) contained in the first evaluation chunk of the endpoint-discovery dataset. Note that the models CB-PM+ED and RO-PM+ED were fine-tuned with mixed samples to both semantic-search tasks. We evaluated the performance of these models on both tasks in two independent iterations.

Beforehand, we converted all QA samples into their numerical representation, i.e., into tokenized samples, by using the tokenizer described in Sect. 4.6. Similar to what we did for fine-tuning, we set a limit of three unanswerable tokenized samples for each QA sample. The tokenizer converted the 110,877 QA samples of the parameter-matching task into 328,435 tokenized samples. 129,359 (39.39%) were answerable, and 199,076 (60.61%) were unanswerable. Moreover, the fragments of these tokenized samples had between one and 126 parameter paths, with a mean of 43.24 and a median of 40 paths.

The tokenization of the other 5536 QA samples of the endpoint-discovery task resulted in 16,162 tokenized samples, with 6604 (40.86%) being answerable and 9558 (59.14%) being unanswerable. Their fragments contained between one and 78 endpoint paths, with a mean of 31.93 and a median of 30 paths.

6.2.1 Accuracy of fine-tuned models

To measure the performance of a specific fine-tuned model, we let the model predict the answer spans for all tokenized samples of a specific task and collected the ranked results for each sample by applying the steps described in Sect. 6.1. For performance reasons, we considered only the 20 highest-ranked spans for each tokenized sample and not all 512×512 possible combinations when interpreting the model's output. Afterward, we used the *top-k* metric to assess the model's accuracy: A tokenized sample was correctly predicted if its correct answer was under the k highest ranked results. The accuracy for k, also known as *Accuracy@k*, was the number of correctly predicted tokenized samples under the top-k metric divided by the total number of tokenized samples of the specific task.

We did not only measure the performance of the *entirely* fine-tuned models, i.e., after all, ten epochs of fine-tuning, but for all ten available checkpoints, which means after one to ten epochs of fine-tuning.

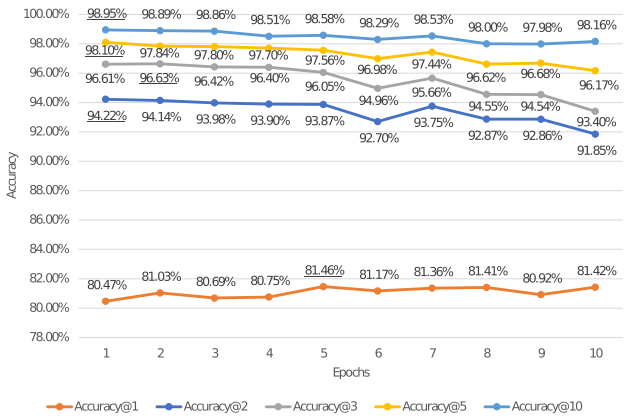
This provided us, on the one hand, insights into the fine-tuning processes, i.e., how the models evolved over ten epochs, and helped to identify side effects, like overfitting. On the other hand, with these fine-grained measurements, we could determine and select the best checkpoint for each model for subsequent tests, e.g., the robustness analysis, and the later utilization of a model as part of the search engine.

In the following, we first discuss the evaluation results of the models we fine-tuned to a specific task and afterward present the results of the models fine-tuned to both tasks.

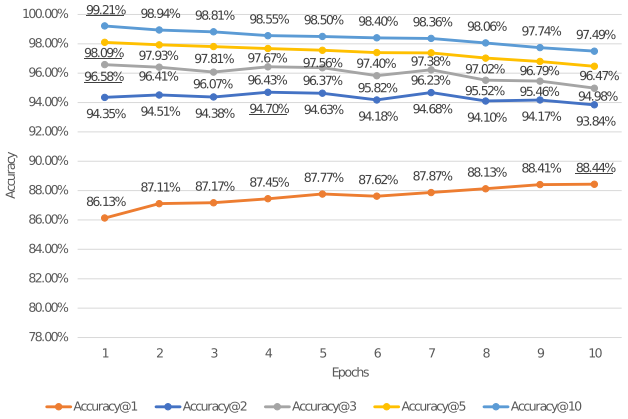
Figure 5 shows the accuracy for $k = 1, 2, 3, 5$, and 10 for all ten checkpoints of the four models CB-PM, CB-ED, RO-PM, and RO-ED.

Already one epoch of fine-tuning was sufficient to adjust all four base model instances to their designated task so the resulting models CB-PM, RO-PM, CB-ED, and RO-ED achieved an accuracy of more than 80% for $k = 1$. All models could preserve or even improve this value over the following epochs, except RO-PM, whose accuracy dropped under 80% after the seventh epoch.

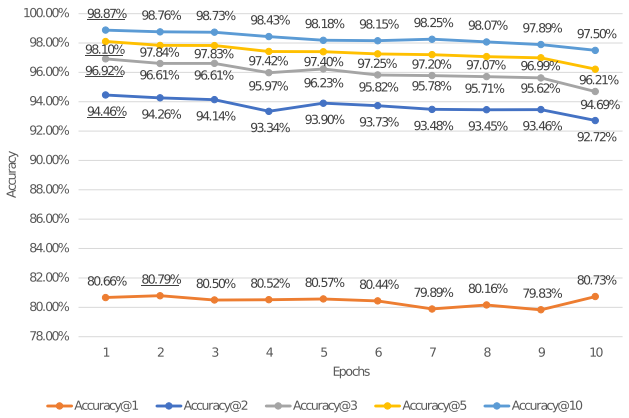
In general, we noticed that the models CB-ED and RO-ED, which we fine-tuned exclusively to the endpoint-discovery task, performed better for $k = 1$ than their counterpart models CB-PM and RO-PM, which we fine-tuned to the parameter-matching task. While CB-PM and RO-PM achieved their highest accuracies for $k = 1$ after the fifth epoch (81.46%) and second epoch (80.79%), respectively, the highest accuracies of the models CB-ED and RO-ED were roughly 7% higher, with 88.44% and 87.80%, respectively, after ten epochs of fine-tuning. This result was unexpected since we fine-tuned the models for the endpoint-discovery task with a set of tokenized samples that was twenty times smaller compared with the set used for fine-tuning the other models to the parameter-matching task. Moreover, CB-PM and RO-PM had their highest accuracies after five and two epochs of fine-tuning, respectively. The models, fine-tuned to the endpoint-discovery task, were steadily improved with each additional epoch and achieved their highest accuracy after the final



