Enhancing REST API Testing with NLP Techniques

Myeongsoo Kim
Georgia Institute of Technology
Atlanta, Georgia, USA
mkim754@gatech.edu

Davide Corradini
University of Verona
Verona, Italy
davide.corradini@univr.it

Saurabh Sinha
IBM Research
Yorktown Heights, New York, USA
sinhas@us.ibm.com

Alessandro Orso
Georgia Institute of Technology
Atlanta, Georgia, USA
orso@cc.gatech.edu

Michele Pasqua
University of Verona
Verona, Italy
michele.pasqua@univr.it

Rachel Tzoref-Brill
IBM Research
Haifa, Israel
rachelt@il.ibm.com

Mariano Ceccato
University of Verona
Verona, Italy
mariano.ceccato@univr.it

ABSTRACT

RESTful services are commonly documented using OpenAPI specifications. Although numerous automated testing techniques have been proposed that leverage the machine-readable part of these specifications to guide test generation, their human-readable part has been mostly neglected. This is a missed opportunity, as natural-language descriptions in the specifications often contain relevant information, including example values and inter-parameter dependencies, that can be used to improve test generation. In this spirit, we propose NLPtoREST, an automated approach that applies natural language processing techniques to assist REST API testing. Given an API and its specification, NLPtoREST extracts additional OpenAPI rules from the human-readable part of the specification. It then enhances the original specification by adding these rules to it. Testing tools can transparently use the enhanced specification to perform better test case generation. Because rule extraction can be inaccurate, due to either the intrinsic ambiguity of natural language or mismatches between documentation and implementation, NLPtoREST also incorporates a validation step aimed at eliminating spurious rules. We performed studies to assess the effectiveness of our rule extraction and validation approach, and the impact of enhanced specifications on the performance of eight state-of-the-art REST API testing tools. Our results are encouraging and show that NLPtoREST can extract many relevant rules with high accuracy, and significantly improve testing tools' performance.

CCS CONCEPTS

• **Information systems → RESTful web services**; • **Software and its engineering → Software testing and debugging**.

ISSTA '23, July 17–21, 2023, Seattle, WA, United States

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598131

KEYWORDS

Natural Language Processing for Testing, Automated REST API Testing, OpenAPI Specification Analysis

ACM Reference Format:
Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. 2023. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598131

1 INTRODUCTION

The REST (REpresentational State Transfer) architectural style [9] is becoming the de-facto standard for designing and implementing modern web APIs. Developers of RESTful (or simply REST) services usually provide, together with the API, a structured document that describes how the API can be accessed. The most popular standard for these documents is OpenAPI [31], which provides a formal syntax for documenting REST APIs and allows for specifying essential information such as endpoints, HTTP methods, authentication details, and input/output information (e.g., input parameters for requests and their schema, request responses and their schema, and parameter descriptions). REST API documentation that adheres to the OpenAPI standard is called an *OpenAPI specification*. To illustrate, Figure 1 presents two sample OpenAPI specifications.

OpenAPI specifications are for the most part machine-readable, but they also let developers add, in description fields, natural-language descriptions of some aspects of an API. In fact, many OpenAPI specifications rely heavily on natural-language comments to describe additional details of operation parameters.

Although many automated testing tools have been presented in the literature that leverage the *machine-readable* part of OpenAPI specifications to guide test generation, most of these tools completely ignore the *human-readable* descriptions in these specifications. We believe that this is a major limitation of the state of the art in REST API testing, as these natural-language descriptions can contain relevant and useful information for testing.

```yaml
paths:
  /check:
  post:
    summary: Check a text
    description: >- The main feature - check a text with LanguageTool
                    for possible style and grammar issues.
    parameters:
     - name: text
       in: formData
       type: string
       description: The text to be checked. This or 'data' is required.
       required: false
     - name: data
       in: formData
       type: string
       description: The text to be checked.
       required: false
     - name: language
       in: formData
       type: string
       description: >- A language code like 'en-US', 'de-DE', 'fr'
                       or 'auto'.
       required: true
```

(a) LanguageTool [20] OpenAPI specification excerpt

```yaml
paths:
  /search:
    get:
      operationId: Web_Search
      parameters:
       - name: Accept-Language
         in: header
         type: string
         description: A comma-delimited list of one or more languages
                      to use for user interface strings.
       - name: count
         in: query
         type: integer
         format: int32
         description: The maximum value is 50.
```

(b) Bing Web Search OpenAPI specification excerpt

Figure 1: Sample OpenAPI specification fragments to illustrate rule extraction from human-readable descriptions.

Consider, for instance, the `description` field associated with the parameter `language` in lines 21–22 of Figure 1(a), which provides examples of valid values (i.e., language codes) that the parameter can take. This information can be leveraged by a testing tool to create requests with valid parameter values. Conversely, testing tools that ignore this information often generate requests with random values for `language`, which is likely to result in trivial failures. Figure 1(b) shows two other forms of useful information in parameter descriptions: *parameter constraints* and *parameter formats*. For example, the description in line 15 specifies the constraint that parameter `count` must have a maximum value of 50, while the description for `Accept-Language` (lines 9–10) states that values for that parameter must be a comma-delimited list. This additional information can clearly help testing tools generate more effective tests.

Natural-language descriptions in OpenAPI specifications can also help identify *inter-parameter dependencies* [25]. Consider line 11 in Figure 1(a), which states that either the `text` or `data` parameter must be specified when invoking endpoint `/check`.

To the best of our knowledge, and as we discuss in more detail in the related work section, no existing technique can automatically extract *all* these types of information from descriptions in OpenAPI specifications in a flexible way and leverage it to improve testing. To bridge this gap, we present NLPtoREST, a fully automated technique that (1) leverages specially tailored natural language processing (NLP) techniques to infer different types of rules from the human-readable part of an OpenAPI specification, (2) encodes these rules using OpenAPI-compliant keywords, and (3) adds the encoded rules to the original specification to produce an enhanced specification that can be transparently utilized by any REST API test generator.