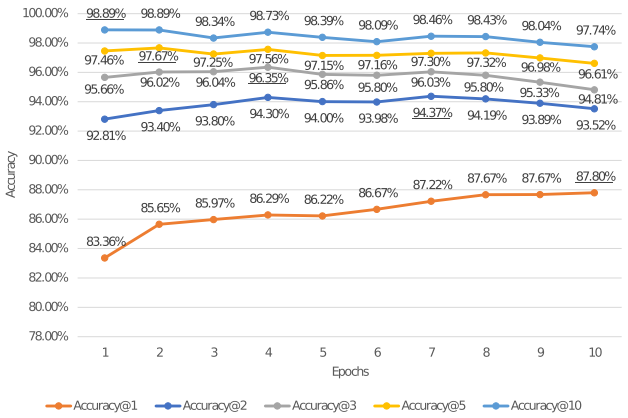
(a) CB-PM applied to parameter matching



(b) CB-ED applied to endpoint discovery



(c) RO-PM applied to parameter matching



(d) RO-ED applied to endpoint discovery

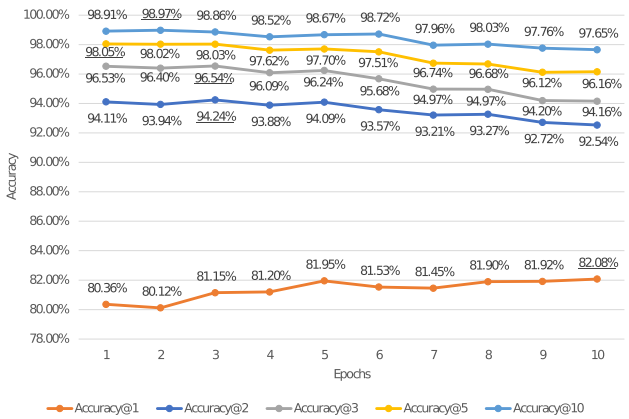
Fig. 5 Accuracy of the models CB-PM, RO-PM, CB-ED, and RO-ED for k = 1, 2, 3, 5, and 10 after one to ten epochs of fine-tuning

epoch of fine-tuning. This suggests it might be worth continuing fine-tuning these models for another few epochs for even better results.

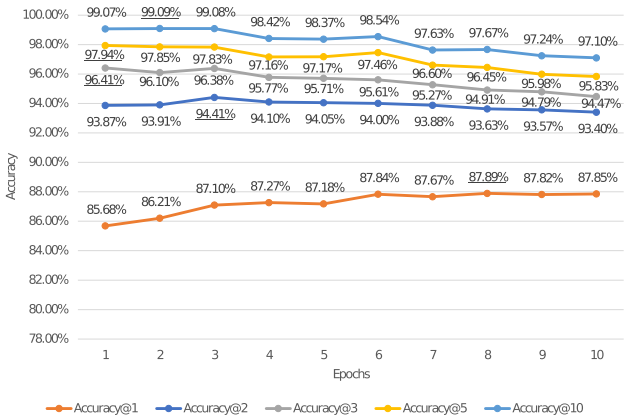
Another interesting aspect was that the fine-tuned models relying on CodeBERT performed slightly better on both tasks for $k = 1$ than those resulting from fine-tuning RoBERTa. However, the differences between the highest accuracy values of both variants, i.e., CB-PM with 81.46% compared with RO-PM with 80.79% and CB-ED with 88.44% versus RO-ED with 87.80%, were with less than 1% very marginal.

The accuracy values for $k = 2, 3, 5$, and 10 followed a similar pattern in all four models: The highest accuracy values for these ks were achieved between the first and the fourth epoch of fine-tuning, except for RO-ED, which had the highest accuracy for $k = 2$ after the seventh epoch. Then, the values started slightly to drop over the remaining epochs, which suggests minimal overfitting while fine-tuning.

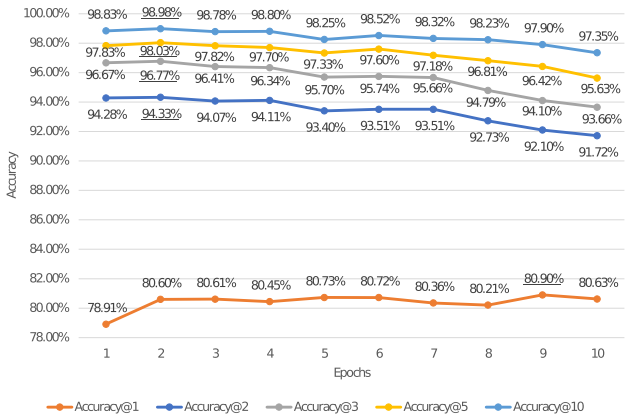
The measured performance of the CB-PM+ED and RO-PM+ED models, which we fine-tuned with mixed samples to both tasks, was almost similar to the performance of the four models that we fine-tuned to specific tasks: Fig. 6 reveals that the endpoint-discovery task performed better on the CB-ED model, i.e., the model that we fine-tuned exclusively to this task, than on the CB-PM+ED model. In detail, the CB-PM+ED model achieved its highest accuracy of 87.89% for $k = 1$ after eight epochs for this task, whereas the CB-ED model performed 0.55% better. The parameter-matching task, however, yielded a slightly higher accuracy of 82.08% for $k = 1$ when performed on the CB-PM+ED model compared with the CB-PM model (81.46%). The parameter-matching task seems to benefit from these few additional endpoint-discovery task samples used for fine-tuning the CB-PM+ED model. However, from the perspective of the endpoint-discovery task, the mix of samples decreased the performance of the CB-PM+ED model when applied to this task. We assume this might have something to do with the stringent syntax of



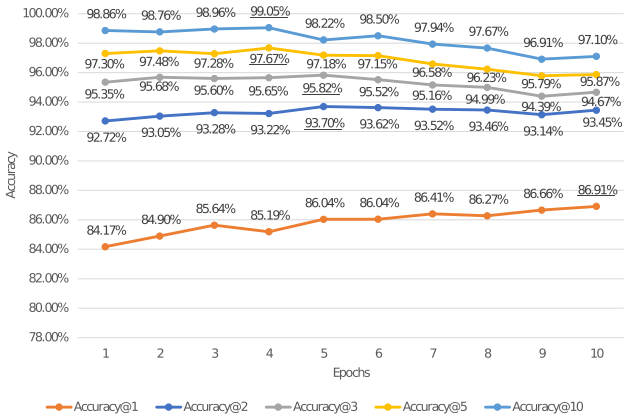
(a) CB-PM+ED applied to parameter matching



(b) CB-PM+ED applied to endpoint discovery



(c) RO-PM+ED applied to parameter matching



(d) RO-PM+ED applied to endpoint discovery

Fig. 6 Accuracy of the models CB-PM+ED and RO-PM+ED for k = 1, 2, 3, 5, and 10 after one to ten epochs of fine-tuning

endpoint paths in the QA samples of the endpoint-discovery task. The path of a parameter solely consists of entity names plus special characters for indicating arrays and path segment delimiters, e.g., `users[*].name`. The path of an endpoint also encompasses entity names for the segments of the URI path, but the last segment of the endpoint path is always the HTTP verb, e.g., `GET` in `users.{userId}.get`. Fine-tuning the model with a large number of samples that do not follow this endpoint path syntax, in particular having no HTTP verb in their paths, might have the effect that the model pays less attention to the HTTP verb of an endpoint when matching it with a description, which leads to inaccurate results. The parameter-matching task, however, might benefit from these few additional samples of the endpoint-discovery task as these samples contribute further entity-description pairs. Due to their relatively small number, the contained HTTP verbs might not disturb the overall performance of the parameter-matching task. However, with only approximately 0.50% differences between the highest accuracy values for both tasks in the different models, it is difficult to support this assumption. Nonetheless, we recommend considering these findings when extending the approach to other, i.e., future semantic-search tasks: A task that uses a task-specific syntax for Web API elements might perform better on a model that has been fine-tuned exclusively to this task. In contrast, tasks with a more generic, i.e., task-independent, syntax might benefit from fine-tuning with mixed samples.

By comparing the accuracy values of CB-PM+ED and RO-PM+ED, we could again observe that the CodeBERT model was the better base-model candidate. The CB-PM+ED model, which was a fine-tuned CodeBERT model, outperformed the RO-PM+ED model, relying on RoBERTa. Nevertheless, the differences between these highest accuracy values were minimal again. While CB-PM+ED yielded its highest accuracy of 82.08% for $k = 1$ on the parameter-matching task after the tenth epoch of fine-tuning, the highest accuracy of the RO-PM+ED model for $k = 1$ was 1.18% less, with 80.90% after the ninth epoch. Similarly, the difference between the highest

accuracy values for the endpoint-discovery task was only 0.98%. The CB-PM+ED achieved its highest accuracy of 87.89% for $k = 1$ after the eighth epoch of fine-tuning, while RO-PM+ED yielded 86.91% after the final epoch.

With these marginal differences in all model configurations, it is difficult to find an explanation for this result and to nominate one of the two base models as the clear winner. It seems that neither the large amount of pre-training data of RoBERTa nor the pre-training with PL, i.e., syntax language, in the case of CodeBERT, could yield a decisive advantage over the other. To achieve an even better performance, i.e., higher accuracy, on these semantic-search tasks, we should consider fine-tuning a base model that has been pre-trained to the language domain of Web APIs, such as ServiceBERT [35], which is, unfortunately, to the best of our knowledge, not publicly available.

As part of our future work, we therefore plan to pre-train a base model similar to Wang et al. [35]. Although the authors of RoBERTa [16] could not recommend NSP as a pre-training objective, we believe that a modified version of NSP as a pre-training method could improve the performance of both as well as future semantic-search tasks. Instead of presenting pairs of following or not-following sentences to the model, which is the original NSP procedure (see Sect. 2.1.1), we would feed combinations of Web API element paths, i.e., syntax, and their NL descriptions into the model. As a training objective, the model has to predict whether a description belongs to a Web API element, which is similar to the fine-tuning objective of the QA downstream task.

The accuracy value of the best-performing model CB-ED on the endpoint-discovery task is comparable with or even better than the performance of state-of-the-art approaches that we reviewed in Sect. 2.2: With 88.44% for $k = 1$ and 99.21% for $k = 10$, the CB-ED model achieved a similar accuracy on the endpoint-discovery task compared with the model proposed in [36], which covered 1127 Web APIs and 9004 endpoints and yielded an accuracy of 91.13% for $k = 1$ and 97.42% $k = 10$. Our model even outperformed the larger model of [36] that covered 9040 Web APIs with 49,083 endpoints and achieved an accuracy of 78.85% for $k = 1$ and 89.69% for $k = 10$. Moreover, in contrast to their model architecture, our approach is not limited to a pre-defined set of Web APIs and their endpoints but can also process endpoints that were unknown when the model was compiled. Ultimately, it is essential to mention that these comparisons must be interpreted with caution because different datasets were used to evaluate each approach.

Unfortunately, the state-of-the-art approach SMAT [40], which we reviewed in Sect. 2.3, used another evaluation metric as we did. As a consequence, we could not compare the performance of SMAT with the CB-PM+ED model on the parameter-matching task.

For all subsequent tests, including the following manual analysis of incorrect samples and the robustness analysis, we chose the model and the checkpoint yielding the *best* accuracy for each task. We chose the CB-ED model after the tenth epoch of fine-tuning (88.44% for $k = 1$) for the endpoint-discovery task and the CB-PM+ED model after the fifth epoch of fine-tuning (81.95% for $k = 1$) for the parameter-matching task. Although CB-PM+ED yielded a slightly better accuracy of 82.08% for $k = 1$ after the tenth epoch compared with its accuracy after the fifth epoch, we assessed the latter-mentioned checkpoint as the better choice: Compared with the fifth epoch, the accuracy was after the tenth epoch of fine-tuning with a plus of 0.13% only marginally higher for $k = 1$, but the accuracy values for $k = 2, 3, 5$, and 10 dropped by more than 1%.

6.2.2 Analysis of incorrect answers

In all eight graphs in Figs. 5 and 6, we noticed a relatively large step between accuracy values for $k = 1$ and $k = 2$, while values for $k = 2, 3, 5$, and 10 were closer. This *might* indicate that for the majority of incorrectly predicted answers for $k = 1$ the respective model was confronted with a binary decision after it had excluded all other choices, but it chose the wrong Web API element.

We manually analyzed tokenized samples answered incorrectly to investigate the causes of incorrect decisions. We randomly picked 100 tokenized samples for each task that the respective model, i.e., CB-PM+ED and CB-ED, had answered incorrectly for $k = 1$. We analyzed the properties of each of these tokenized samples manually. We tried to identify patterns in their questions and fragments that seemed to impede the model from predicting the correct answer.

Among these 200 tokenized samples, we could identify four patterns that occurred in samples of both tasks, which seem to be a general limitation of our approach. Additionally, we identified three patterns that were exclusively present in the samples of the endpoint-discovery task and another task-specific pattern that occurred in the samples of the parameter-matching task.

Note that the following pattern definitions and the number of occurrences slightly differ from those presented in our previous work [15]: On the one hand, we adjusted some pattern definitions to distinguish them better from each other and detect them more unequivocally in a tokenized sample. On the other hand, in this work, we found a model configuration, namely CB-PM+ED, that performed slightly better on the parameter-matching task than the CB-PM model we analyzed in our previous work. Therefore, we had to repeat this manual analysis with another 100

randomly picked tokenized samples for the parameter-matching task.

In particular, we labeled each of these 200 tokenized samples with the pattern that, in our subjective judgment, was the most likely cause of an incorrect answer. Nevertheless, we did not conduct any further experiments to confirm that the identified issues and the related patterns were indeed causing incorrect answers.

We identified the following four patterns in the tokenized samples of both tasks:

Domain-specific descriptions (DS): In seven samples of the parameter-matching and 20 samples of the endpoint-discovery task, we found that the NL description was too domain-specific. A developer unfamiliar with the respective domain would probably have difficulty choosing the correct parameter or endpoint.

Missing context in description (MC): In 72 samples of the parameter-matching task and another 22 samples of the endpoint-discovery task, we observed that there was not enough context about the addressed parent entities in the NL descriptions of the questions. As a result, multiple Web API elements, i.e., parameters or endpoints, came into consideration. Examples of this were descriptions like "Gets or sets the delivery URL.", where a parameter named `DeliveryUrl` could be found multiple times within the original schema attached to different parent entities, e.g., `MediaStreams[*].DeliveryUrl` and `MediaSources[*].MediaAttachments[*].DeliveryUrl`.

Among all 94 tokenized samples that lacked context in descriptions, we furthermore identified four measures that the models took when being confronted with such a situation:

In 29 cases, the respective model chose the only visible Web API element in the fragment that matched the description since the correct answer lay outside the fragment. In another 23 cases, multiple elements in the fragment apparently matched the description. Therefore, the model guessed the answer. In three cases, the model identified another element that seemed to match the description even better, although the correct answer was in the fragment. In the remaining 39 cases, the model excluded all Web API elements, including the correct answer, and predicted the sample as unanswerable. This happened especially with short descriptions, e.g., "Gets or sets the type", which mentioned generic artifacts like "type", "identifier", "status", and "name".