The rule set extracted through NLP techniques can contain spurious or incorrect rules due to the ambiguity of natural-language descriptions, limitations of NLP-based rule inference, or mismatch between API specification and implementation (schema-mismatch faults, for example, are a common category of REST API faults [23]). To mitigate this issue, NLPtoREST incorporates a validation process that checks the NLP-extracted rules against the API implementation by crafting and executing validation test cases and discards rules that fail such validation.

To evaluate our approach, we performed multiple studies using a set of nine REST services, including popular services such as Spotify and the FDIC Bank Data API. First, we assessed the effectiveness of rules extraction and validation against a manually-computed ground truth. NLPtoREST achieved ~50% precision and ~94% recall in the purely NLP-based rules extraction; after validation, the precision increased to ~79%, with 3% reduction in recall.

Second, we compared our approach with RestCT [39]. To the best of our knowledge, RestCT is the only other approach in the literature that uses NLP to extract rules from the human-readable part of a specification, but it only supports inter-parameter dependencies. Our approach was considerably more effective, as it extracted 15 of the 19 inter-parameter dependencies in the benchmark considered, whereas RestCT was unable to extract any.

Finally, we evaluated whether the enhanced specifications created by NLPtoREST can improve the performance of existing REST API testing tools. We studied the performance of eight state-of-the-art REST test generation tools, when provided with the original and enhanced specifications, in terms of (1) code coverage, (2) rate of successful requests, rejected requests, and server error responses, and (3) unique faults found. Using the enhanced specification, the coverage achieved by the testing tools increased, on average, from 11.35% to 23.10% for branch coverage, from 24.96% to 37.52% for statement coverage, and from 22.13% to 33.54% for method coverage. The enhanced specifications contributed to increasing the rate of successful requests (+20%) and server error responses (+2%) and decreasing the rate of rejected requests (-7%). The number of unique server error responses returned by the API increased, on average, by 4%, with the maximum increase of 98.9%. We believe these results are promising, especially considering that there are several directions in which the technique can be extended and improved.

The main contributions of this work are:

- A novel technique for extracting rules from natural-language descriptions in OpenAPI specifications, validating the rules to improve their accuracy, and generating enhanced OpenAPI specifications that existing REST API testing tools can transparently use.
- Empirical results showing that NLPtoREST can extract rules accurately, and the extracted rules can considerably improve the performance of existing REST API testing tools.
- An artifact [29] with the NLPtoREST tool and experimental data that can be used for replicating and extending our work.

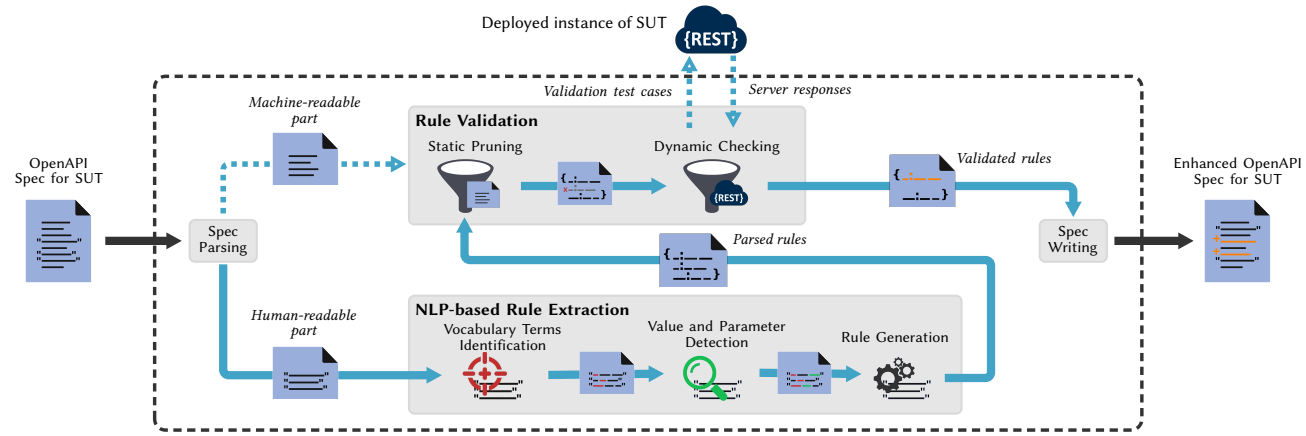

Figure 2: Overview of our NLPtoREST approach.

Table 1: Types of rules identified from natural-language descriptions in API specifications in our preliminary study.

REST service	API endpoint(s)	No. of Params	Rule Categories: Param type/fmt	Param cons	Param ex	Oper cons
Bing Web Search	Search	21	15	10	14	9
Forte	Create application	99	13	67	41	8
Foursquare	Search venues	14	2	0	0	3
GitHub	Get user repos	5	1	0	3	2
Google Geocoding	Geocode request	9	4	0	4	5
Google Maps	Nearby search	11	0	1	2	4
PayPal	Create invoice	114	12	43	5	2
Stripe	Create coupon Create product	28	5	9	3	9
Tumblr	Create post Get blog likes	35	10	2	10	1
Yelp	Search businesses	14	0	0	9	4
Total		**350**	**62**	**131**	**91**	**47**

2 PRELIMINARY STUDY

To assess the kind of information available in the human-readable part of OpenAPI specifications, we manually analyzed the specifications of a set of API endpoints from a benchmark used in previous work [25], which includes 10 real-world REST APIs. This analysis led to the identification of four categories of rules.

Parameter type/format. Rules that define the type and format of a parameter (e.g., `type: string` and `format: date`).

Parameter constraint. Rules that restrict the possible values of a parameter (e.g., a parameter's maximum value).

Parameter example. Rules that specify sample values that a parameter can take.

Operation constraint. Rules that define some specific conditions parameters must satisfy (e.g., inter-parameter dependencies).

Table 1 lists, for each of the OpenAPI specifications and endpoints we analyzed, the number of parameters, the number of rules we identified in the natural-language descriptions, and their categories. In total, we identified over 330 rules, consisting of 62 parameter types/formats, 131 parameter constraints, 91 parameter examples, and 47 operation constraints. These data show that there is considerable scope for identifying meaningful rules from natural-language descriptions in OpenAPI specifications, and guided the development of our approach.

3 OUR APPROACH

We first provide an overview of the NLPtoREST approach (§3.1), introduce the key terminology used in the paper (§3.2), and then describe its two main components in detail: NLP-based Rule Extraction (§3.3) and Rule Validation (§3.4).

3.1 Approach Overview

Figure 2 presents an overview of our technique. The inputs to the technique are the OpenAPI specification of the service under test (SUT) and a deployed instance of the SUT. The output of the technique is an enhanced version of the specification, in which rules extracted from natural-language descriptions in the specification have been added using OpenAPI-supported keywords (where a keyword could be part of the core OpenAPI syntax or an OpenAPI extension [33]). For instance, the rules extracted from the descriptions in lines 21–22 of Figure 1(a) and line 15 of Figure 1(b) would be added to the specification using core OpenAPI syntax "`examples: {1:en-US, 2:de-DE, 3:fr, 4:auto}`" and "`maximum: 50`", respectively. Conversely, the inter-parameter dependency rule inferred from the description in line 11 of Figure 1(a) would be added using the OpenAPI extension keyword "`OnlyOne(text, data)`" [24]. (For this example, as we discuss later, the NLP-based analysis of NLPtoREST extracts the rule as *inclusive or*, i.e., `Or(text, data)`, and the validation step modifies it to *exclusive or*, i.e., `OnlyOne(text, data)`, based on the actual service implementation).

Initially, NLPtoREST parses the OpenAPI specification to extract its human-readable and machine-readable parts, where the former consists of natural-language text contained in `description` fields associated with parameters in the specification. After this preliminary step, NLPtoREST takes these two parts as input and performs *NLP-based Rule Extraction* and *Rule Validation*. Specifically, the NLP-based Rule Extraction module analyzes the human-readable part of a specification to find useful information to extract as rules that could be added to the machine-readable part of the specification, whereas the Rule Validation module validates the extracted rules so that rules that fail validation are discarded.

The NLP-based Rule Extraction phase looks for potential rules in the human-readable part by scanning for a set of search terms

using a custom Word2Vec [11] model, pre-trained on specific OpenAPI terminology (*Vocabulary Terms Identification*). If a description contains one of the search terms, it next looks for values and inter-parameter dependencies using a constituents grammar (*Value and Parameter Detection*). Finally, this phase creates a rule for each (1) keyword and corresponding value and (2) inter-parameter dependency detected. The rules are in the form of key-value pairs (e.g., "`maximum: 50`") or inter-parameter constraints (e.g., "`Or(text, data)`", "`IF videoDef THEN type==video`"). The result of this phase is a set of candidate rules.

The Rule Validation phase checks the candidate rules against the service specification and implementation. Specifically, it first statically analyzes combinations of rules to discard incompatible ones (*Static Pruning*). Then, it generates validation test cases (i.e., HTTP requests) for various rule combinations and executes them against the deployed SUT. When a test case is successfully executed (i.e., a `2XX` status code is obtained), the corresponding rules are further processed by a fine-tuning phase. Rules that pass the complete validation process are marked as *valid*.

The final step of NLPtoREST (*Spec Writing*) adds the validated rules, in machine-readable format, to the original specification, yielding an enhanced OpenAPI specification for the SUT.

3.2 Terminology

OpenAPI specifications describe a RESTful API in terms of its *operations* (i.e., endpoints and HTTP verbs) and their corresponding input and output parameters (names, types, and possibly formats). We use the term *OpenAPI vocabulary* to indicate the set of keywords and string literals used in OpenAPI specifications, where *string literals* correspond to predetermined values for a given *keyword*, such as the value `string` for keyword `type`. We refer to an entry in the OpenAPI vocabulary as an *OpenAPI vocabulary term*. A complete list of such terms can be found in the OpenAPI specification [10]. The term *rule* indicates either a key-value pair, for rules that involve a keyword and possible values for that keyword, or an inter-parameter constraint, for rules that involve inter-parameter dependencies. In turn, an *inter-parameter constraint* can have one of three formats: `IF condition THEN constraint`, `Operator(`$parameter_1$`, ..., `$parameter_n$`)`, or a combination of the two (e.g., `IF condition THEN Operator(`$parameter_1$`, ..., `$parameter_n$`)`, where `Operator` is one of the four inter-parameter dependency types `Or`, `OnlyOne`, `AllOrNone`, and `ZeroOrOne` [24].[1]

3.3 NLP-based Rule Extraction

Extracting rules from OpenAPI specification `description` fields is challenging due to the many ways in which concepts can be expressed in natural language. For instance, the fact that the maximum value of a parameter is 50 can be stated as "the maximum value is 50", "the value is up to 50", "the value can't be larger than 50", and so on. Similarly, an inter-parameter dependency could be described in many ways, such as "you must also set X" or "X must also be specified" [25]. For this reason, simple pattern-based approaches tend to be ineffective, as our evaluation comparing NLPtoREST against one such technique [40] shows (§4.3). To address these challenges, we instead propose a more flexible approach that leverages a custom pre-trained NLP model and constituency-parse-tree-based sentence analysis [19]. In the rest of this section, we describe the three parts of this approach: vocabulary terms identification (§3.3.1), value and parameter detection (§3.3.2), and rule generation (§3.3.3).