Invisible parameter or endpoint (IV): In another ten cases in the parameter-matching task and 12 cases in the endpoint-discovery task, the correct answer lay outside the presented fragment, i.e., was invisible to the model. Instead of predicting such a tokenized sample as unanswerable, the respective model chose another visible Web API element in the fragment that also seemed to match the description. At first glance, these 22 cases appear similar to the 29 cases of the MC pattern. However, in contrast to the latter, missing context in the description was not the cause but merely the invisibility of the Web API element being the correct answer. In detail, we found that if the invisible element had been in the fragment of the respective tokenized sample, the model would probably have predicted it as the answer since it would have matched the description even better than the chosen element.

Not understandable description (NU): In nine tokenized samples of the parameter-matching task and another 16 samples of the endpoint-discovery task, we observed that the models could not understand the descriptions and, therefore, predicted incorrect answers. For these 25 cases, we could not find any explanation for making an incorrect prediction.

Moreover, we identified one additional pattern in the incorrect samples of the parameter-matching tasks and three further patterns for the endpoint-discovery task:

Missing context in schema (MCS): In two tokenized samples of the parameter-matching task, the descriptions mentioned entities, e.g., "payment", that might be part of a Web API endpoint but did not occur in the schemas and were invisible to the model. These missing entity names in syntax seemed to confuse the model.

Missing information in description (MI): In three cases of the endpoint-discovery task, the descriptions were too sparse. While these descriptions mentioned all necessary entities to make a prediction, they omitted hints regarding the HTTP verb, path parameters, etc.

Defective description (DD): In another three cases of the endpoint-discovery task, the descriptions seemed to be defective. This might result from removing URIs and sentence truncation during data preparation (see Sect. 4.2). As a consequence, essential keywords were missing in the descriptions, impeding the model from making correct predictions.

Antipatterns in endpoint design (ED): In contrast to previously mentioned patterns, it was not only the NL description or the visibility of a Web API element in the fragment that impeded the model from making correct predictions: We identified 24 tokenized samples of the endpoint-discovery task in which poor design in presented endpoints, i.e., in the syntax, was the reason for incorrect answers.

In detail, we noticed 11 samples in which the Web API designers chose HTTP verbs for endpoints that misled the model as their semantics did not match the respective NL description. In these samples, developers of Web APIs confused the meaning of the HTTP verbs `PUT` and `POST` and misused these two HTTP verbs. Examples were descriptions like "Create a manual journal", which referred

to the endpoint `manualjournals.put`, but the model predicted `manualjournals.post` as the correct answer. Furthermore, the use of `POST` instead of `GET` for tunneling a read operation also confused the model, as well as the opposite case where `GET` was used instead of `POST` for unsafe controlling operations. Remarkably, in almost all 11 cases, the model tried to choose an endpoint with an HTTP verb that, from the perspective of common RESTful API design rules [43], better suited the NL description.

In another eight cases, the API designer used singular nouns for collections and, vice versa, plural nouns for document resources, which can be considered an antipattern according to [4, 43].

Tables 3 and 4 list the number of occurrences of patterns we identified in the tokenized samples of the parameter-matching and endpoint-discovery task, respectively. All 200 tokenized samples were incorrect for $k = 1$, but the correct answers could be found on another rank. For the following discussion, we grouped these numbers by the rank of their correct answers in both tables.

Table 3 Occurrences of the patterns Missing Context in Description (MC), Invisible Parameter (IV), Not Understandable Description (NU), Domain-Specific Descriptions (DS), and Missing Context in Schema (MCS) in tokenized samples of the *parameter-matching task*, grouped by rank of correct answer

Rank	#MC	#IV	#NU	#DS	#MCS	Total
$r = 2$	50	6	3	2	2	63
$3 \leq r < 5$	14	3	3	2	0	22
$5 \leq r < 10$	4	1	2	2	0	9
$r \geq 10$	4	0	1	1	0	6
Total	72	10	9	7	2	100

Table 4 Occurrences of the patterns Antipatterns in Endpoint Design (ED), Missing Context in Description (MC), Domain-Specific Descriptions (DS), Not Understandable Description (NU), Invisible Parameter (IV), and Missing Information in Description (MI), and Defective Description (DD) in tokenized samples of the *endpoint-discovery task*, grouped by rank of correct answer

Rank	#ED	#MC	#DS	#NU	#IV	#MI	#DD	Total
$r = 2$	12	11	11	8	8	1	0	51
$3 \leq r < 5$	3	6	1	2	1	0	1	14
$5 \leq r < 10$	4	2	1	1	1	0	2	11
$r \geq 10$	5	3	7	5	2	2	0	24
Total	24	22	20	16	12	3	3	100

In general, in both tasks, the correct answer was on the second rank for the majority of incorrectly predicted samples. This observation is aligned with the large step between accuracy values for $k = 1$ and 2 that we noticed in all eight graphs in Figs. 5 and 6. For instance, an invisible parameter or endpoint (IV) led in both tasks to an incorrect answer on the first rank. However, for the majority of these tokenized samples to which this pattern applied, the respective model correctly predicted the samples as unanswerable as the second choice. It seems that the respective model tends to exclude all options, except the wrong one, rather than guessing Web API elements that do not match the description in the question. The score for the prediction that the sample is unanswerable is higher than the score for any other Web API element, except for the incorrect answer.

Similarly, the correct answer could be found on the second rank for more than half of the tokenized samples of the endpoint-discovery task that the model did not understand (NU and DS).

Although the model tended to follow the principles and best practices of REST when choosing an endpoint, it chose the correct answer on the second rank for 12 of these 24 tokenized samples suffering from poor endpoint design. Moreover, we observed that the model had predicted nine of these 12 tokenized samples as unanswerable on the first rank. In these nine cases, the model could not associate any endpoint of the respective fragment with the description without conflicting principles and best practices of REST. For the answer on the second rank, the model was more liberal toward an imperfect endpoint design, though.

The problem of missing context in the description (MC) appeared to be the most common cause of an incorrect answer in the parameter-matching task (72 cases) but also affected 22 tokenized samples of the endpoint-discovery task.

At first glance, it seems that this problem can be solved by complementing descriptions with further details about parent entities, which should be considered in the later application of the approach as a search engine. However, this pattern also revealed a limitation of our approach: Neither in the QA samples used for fine-tuning nor in the evaluation did we consider that a QA sample might have alternative answers besides the main answer. A QA sample would have alternative answers if multiple Web API elements in its paragraph shared the same or at least similar descriptions as in its question. These Web API elements sharing the same description would be the alternative answers.

On the one hand, we did not consider this case for fine-tuning due to a technical limitation of QA applied to a BERT model: To convert the prediction of a BERT model into an answer span, we choose the start and end

combination that has the highest score (see Sect. 6.1). Consequently, a single prediction cannot result in multiple spans and independent answers on the same rank. The same limitation applies to the fine-tuning process. Each QA sample used to fine-tune the model must specify one start and end token of the span containing the answer but cannot specify multiple answer spans. Nevertheless, we can fine-tune the model with many independent QA samples sharing the same question but having different answers, which is what we have done in our approach.