[1]Our approach currently supports extraction of simpler forms of these rules where the arguments to `Operator` are parameters of service operations; in the more general formulation of these operators, an argument can be a predicate [24].

Algorithm 1 Search term detection

```
1:  procedure SearchTermDetection(s, vocabularyTerms, sentencesTermsMap)
2:      similarityThreshold ← 0.7                    ▷ Threshold for cosine similarity
3:      for vt in vocabularyTerms do                                          ▷ loop1
4:          searchTerms ← getSearchTerms(vt)
5:          for w in s do                                                     ▷ loop2
6:              for st in searchTerms do                                      ▷ loop3
7:                  if CosineSimilarity(w, st) ≥ similarityThreshold then
8:                      sentencesTermsMap.add(s, vt)
9:                      continue loop1
10:                 end if
11:             end for
12:         end for
13:     end for
14: end procedure
```

3.3.1 Vocabulary Terms Identification. This step analyzes the description field of each parameter in the original OpenAPI specification to identify sentences that may contain rules. To define this part of the technique, we first identified common linguistic patterns used to describe rules and involving OpenAPI vocabulary terms based on our preliminary study in §2 and the common patterns for inter-parameter dependencies described in [25]. From these linguistic patterns, we then identified *search terms* that, if present, may indicate that the sentence contains the corresponding rule or OpenAPI vocabulary term. This resulted in a total of 55 mappings. Although we do not show the mappings here for space reasons, they are available in our artifact [29] (file `settings2.yaml`). We note that, although these relations were derived from our manual analysis of the REST services in Table 1, our evaluation was performed on a different set of services to avoid overfitting and provide evidence of the generalizability of the results.

The use of a fixed set of search terms is limited because different words with similar meanings could be used instead of a specific term. To address this problem, we use word embeddings [11] to detect semantic similarity between words. Although many pre-trained models are publicly available, they are trained using general data sources and may not be ideal in the specific domain of OpenAPI documents. We therefore trained a custom Word2Vec model, `restW2V`, on parameter descriptions taken from numerous OpenAPI documents. Specifically, we trained the model on 1,875,607 text sets taken from 4,064 REST API specifications, using FastText [15], to create a custom model more suited to the terminology and phrases commonly used in OpenAPI specifications.

Given the mappings we identified and `restW2V`, NLPtoREST processes, using Algorithm 1, each sentence in each description of each parameter in the OpenAPI specification of the SUT. The algorithm takes as input a sentence *s*, a set of vocabulary terms, and a map (*sentencesTermsMap*) that associates sentences with the vocabulary terms. In this context, the set of vocabulary terms used is the complete set of OpenAPI vocabulary terms. The outer loop (`loop1`) iterates over all the possible OpenAPI vocabulary terms. For each vocabulary term *vt*, it first extracts the relevant search terms associated with that term (line 3) and then iterates over each word

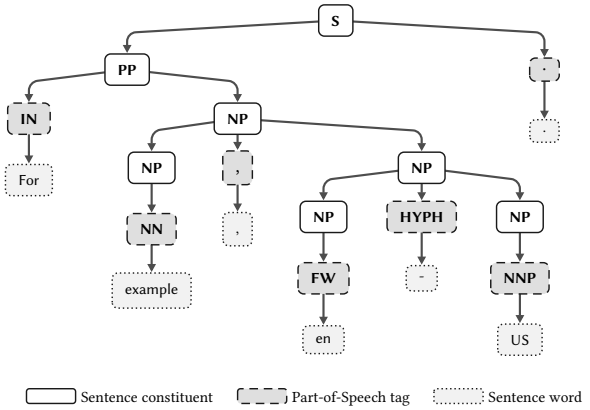

Figure 3: An example of a constituency parse tree.

w in *s* (loop2) and search term *st* (loop3). If the cosine similarity (according to restW2V) of *w* and *st* is 0.7 or greater (line 7), the algorithm adds *s* and *vt* to the map provided as input (line 8) and continues with the next vocabulary term (line 9). When all the sentences have been processed by the algorithm, *sentencesTermsMap* would contain a map from each sentence to any OpenAPI vocabulary term related to that sentence.

Intuitively, this step identifies all the sentences that describe something related to an OpenAPI vocabulary term, such as a keyword, and that may therefore contain a rule. For instance, for the sentence "A language code like 'en-US', 'de-DE', 'fr' or 'auto'" from Figure 1(a), Algorithm 1 would detect the search terms `like`, which is associated with the OpenAPI vocabulary term `example` (see `settings2.yaml` [29]), and would therefore map that sentence to `example`. In the next section (§3.3.2), we describe how NLPtoREST would then identify, in that same sentence, values `en-US`, `de-DE`, `fr` and `auto`. These values, together with keyword `example` would define the possible rule "`examples: {1:en-US, 2:de-DE, 3:fr, 4:auto}`".

3.3.2 Value and Parameter Detection. After identifying the OpenAPI vocabulary terms in a sentence, the NLP-based Rule Extraction employs two strategies to detect the associated values. It primarily uses regular expressions to parse enumerated or quoted strings. If this approach does not yield results, the algorithm leverages the *constituency parse tree* [16] as a fallback method.

The *constituency parse tree* represents the grammatical structure of a sentence in a tree format. This method utilizes part-of-speech (PoS) tagging and sentence constituents for its operation. *PoS tagging* categorizes each word in a sentence into its grammatical function (e.g., noun, verb, adjective, adverb), while *sentence constituents* refer to the structural components of a sentence, such as phrases or clauses. For instance, Figure 3 illustrates the constituency parse tree for the sentence "For example, en-US.", a simplified version of a sentence in Figure 1(a). Each terminal node in the tree signifies a word in the sentence; its parent node is a PoS tag and its other ancestor nodes correspond to its constituent nodes.