On the other hand, we assumed that in Web APIs, any Web API element in a syntax structure, i.e., paragraph, does not only have a unique syntax, expressed through a distinct path, but also a unique purpose. Hence, every unique Web element should have its unique description. We do accept multiple suggestions in the form of a ranked list of Web API elements, which the model predicts. However, we do not accept multiple correct answers, which means that only one of these suggested elements could be the correct answer. The manual analysis of incorrectly answered tokenized samples has shown that there were cases in which different Web API elements in the same paragraph shared the same or a similar description, which caused a decision problem. We consider this a quality issue of some QA samples in the dataset as, in most cases, the description was too sparse and did not contain enough context. Nevertheless, it is interesting that the respective model tended to exclude all Web API elements when confronted with a too-short, not-meaningful description, which is, from our point of view, a rational and legitimate decision.

6.3 Robustness

The manual analysis of incorrect samples in Sect. 6.2.2 revealed that inaccurate and sparse NL descriptions could lead to inconclusive or even incorrect answers because many Web API elements might come into consideration. However, NL descriptions are not only subject to strong variations in terms of verbosity and abundance of embedded information. In the later application of the fine-tuned models as a search engine, we must expect variations in wording and structure in the NL descriptions of the input queries. A developer who uses the search engine might be unaware of how to prompt the model, i.e., how to formulate optimal queries to obtain accurate results.

To test the robustness of our fine-tuned models CB-PM+ED and CB-ED, we let both models predict answer spans of task-specific QA samples with rephrased NL descriptions and compared the measured accuracies with those that resulted from processing the original QA samples. We defined three test cases. Each case addressed a specific variation of the original NL description. In detail, we wanted to know whether the performance, i.e., the measured accuracy, of a fine-tuned model is negatively affected: (1) if we translate the NL description into a question, (2) if we replace important keywords that also occur in the answer, i.e., in syntax, with synonyms in the NL description, and (3) if we change the word order or sentence structure of the original description. We assume these cases might also occur in practical applications, i.e., when a model is used as a search engine.

We used statistical hypothesis testing to support our findings as we could conduct these tests only with a limited number of QA samples since we had to rephrase NL descriptions manually. More precisely, we applied binomial testing [44] to test the statistical significance of deviations between an expected distribution that resulted from processing the original QA samples and distributions resulting from processing rephrased QA samples. Furthermore, we defined the model's accuracy for a specific k as the probability of success π.

For each case, we formulated one null hypothesis, stating that the respective modification of the NL description does not affect, or even increases the accuracy for a specific k, i.e., $\pi \geq \pi_0$, and one alternative hypothesis, claiming that the modification decreases the model's accuracy for a specific k, i.e., $\pi < \pi_0$. They are listed in Table 5. While this table lists only three hypotheses, we tested these hypotheses individually for each model and different ks, namely $k = 1, 2, 3, 5$, and 10, with independent sample sets. Moreover, we set a targeted significance

Table 5 Null hypotheses with their alternative hypotheses for the three different cases

Case	Null Hypothesis	Alternative Hypothesis
Questions	$\mathbf{H}_0^1$: Translating descriptions into questions does not affect, or increases the model's accuracy for a specific k, i.e., $\pi \geq \pi_0$	$\mathbf{H}_1^1$: Translating descriptions into questions decreases the model's accuracy for a specific k, i.e., $\pi < \pi_0$
Synonyms	$\mathbf{H}_0^2$: Replacing keywords with synonyms does not affect, or increases the model's accuracy for a specific k, i.e., $\pi \geq \pi_0$	$\mathbf{H}_1^2$: Replacing keywords with synonyms decreases the model's accuracy for a specific k, i.e., $\pi < \pi_0$
Word order/ sentence structure	$\mathbf{H}_0^3$: Changing the word order or sentence structure does not affect, or increases the model's accuracy for a specific k, i.e., $\pi \geq \pi_0$	$\mathbf{H}_1^3$: Changing the word order or sentence structure decreases the model's accuracy for a specific k, i.e., $\pi < \pi_0$

level of $\alpha = 0.05$ that we adjusted to $\alpha = 0.05/15 = 0.00\overline{3}$ using the Bonferroni correction method [45] as we tested 15 hypotheses per model. If the calculated p-value was less than this adjusted significance level, we rejected the respective null hypothesis and accepted its alternative for a specific case, model, and k.

We randomly picked 50 QA samples for each task from the second evaluation chunk of the respective task-specific dataset. Then, we created three copies of each QA sample and manually rephrased the original NL description:

Questions: We used the first copy of each original QA sample to translate the NL description into a question. In detail, we prefixed the descriptions of 25 QA samples of the parameter-matching task with the phrase "What is the property which contains" and the other 25 descriptions with "Which is the parameter that provides". Similarly, we prefixed the descriptions of 25 QA samples of the endpoint-discovery task with "What is the endpoint that" and the descriptions of the remaining 25 QA samples with "Which operation". If necessary, we changed the word order in the remaining sentence and fixed potential grammatical errors so that the resulting description was a question. We changed the original description of a QA sample of the parameter-matching task from "The search algorithm specified for the study." to "What is the property which contains the search algorithm specified for the study?", for instance.

Synonyms: In the second copy of each original QA sample, we individually replaced keywords that occurred in both the NL description and the syntax of the correct answer with synonyms. "The product's unit price for the order" was changed to "The article's cost per entity for the purchase", for instance, so that none of the words in the description was contained in the related parameter `orders[*].products[*].unitPrice`.

Word order: We changed the sentence structures and reordered words in the description of the third copy but without altering the meaning. We modified the description "Returns a part by part number." to "Given a part number, it returns a part.", for instance.

We took care that the original QA samples and their rephrased versions did not exceed the maximum input size of 512 tokens. This ensured comparability between the predicted answers of the different QA sample versions. For each QA sample, there was only one tokenized sample, and the correct answer could not move from one to another sample due to rephrasing, which avoids side effects like the pattern of an invisible parameter or endpoint (IV, see Sect. 6.2).

After rephrasing the NL descriptions in the created copies, we let the models process the rephrased as well as the original QA samples and measured for each sample set the accuracies for $k = 1, 2, 3, 5$, and 10 that the respective model achieved. Table 6 lists these measured accuracies plus the p-values for the rephrased QA sample sets that we

Table 6 Accuracies that the models CB-PM+ED and CB-ED yielded on the original and rephrased sample sets plus the calculated p-values. P-values in bold are lower than the adjusted significance level