Our preliminary studies suggest that enumerated or quoted strings frequently contain critical values in OpenAPI descriptions. Hence, we introduced a regular expression-based detector to directly extract such values. Currently, the detector recognizes enumerated strings starting with "- " or "* ", and quoted strings using

Algorithm 2 Value detection

```
1:  procedure ValueDetection(sentence)
2:      values ← ∅
3:      stopwords ← GetStopwords()                    ▷ Load NLTK based stopwords
4:      regexValues ← RegularExpDetection(sentence)   ▷ First try detection using regular expressions
5:      if regexValues = ∅ then    ▷ If no values detected using regular expressions
6:          tree ← sentence.getConstituencyParseTree()
7:          constituents ← tree.getConstituents(NP)
8:          pos ← tree.getPoS(NOUN, NUM)
9:          for node in constituents do
10:             leafNodes ← node.getLeaves()
11:             childPos ← [child.getPoS() for child in node.getChildren()]
12:             if "," is in childPos then
13:                 commaSets ← node.getCommaSeparatedSets()
14:                 for set in commaSets do
15:                     if set ≠ 'and' and set ≠ 'or' then
16:                         values ← values ∪ set
17:                     end if
18:                 end for
19:             else
20:                 for leaf in leafNodes do
21:                     if leaf.getParent() ∈ pos then
22:                         values ← values ∪ leaf
23:                     end if
24:                 end for
25:             end if
26:         end for
27:     else
28:         values ← regexValues    ▷ If values were detected using regular expressions, use them
29:     end if
30:     values ← values − stopwords                   ▷ Filter out stopwords
31:     return values
32: end procedure
```

double quotes ("), single quotes ('), backticks (`), along with bold strings wrapped in asterisks (*). We intend to further enhance our regular expression detection capabilities to accommodate more markdown features in future iterations.

When processing a sentence, NLPtoREST employs Algorithm 2. This algorithm is designed to detect values within the sentence using two main strategies: regular expressions and constituency parse tree analysis. The algorithm first initializes a set of stopwords, which are derived from the NLTK [13] and supplemented with OpenAPI-specific vocabulary terms (line 3). It then attempts to detect values using regular expressions (line 4). If this initial attempt does not yield any results (i.e., no values are detected using regular expressions), the algorithm then resorts to analyzing the sentence's constituency parse tree (lines 6–7) for all the NP (noun phrase) constituents. A noun phrase is a group of words that functions as a single unit within a sentence and includes a noun. If a noun phrase contains commas, the algorithm assumes that these commas separate different sets of words, each of which could potentially represent a value. It then extracts these comma-separated sets and considers each set as a potential value (lines 13–18). If a comma is not present within the noun phrase, the algorithm instead looks at all the words within the noun phrase whose Part-of-Speech (PoS) tag corresponds to a noun or a number. Each of these words is considered as a potential value (lines 20–24). If values were detected using regular expressions, these values are used (line 28). The algorithm then filters out any stopwords from the detected values (line 30). Finally, the algorithm returns all the values it has detected, excluding any stopwords, as the final result (line 31).

Beyond merely identifying relevant values, NLPtoREST also needs to pinpoint parameter names within sentences. This is crucial

Algorithm 3 Inter-parameter dependency parser

```
1: procedure IPDParser(sentence, operation)
2:     tree ← sentence.getConstituencyParseTree()
3:     sbar_node ← tree.getConstituents(SBAR)        ▷ Subtree node representing a subordinate clause
4:     main_node ← tree.getConstituents(S) − sbar_node        ▷ Subtree node representing a sentence without subordinate clauses
5:     main ← main_node.getLeavesAsSentence()
6:     sbar ← sbar_node.getLeavesAsSentence()
7:     main_voc ← ∅, main_param ← ∅, main_val ← ∅
8:     sbar_voc ← ∅, sbar_param ← ∅, sbar_val ← ∅
9:     IPDTerms ← subset of OpenAPI vocabulary terms relevant for IPDs
10:    sentencesTermsMap ← ∅
11:    SearchTermDetection(main, IPDTerms, sentencesTermsMap)
12:    main_voc ← sentencesTermsMap[main]
13:    sentencesTermsMap ← ∅
14:    SearchTermDetection(sbar, IPDTerms, sentencesTermsMap)
15:    sbar_voc ← sentencesTermsMap[sbar]
16:    for pm in operation.getParameters do
17:        for word in main do
18:            if CosineSimilarity(word, pm) ≥ 0.7 then
19:                main_param ← main_param ∪ pm
20:            end if
21:        end for
22:        for word in sbar do
23:            if CosineSimilarity(word, pm) ≥ 0.7 then
24:                sbar_param ← sbar_param ∪ pm
25:            end if
26:        end for
27:    end for
28:    ExampleTerms ← relevant terms for Parameter example (see §2)
29:    sentencesTermsMap ← ∅
30:    SearchTermDetection(main, ExampleTerms, sentencesTermsMap)
31:    if sentencesTermsMap[main] ≠ ∅ then        ▷ There is a match
32:        main_val ← ValueDetection (main)
33:    end if
34:    sentencesTermsMap ← ∅
35:    SearchTermDetection(sbar, ExampleTerms, sentencesTermsMap)
36:    if sentencesTermsMap[sbar] ≠ ∅ then        ▷ There is a match
37:        sbar_val ← ValueDetection (sbar)
38:    end if
39:    return (main_voc, main_param, main_val, sbar_voc, sbar_param, sbar_val)
40: end procedure
```

for detecting dependencies between different parameters. To handle this task, we employ the Inter-Parameter Dependency Parser (IPDParser), as outlined in Algorithm 3. This algorithm accepts as input a sentence and its corresponding operation (i.e., the operation that contains the parameter described within that sentence). Utilizing Algorithms 1 and 2, IPDParser extracts vital terms, parameter names, and values from both the main and subordinate clauses of the sentence. Here is how the IPDParser works: firstly, it generates a constituency parse tree for the given sentence, using Part-of-Speech (PoS) tags, and extracts the main and subordinate clauses from this tree (lines 2–6). This forms the foundation for our extraction process. Note, for simplicity, the current version of IPDParser in Algorithm 3 does not handle multiple subordinate or main clauses.

Next, IPDParser populates the sets of relevant vocabulary terms, parameter names, and values from both the main and subordinate clauses. It selects a set of OpenAPI vocabulary terms that are pertinent to inter-parameter dependencies [25], and applies the `SearchTermDetection` algorithm to identify these terms within the main and subordinate clauses (lines 9–15). The algorithm then enters a loop (lines 16–27), where it inspects each parameter name from the input operation, searching for corresponding terms in the main and subordinate clauses. This matching process uses a cosine similarity approach (based on `restW2V`) similar to the one employed in Algorithm 1. Subsequently, the algorithm selects a set of terms that are relevant for parameter example rules (line 28, also see §2). It again leverages the `SearchTermDetection` algorithm to identify these terms within the main (lines 29–30) and subordinate (lines 34–35) clauses. When a match is found, it utilizes the `ValueDetection` algorithm to pinpoint corresponding values for the main (line 32) and subordinate (line 37) clauses.

To conclude, the IPDParser outputs a tuple that encapsulates the identified relevant terms, parameters, and values for both the main and subordinate clauses. This comprehensive output aids in a detailed understanding of inter-parameter dependencies within the given sentence.

3.3.3 Rule Generation. The final step of NLP-based Rule Extraction involves generating OpenAPI-compliant rules based on the information computed in the previous steps. Rule generation is performed in a syntax-driven manner for each vocabulary term identified in each description. For example, consider the description in lines 21–22 of Figure 1(a). The search term detection step would match the word "like" with the vocabulary term `examples` (see file `settings2.yaml` [29]), and the value-detection step would discover the values `en-US`, `de-DE`, `fr`, and `auto`. This information would then be combined in the format conforming to the syntax of the `examples` keyword: "`examples: {1:en-US, 2:de-DE, 3:fr, 4:auto}`".

For inter-parameter dependencies, IPDParser would identify inter-parameter dependency terms, parameter names, and values for both the subordinate and main clauses. Rules would then be generated for each clause, separately, again in a syntax-driven way. For example, for the description in line 11 of Figure 1(a), the vocabulary term "or" and parameter names "text" and "data" would be identified in the main clause, from which a rule would be created as `Or(text, data)`. In the presence of a subordinate clause, and inter-parameter dependencies within it, rules would be generated for both the subordinate and the main clauses, and then they would be concatenated using the `Requires` dependency.

3.4 Rule Validation

The NLP-based rule extractor may generate spurious or incorrect rules for different reasons, such as ambiguity in natural-language descriptions, limitations of NLP-based rule inference, or mismatches between API specification and implementation. To mitigate this problem, we introduced the Rule Validation module, a component that performs validity checks and eliminates rules that fail such checks. In some cases, Rule Validation can also make corrections to the extracted rules and fix rules that failed the validity check.

Rule Validation consists of a *static pruning* phase, in which combinations of rules are discarded by *statically* analyzing the compatibility among rules, and a *dynamic checking* phase, in which combinations of rules are validated *dynamically* by submitting requests to a deployed instance of the API and checking the HTTP responses obtained.

Rule Validation is combinatorial in nature, as it is necessary to consider combinations of rules to validate them because each rule depends on the context in which it is applied—an API operation, in this case. For each operation in the API, our validation considers all the rules associated with it, that is, (1) rules associated with the individual parameters belonging to the operation (e.g., a constraint

for a parameter in the operation), and (2) rules involving multiple parameters of the operation (e.g., inter-parameter dependencies). As we discuss in more detail below, Rule Validation must check 2^n combinations in the worst case, with n being the number of rules to validate. Fortunately, static pruning lets us dramatically reduce the number of combinations to consider in the dynamic check.

3.4.1 Static Pruning. Given a set of rules associated with an API operation, the static pruning phase discards syntactically incompatible rule combinations—combinations that contain at least one rule incompatible either with the other rules in the combination or with those in the OpenAPI specification. To do this, we defined a set of rule compatibility policies derived from the OpenAPI standard and its extensions. If all the rules within a combination comply with our policies, we consider the combination syntactically compatible. For the sake of space, here we list only a subset of our policies, which are provided in their entirety in our artifact [29].

- If a parameter is required (`required: true`), then the same parameter cannot appear in the inter-parameter dependency rules `Or`, `OnlyOne`, `ZeroOrOne`, or `AllOrNone`, as these rules would imply that the parameter is not actually required.
- The `maximum` rule is allowed only on numeric parameters.
- A `maximum: a` rule is allowed only for numeric parameters and only if, for the parameter in question, there are no `minimum: b` rules such that $b > a$.
- Rule `exclusiveMaximum: true` is only allowed if the parameter is numeric and a maximum value is defined for the parameter.
- Example, enum, and default values must match the type defined for the parameter.
- Multiple default values cannot exist for the same parameter.

As soon as an incompatibility in a combination is identified, we immediately discard all other combinations that contain the same incompatible rules. This pruning can dramatically reduce the number of combinations to actually check, thus considerably improving the overall efficiency of static pruning.