QA sample set	CB-PM+ED			CB-ED		
	k	*Accuracy*	*p-value*	*k*	*Accuracy*	*p-value*
Original	1	82%		1	86%	
	2	90%		2	90%	
	3	94%		3	90%	
	5	94%		5	90%	
	10	96%		10	96%	
Questions	1	78% (-4%)	0.281	1	86%	0.562
	2	84% (-6%)	0.122	2	92% (+2%)	0.75
	3	90% (-4%)	0.179	3	92% (+2%)	0.75
	5	94%	0.584	5	92% (+2%)	0.75
	10	98% (+2%)	0.870	10	96%	0.6
Synonyms	1	52% (-30%)	**<0.001**	1	22% (-64%)	**<0.001**
	2	58% (-32%)	**<0.001**	2	46% (-44%)	**<0.001**
	3	62% (-32%)	**<0.001**	3	50% (-40%)	**<0.001**
	5	74% (-20%)	**<0.001**	5	58% (-32%)	**<0.001**
	10	90% (-6%)	0.049	10	76% (-20%)	**<0.001**
Word order/ sentence structure	1	78% (-4%)	0.281	1	88% (+2%)	0.719
	2	90%	0.569	2	90%	0.569
	3	92% (-2%)	0.353	3	90%	0.569
	5	94%	0.584	5	90%	0.569
	10	96%	0.6	10	96%	0.6

computed as follows: We used the measured accuracy for a specific k resulting from processing the original QA samples as the expected probability of success π_0 to calculate the p-value for each rephrased QA sample set with

$$p = \sum_{i=0}^{c} \binom{n}{i} \pi_0^i (1 - \pi_0)^{n-i}$$

, where n is the number of QA samples and c the number of correctly predicted QA samples.

Translating the original NL description into questions, changing the sentence structure, and reordering words had almost no effect on the accuracy of both models. While the accuracy values of the CB-PM+ED model for $k = 1, 2,$ and 3 dropped slightly for QA samples whose descriptions were translated into questions, the CB-ED model could sustain its accuracy for $k = 1$ and even increase the accuracies for $k = 2, 3,$ and 5 for samples of this case. Similarly, changing the sentence structure and reordering words let the accuracy values of CB-PM+ED for $k = 1$ and 3 drop by a few percent, while CB-ED could increase its accuracy for $k = 1$ by 2%. The calculated p-values exceeded our adjusted significance level ($\alpha = 0.00\overline{3}$). Therefore, we retained the null hypotheses $\mathbf{H}_0^1$ and $\mathbf{H}_0^3$ and rejected the alternatives $\mathbf{H}_1^1$ and $\mathbf{H}_1^3$ for both models and all ks. This means that translating the original description into questions, changing the sentence structure, and reordering words did not significantly decrease the model's accuracy.

The use of synonyms, however, impeded both models from making correct predictions. The accuracy values of the CB-PM+ED model for $k = 1, 2,$ and 3 dropped by roughly 30% and by 20% for $k = 5$. The drop in the accuracies of the CB-ED model was even worse as the model achieved only an accuracy of 22% for $k = 1$, which was 64% less than the baseline accuracy achieved with the original samples. Also, the accuracy values for $k = 2, 3, 5,$ and 10 dropped by 44%, 40%, 32%, and 20%, respectively.

For both tasks, we noticed that the models were rather able to associate synonyms of common words, like "enabled" with "activated", "name" with "label", and "secret" with "confidential token", than domain-specific words and their synonyms, like "regions" and "areas". This may be explained by the fact that common words and combinations of synonyms, e.g., "activate" in NL description and "enable" in syntax, occur more frequently in different OpenAPI documentation and, therefore, also fine-tuning samples than combinations of synonymous domain-specific words. This is especially true in endpoint descriptions where different NL verbs are used to describe the behavior of a limited number of syntactically standardized HTTP verbs so the model can learn various synonyms for the same HTTP verb. For the endpoint-discovery task, we observed that in 28 of 50 cases, the CB-ED model was able to successfully map a verb in the description, e.g., "retrieve" or "generate", to the respective HTTP verb, e.g., `GET` or `PUT`, and suggested endpoints with the identified HTTP verb.

However, we also noticed that multiple Web API elements within the same paragraph often have similar syntax names with synonymous words, although addressing completely different semantic concepts. This confused the models. For instance, after replacing the keyword "kind" in the description "The kind of resources that are supported in this SKU." with "sort", the CB-PM+ED model suggested `value[*].resourceType` be the correct answer instead of `value[*].kind`. Moreover, in this sample, the model seemed to focus on the keyword "resources" since the word "sort" was not present in any Web API element in the paragraph. From this and other samples, we conclude that in case of doubt, i.e., multiple Web API elements apparently match a given word, the respective model seeks other keywords to commit to an answer. If a paragraph consists of semantically and syntactically distinct Web API elements, the chance of identifying the correct answer is higher, even if described with synonyms.

For both models and all ks, except $k = 10$ for CB-PM+ED, we rejected the null hypothesis $\mathbf{H}_0^2$ and accepted the alternative $\mathbf{H}_1^2$ as the calculated p-values were lower than the adjusted significance level.

7 Applicability

After evaluating the fine-tuned models, we discuss their application as a search engine. In detail, we propose an architecture for a search engine for the semantic-search tasks we cover with our approach. Furthermore, we comment on possible extensions and further applications of this search engine and, in general, our approach.

7.1 Search engine architecture

As the two semantic-search tasks yielded their best performance on different models, we suggest using two different search engine instances, one with the CB-PM+ED model for the parameter-matching task and another with the CB-ED model for the endpoint-discovery task. Nevertheless, the architecture for both instances is the same.

Figure 7 illustrates the architecture of the search engine. Several components that we already used for fine-tuning and evaluation can be reused. While the processes for tokenization and output interpretation are mostly the same as described in Sect. 4, only the data preparation process slightly differs: For fine-tuning and evaluation, we

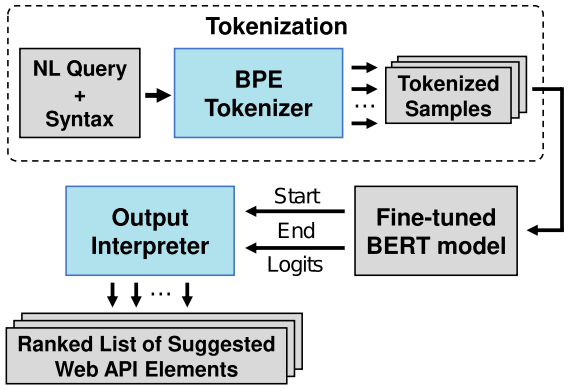


Fig. 7 The architecture of the search engine using a fine-tuned BERT model

prepared datasets with QA samples that we extracted from 2321 OpenAPI documentation. The first step of using the search engine, however, is that the developer, i.e., the user of the system, inputs a syntax structure, e.g., a payload schema or a list of endpoints, and an NL query. Then, the search engine transforms, i.e., serializes, the syntax structure into a linear list of Web API element paths. Afterward, the tokenizer splits the query and the list of Web API element paths into tokens and converts each token into its numerical representation. Similar to the tokenization of QA samples for fine-tuning and evaluation, we have to expect multiple tokenized samples if the input sequence exceeds the limit of 512 tokens. In the next step, the respective model processes all tokenized samples and yields one prediction for each sample. Finally, we input these predictions into the output interpreter (see Sect. 6.1) to obtain a ranked list of suggested Web API elements for each tokenized sample.

For performance reasons, limiting the number of predictions the output interpreter should process might be reasonable. Or even limit the number of results per tokenized sample and present only the highest-ranked results for each sample to the developer, who reviews the suggestions of the search engine. Nevertheless, as the accuracy values for $k = 2$ of the models CB-PM+ED and CB-ED were roughly 12% and 5% higher compared with $k = 1$, respectively, we recommend not only accepting the highest-ranked result but also considering suggestions at least on the second rank.

7.2 Further applications and extensions

In our current approach, the fine-tuned models consider only the syntax of Web API elements to determine a match between an element and an NL query. For more accurate predictions in the two semantic-search tasks, as well as future tasks, it might be reasonable to construe the semantics of a Web API element not only from its syntax description but also to compare an NL query with the NL description of the element. We have not considered this additional feature in our work yet, since, to the best of our knowledge, there exists no dataset containing samples of matching NL queries and NL descriptions of Web API elements that could be used to fine-tune an additional model. Nevertheless, instead of fine-tuning another model, we could also rely on a pre-trained Sentence Transformer, like SBERT [46], to calculate the semantic distance between an NL query and an NL description of a Web API element as an additional matching criterion. As part of our future work, we will investigate whether a pre-trained SBERT model applies to the language domain of Web APIs and could be integrated into our approach.

In the current implementation, we only tag arrays and URI parameters in the path of Web API elements with special characters (see Sect. 3.2) to express their role. Especially for the parameter-matching task, it could be helpful to encode also the data type, e.g., string, number, integer, or boolean, of the parameter into paths, e.g., `users`$[*]$`.surname`$<$`str`$>$. With this additional information, the model might consider syntactic compatibility when matching a parameter with an NL description.

Furthermore, as an extension of the parameter-matching task, it is possible to add further logic to the respective search engine that allows not only the identification of parameters matching an NL query but also the mapping of entire schemas. Given a schema with output parameters, e.g., a response schema, and a schema with input parameters, e.g., a request schema, the underlying model could suggest possible pairs of output and input parameters. Technically, an algorithm would traverse the output schema and extract the NL description of each output parameter. Then, for each NL description, the model would predict ideally one parameter from the input schema that matches the respective description. To verify predicted results, we should consider the predictions of one direction, namely from output to input parameters, and the results of the other direction from input to output parameters. Accordingly, the algorithm should also iterate over the list of input parameters and let the model propose the parameter in the output schema that matches the description of the respective input parameter. Ideally, both mappings, from output to input schema and the opposite case, should be identical.

8 Conclusion

In this article, we proposed a Transformer encoder model for the semantic search in state-of-practice Web API documentation, consisting of structured syntax and natural

language descriptions for the Web API's semantics. We modeled semantic search as a question-answering task and aligned our model architecture to this task: Given a natural language query (question) and a syntax structure in the Web API documentation (paragraph), the model chooses the Web API element (answer) in the structure that matches the query best. Our approach covers two specific semantic-search tasks, namely endpoint discovery and parameter matching.

Technically, we fine-tuned different pre-trained BERT models to these semantic-search tasks. For fine-tuning and evaluation, we extracted 1,085,051 question-answering samples for the parameter-matching task and another 55,659 for the endpoint-discovery task from 2321 real-world OpenAPI documentation. We tested six fine-tuning strategies with different combinations of base models and task-specific samples. In detail, we fine-tuned the two pre-trained base models CodeBERT and RoBERTa either to one dedicated task or to both tasks. We evaluated all six fine-tuned models in terms of generalizability with with-held samples using the top-k accuracy metric. The results revealed that the fine-tuned models relying on CodeBERT performed slightly better than those resulting from fine-tuning RoBERTa. However, the differences between the highest accuracies for $k = 1$ were very marginal, with roughly 1%, so it was difficult to nominate a clear winner. Another finding was that the parameter-matching task performed better on a base model we fine-tuned to both tasks. In contrast, the endpoint-discovery task had its highest accuracy on a base model fine-tuned to this task exclusively. For the parameter-matching task, the best-performing model yielded an accuracy of 81.95% for $k = 1$ and 94.09% for $k = 2$. The best model for the endpoint-discovery task performed even better, with an accuracy of 88.44% for $k = 1$ and 93.84% for $k = 2$.

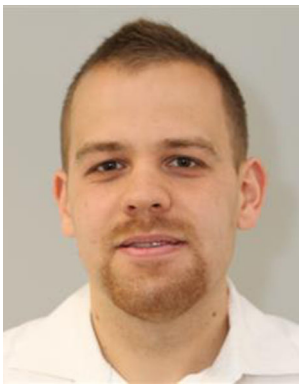
For both tasks and their best-performing models, we manually analyzed samples that were incorrectly answered. We identified patterns like missing context in the natural language description, i.e., query, that impeded the model from making correct predictions. Moreover, we tested the robustness of the best-performing models by rephrasing NL queries. The models were robust to changes in the sentence structure, but the accuracy significantly dropped when using synonyms for domain-specific words.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Jacobson, D., Brail, G., Woods, D.: APIs: A Strategy Guide. O'Reilly Media Inc., Sebastopol (2011)
2. Webber, J., Parastatidis, S., Robinson, I.: REST in Practice: Hypermedia and Systems Architecture. O'Reilly Media Inc., Sebastopol (2010)
3. Rodríguez, C., et al.: REST APIs: A large-scale analysis of compliance with principles and best practices. In: International Conference on Web Engineering (ICWE 2016). Lecture Notes in Computer Science (LNISA), vol. 9671. Springer, Berlin (2016)
4. Palma, F., et al.: Semantic Analysis of RESTful APIs for the Detection of Linguistic Patterns and Antipatterns. Int. J. Coop. Inf. Syst. **26**, 1742001 (2017). https://doi.org/10.1142/S0218843017420011
5. Aiello, M.: A challenge for the next 50 years of automated device composition. In: International Conference on Service-Oriented Computing (ICSOC 2022), pp. 635–643. Springer Nature, Cham (2022)
6. Martin, D., et al.: OWL-S: semantic markup for web services (2004). https://www.w3.org/Submission/OWL-S/. Accessed 30 Aug 2023
7. de Bruijn, J., et al.: Web service modeling ontology (WSMO) (2005). https://www.w3.org/Submission/WSMO/. Accessed 30 Aug 2023
8. Farrell, J., Lausen, H.: Semantic annotations for WSDL and XML schema (2007). https://www.w3.org/TR/sawsdl/. Accessed 30 Aug 2023
9. Cremaschi, M., De Paoli, F.: Toward automatic semantic API descriptions to support services composition. In: European Conference on Service-Oriented and Cloud Computing (ESOCC 2017), pp. 159–167. Springer, Cham (2017)
10. Cremaschi, M., De Paoli, F.: A practical approach to services composition through light semantic descriptions. In: European Conference on Service-Oriented and Cloud Computing (ESOCC 2018), pp. 130–145. Springer, Cham (2018)
11. Haupt, F., Leymann, F., Vukojevic-Haupt, K.: API governance support through the structural analysis of REST APIs. Comput. Sci. **33**, 291–303 (2018). https://doi.org/10.1007/s00450-017-0384-1
12. Sun, C., Qiu, X., Xu, Y., Huang, X.: How to fine-tune BERT for text classification. In: China National Conference on Chinese Computational Linguistics (CCL 2019), pp. 194–206. Springer, Cham (2019)
13. Liu, Y., Lapata, M.: Text summarization with pretrained encoders. In: 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pp. 3730–3740. Association for Computational Linguistics, Hong Kong (2019). https://doi.org/10.18653/v1/D19-1387
14. Feng, Z., et al.: CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics (EMNLP 2020), pp. 1536–1547.