3.4.2 Dynamic Checking. Dynamic checking tries to verify the validity of rule combinations by interacting with the REST API. In general, each rule can be

- correct and necessary to drive test generation to a successful request (e.g., a correct `enum` value);
- incorrect without causing the request to fail (e.g., a required parameter that is not actually required in the implementation);
- incorrect causing the request to fail (e.g., a wrong `enum` value).

Using different combinations of rules, starting from the largest one, will eventually allow the technique to identify a maximal combination that (1) includes all the rules necessary to generate a valid request (i.e., `2XX` response code), and (2) does not contain any of the rules that cause the request to fail (`4XX` response code). Note that we do not consider server error responses (`5XX`) as successful for rule validation purposes; in our experience, a server error might occur during the processing of a request, and hide a subsequent `4XX` response that would occur had the processing of the request been completed. For each combination of rules, our approach generates a request that considers all of its rules, along with the rules in the original specification. When a request is valid, NLPtoREST classifies the rules in the current combination as *potentially valid*, indicating that it may contain both correct rules and incorrect rules that do not cause the request to fail. Rules are further checked, and possibly fixed, in the subsequent fine-tuning phase.

Fine-tuning. Fine-tuning is meant to further assess the validity of rules. Consider, for instance, an API operation that accepts a non-mandatory parameter `p` and an incorrectly extracted `required: true` rule for that parameter. This rule is semantically incorrect, but would not cause the request to fail with `4XX`. Therefore, the rule would not be discarded during the first phase of dynamic checking, and it will be considered as *potentially valid*.

Fine-tuning implements a strategy that replays the successful requests identified in the previous phase while applying a series of mutations so as to validate the semantics of each rule. Different types of rules are validated with different strategies, and fine-tuning provides a precise validation strategy for 22 out of the 26 rule types supported by the NLP-based Rule Extraction. For example, to validate rules of type `required`, the request is replayed twice, with and without the required parameter. If the request without the parameter is also accepted by the API, the rule is discarded. For another example, to validate `if-then` rules, the request is replayed twice, once applying the predicate and once applying its negation. If the former request is successful and the latter is rejected, the rule is confirmed as valid.

In addition to validation, the fine-tuning of the inter-parameter dependencies rules `Or`, `OnlyOne`, `AllOrNone`, and `ZeroOrOne` provides a *repair strategy*. Consider, for instance, the `Or(text, data)` rule extracted from the specification of LanguageTool in Figure 1(a). During fine-tuning, the successful request obtained in the previous step is mutated and replayed four times: (1) with both `text` and `data` parameters; (2) with the `text` parameter only; (3) with the `data` parameter only; and (4) with neither `text` nor `data`. Based on the responses obtained (and their status codes), the fine-tuning will return a new rule, that better complies with the API implementation. In this particular case, the returned rule was `OnlyOne(text, data)` because requests (2) and (3) were accepted by the API, while requests (1) and (4) were rejected. The rejection of request (1) indicates that the two parameters cannot be used simultaneously, contrary to what an `Or` rule states. The rule is therefore repaired.

In conclusion, we consider *validated* the rules belonging to a combination not discarded by static pruning, for which dynamic checking could generate a successful request, and that could be successfully processed by fine-tuning. These rules are included in the enhanced OpenAPI specification.

4 EVALUATION

In this section, we present an empirical evaluation of NLPtoREST. Specifically, we assess the effectiveness of NLP-based Rule Extraction and Rule Validation, compare our approach with RestCT [39] (a related tool that performs pattern-matching-based rule extraction from OpenAPI specifications), and investigate how enhanced specifications generated by NLPtoREST can help in improving the testing performance of state-of-the-art REST API testing tools.

4.1 Research Questions

We formulated the following research questions for the evaluation:

RQ1: How effective is NLP-based Rule Extraction in inferring rules from natural-language descriptions in OpenAPI specifications?
RQ2: How effective is Rule Validation in pruning incorrect rules?
RQ3: How does NLPtoREST compare with related NLP-based rule-extraction tools?
RQ4: Can the enhanced specification generated by NLPtoREST improve the performance of REST API testing tools?

To answer RQ1, we compared the rules inferred by NLP-based Rule Extraction against a manually created ground truth and computed true positives, false positives, false negatives, precision, recall, and F_1 score. The ground truth was created by four graduate students with expertise in REST API specifications. These students were not privy to the rules identified in our preliminary experiment. Instead, we provided them with documentation of the OpenAPI syntax and the OpenAPI specifications, and instructed them to infer rules from descriptions. We did not impose any other restrictions on their interpretation. The students worked independently and compared their results, resolving any discrepancies through discussion until reaching a consensus.

To answer RQ2, we compared the validated rules against the ground truth, computing the same metrics that were used for RQ1.

To answer RQ3, we compared NLPtoREST with RestCT [39], a test generation tool that uses pattern matching to extract inter-parameter dependencies from text descriptions. We compare the tools in terms of the number of correct rules extracted.

To answer RQ4, we compared the performance of state-of-the-art REST API testing tools in two settings: (1) the tools take as input the original OpenAPI specification, and (2) the tools take as input the enhanced OpenAPI specification generated by NLPtoREST. To measure tool performance, we used line, branch, and method coverage achieved, and the frequency of successful (2XX), client error (4XX), and server error (5XX) status codes. We expect the enhanced specifications generated by NLPtoREST to help the testing tools generate more successful requests (2XX status code) and requests that trigger server errors (5XX status code), while generating fewer invalid requests (4XX status code). Consequently, we expect the testing tools to achieve higher code-coverage and fault-detection rates (the latter indicated by the number of server errors triggered).

4.2 Experiment Setup

4.2.1 REST API Benchmark. To build the evaluation benchmark, we started with the services used in two recent empirical studies in the area, evaluating online testing of REST APIs [27] and comparing automated REST API testing tools [17]. To the best of our knowledge, the first study includes the largest number of commercial REST APIs, whereas the second one contains the largest number of open-source REST APIs, with 33 REST services in total.