Association for Computational Linguistics (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139
15. Kotstein, S., Decker, C.: Semantic parameter matching in Web APIs with Transformer-based question answering. In: 2023 IEEE International Conference on Service-Oriented System Engineering (SOSE), pp. 114–123. IEEE (2023). https://doi.org/10.1109/SOSE58276.2023.00020
16. Liu, Y.: *et al.* RoBERTa: a robustly optimized BERT pretraining approach (2019). https://doi.org/10.48550/arXiv.1907.11692
17. Vaswani, A., et al.: Attention Is All You Need, NIPS'17, 6000–6010. Curran Associates Inc., Red Hook (2017)
18. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. NIPS **27**, 3104–3112 (2014)
19. Borzunov, A., et al.: Training transformers together, Vol. 176 of Proceedings of Machine Learning Research, pp. 335–342 (PMLR) (2022)
20. Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 4171–4186. Association for Computational Linguistics, Minneapolis (2019)
21. Pearce, K., Zhan, T., Komanduri, A., Zhan, J.: A comparative study of transformer-based language models on extractive question answering (2021). https://doi.org/10.48550/arXiv.2110.03142
22. Segal, E., Efrat, A., Shoham, M., Globerson, A., Berant, J.: A simple and effective model for answering multi-span questions. In: 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 3074–3080. Association for Computational Linguistics (2020). https://doi.org/10.18653/v1/2020.emnlp-main.248
23. Sanh, V., Debut, L., Chaumond, J., Wolf, T.: DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter (2019). https://doi.org/10.48550/arXiv.1910.01108
24. Adhikari, A., Ram, A., Tang, R., Lin, J.: DocBERT: BERT for document classification (2019). https://doi.org/10.48550/arXiv.1904.08398
25. Scheible, R., Thomczyk, F., Tippmann, P., Jaravine, V., Boeker, M.: GottBERT: a pure German language model (2020). https://doi.org/10.48550/arXiv.2012.02110
26. Radford, A., et al.: Language models are unsupervised multitask learners (2019)
27. Celik, D., Elci, A.: Discovery and scoring of semantic web services based on client requirement(s) through a semantic search agent. DBLP **2**, 273–278 (2006). https://doi.org/10.1109/COMPSAC.2006.127
28. Fang, W.-D., Zhang, L., Wang, Y.-X., Dong, S.-B.: Toward a semantic search engine based on ontologies. IEEE Explore **3**, 1913–1918 (2005). https://doi.org/10.1109/ICMLC.2005.1527258
29. Huang, Z., Zhao, W.: Combination of ELMo representation and CNN approaches to enhance service discovery. IEEE Access **8**, 130782–130796 (2020). https://doi.org/10.1109/ACCESS.2020.3009393
30. Merin, B., Banu, W.A.: Discovering web services by matching semantic relationships through ontology. In: 2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS), pp. 998–1002 (2020). https://doi.org/10.1109/ICACCS48705.2020.9074364
31. Jalal, S., Yadav, D.K., Negi, C.S.: Web service discovery with incorporation of web services clustering. Int. J. Comput. Appl. **45**, 51–62 (2023). https://doi.org/10.1080/1206212X.2019.1698131
32. Yang, Y., et al.: ServeNet: A deep neural network for web services classification. In: 2020 IEEE International Conference on Web Services (ICWS), pp. 168–175. IEEE Computer Society, Los Alamitos (2020)
33. Yang, Y., Ke, W., Wang, W., Zhao, Y.: Deep learning for web services classification. In: 2019 IEEE International Conference on Web Services (ICWS), pp. 440–442. IEEE (2019). https://doi.org/10.1109/ICWS.2019.00079
34. Pennington, J., Socher, R., Manning, C.D.: GloVe: global vectors for word representation. In: 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1532–1543. Association for Computational Linguistics (2014). https://doi.org/10.3115/v1/D14-1162
35. Wang, X., et al.: ServiceBERT: a pre-trained model for web service tagging and recommendation. In: International Conference on Service-Oriented Computing (ICSOC 2021), pp. 464–478. Springer, Cham (2021)
36. Liu, L., Bahrami, M., Park, J., Chen, W.-P.: Web API search: discover Web API and Its endpoint with natural language queries. In: International Conference on Web Services (ICWS 2020), pp. 96–113. Springer, Cham (2020)
37. Zeimetz, T., Schenkel, R.: Sample driven data mapping for linked data and web APIs, CIKM '20, pp. 3481–3484. Association for Computing Machinery, New York. https://doi.org/10.1145/3340531.3417438 (2020)
38. Zeimetz, T., Schenkel, R.: Filipo: A sample driven approach for finding linkage points between RDF data and APIs. In: European Conference on Advances in Databases and Information Systems (ADBIS 2021), pp. 244–259. Springer, Cham (2021)
39. Shraga, R., Gal, A., Roitman, H.: ADnEV: cross-domain schema matching using deep similarity matrix adjustment and evaluation. Proc. VLDB Endow. **13**, 1401–1415 (2020). https://doi.org/10.14778/3397230.3397237
40. Zhang, J., Shin, B., Choi, J.D., Ho, J.C.: SMAT: An attention-based deep learning solution to the automation of schema matching. In: European Conference on Advances in Databases and Information Systems (ADBIS 2021), pp. 260–274. Springer, Cham (2021)
41. Jones, D.M.: Operand names influence operator precedence decisions (part 1 of 2) (2008). http://www.knosof.co.uk/cbook/oprandname.pdf. Accessed 30 Aug 2023
42. Paschali, M., Conjeti, S., Navarro, F., Navab, N.: Generalizability vs. robustness: adversarial examples for medical imaging (2018). https://doi.org/10.48550/arXiv.1804.00504
43. Masse, M.: REST API Design Rulebook, 1st edn. O'Reilly Media Inc., Sebastopol (2011)
44. Wohlin, C., et al.: Analysis and Interpretation, pp. 123–151. Springer, Berlin (2012)
45. Shaffer, J.P.: Multiple hypothesis testing. Annu Rev. Psychol. **46**, 561–584 (1995). https://doi.org/10.1146/annurev.ps.46.020195.003021
46. Reimers, N., Gurevych, I.: Sentence-BERT: sentence embeddings using siamese BERT-networks. In: 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pp. 3982–3992. Association for Computational Linguistics, Hong Kong (2019)



Sebastian Kotstein is a researcher and Ph.D. student at the Herman Hollerith Zentrum at Reutlingen University since 2018. He received his B.Sc. degree in Computer Science from Baden-Wuerttemberg Cooperative State University in 2014 and his M.Sc. in Services Computing from Reutlingen University in 2016. He is currently working on methods for (semi-) automatic integration of Web and REST APIs with deep learning and natural language processing techniques.

Christian Decker is a Computer Science professor at the Herman Hollerith Zentrum at Reutlingen University. He received his master's degree and Ph.D. in Computer Science from the University of Karlsruhe. Christian's research interest lies in the study of data appliances for the Internet of Things (IoT). He develops methods to design and engineer IoT smart data services using technologies, such as machine and statistical learning.