We then put this set of services through a filtering process. Our goal was to focus on industrial-sized services, as current testing tools are already very effective on small projects but struggle with larger ones [17]. Following [27], we consider a service to be industrial-sized if its source code consists of more than 10,000 lines of code (LoC). Therefore, we excluded from the benchmark services that have less than 10k LoC. Next, we excluded the services that we had used to build the NLP model for NLPtoREST, to avoid overfitting. This resulted in 12 services. We then tested each service using each REST API testing tool considered (§4.2.2) for one hour to check for any execution issues. Unfortunately, we found that one service, Amadeus v2, had an authentication issue. We contacted the API developers, who informed us that they were in the process of fixing the issue; however, the fix was not available at the time of our experimentation. Additionally, three services—DHL, Marvel, and YouTube—implement rate limiting (allowed requests per hour), which caused more than half of the requests in our testing sessions to be blocked. We, therefore, had to exclude these APIs, replacing YouTube with YouTube Mock (taken from the benchmark of [27]).

Our final benchmark contains nine REST services: Federal Deposit Insurance Corporation (FDIC), LanguageTool, OhSome, OMDB, REST, Genome Nexus, OCVN, Spotify, and YouTube Mock.

4.2.2 REST API Testing Tools. To select the tools, we evaluated state-of-the-art black-box testing tools for REST APIs that generate test cases based on OpenAPI specifications. From the tools studied in [17], we chose the top seven performers. We also considered two additional tools, Morest [22] and RestCT [39], developed recently. Of these, we had to exclude RestCT [39] due to compatibility issues with most of the APIs in our benchmark.[2] Our final set of eight tools includes EvoMasterBB [2], bBOXRT [21], Morest [22], RESTest [26], RESTler [3], RestTestGen [8], Schemathesis [14], and Tcases [18].

4.2.3 Experiment Procedure. We ran the experiments using the cloud computing service provided by Google Cloud. In particular, we used ten e2-standard-4 machines running Ubuntu 20.04. Each machine has 24 2.2GHz Intel-Xeon processors and 128 GB of RAM.

We ran the testing tools with a time budget of one hour. This choice was based on a previous study [17], which showed that within one hour, the code coverage achieved by fuzzers tends to plateau. By fixing the testing time budget, we aimed to ensure a fair comparison of the metrics collected for each REST API in our benchmark. To account for randomness, we repeated the experiments 10 times and collected average metrics across the 10 runs, resulting in 10 hours of execution per testing tool.

To collect code coverage information, we used JaCoCo [34] to instrument the service code. We could measure coverage for only the four open-source services in our benchmark (Genome Nexus, LanguageTool, OCVN, and YouTube Mock); for the remaining services, source code is unavailable for instrumentation.

4.3 Experiment Results

4.3.1 RQ1: Effectiveness of NLP-based Rule Extraction. Table 2 presents the results for RQ1. It shows, for each service, the number of rules in the ground truth (Column 2). Columns 3–8 present data on the accuracy of NLP-based Rule Extraction: the number of rules extracted correctly (true positives or TP), the number of erroneous rules extracted (false positives or FP), the number of rules missed (false negatives or FN), and the aggregate metrics (precision, recall, and F_1 score).

[2]We notified the developers of RestCT about these compatibility issues, which they were working to resolve at the time of our experimentation.

Table 2: Effectiveness of NLP-based Rule Extraction and Rule Validation.

REST Service	No. of Rules in Ground Truth	*NLP-based Rule Extraction*						*Rule Validation*					
		TP	FP	FN	Precision	Recall	F_1	TP	FP	FN	Precision	Recall	F_1
FDIC	45	42	36	3	54%	93%	68%	42	25	3	63% (+16%)	93% (-)	75% (+10%)
Genome Nexus	81	79	3	2	96%	98%	97%	79	3	2	96% (-)	98% (-)	97% (-)
LanguageTool	20	20	12	0	63%	100%	77%	18	2	2	90% (+44%)	90% (-10%)	90% (+17%)
OCVN	17	15	2	2	88%	88%	88%	13	1	4	93% (+5%)	76% (-13%)	84% (-5%)
OhSome	14	13	66	1	16%	93%	28%	12	11	2	52% (+217%)	80% (-14%)	63% (+126%)
OMDB	2	2	0	0	100%	100%	100%	2	0	0	100% (-)	100% (-)	100% (-)
REST Countries	32	28	1	4	97%	88%	92%	28	0	4	100% (+4%)	88% (-)	93% (+2%)
Spotify	88	83	68	5	55%	94%	69%	82	28	6	75% (+36%)	93% (-1%)	83% (+19%)
YouTube	34	30	126	4	19%	88%	32%	28	9	6	76% (+294%)	82% (-7%)	79% (+150%)
Total	**338**	**312**	**314**	**21**	**50%**	**94%**	**65%**	**304**	**79**	**30**	**79% (+58%)**	**91% (-3%)**	**85% (+31%)**

The results demonstrate that NLP-based Rule Extraction is highly effective in extracting rules from the human-readable part of OpenAPI specifications, with overall recall of 94% and only 21 missed rules from the ground truth of 338 rules. This high recall rate is consistent across the services, ranging from the minimum of 88% (for YouTube) to the maximum of 100% (for LanguageTool and OMDB). In terms of precision, NLP-based Rule Extraction was less effective, with 50% precision over all the services and approximately half of the 626 extracted rules being false positives. However, for four of the services (Genome Nexus, OCVN, OMDB, and REST Countries), the technique achieved high precision (96%, 88%, 100%, and 97%, respectively). In general, the low precision highlights the need for a validation phase to filter out incorrect rules.

We discuss two examples to illustrate false positives and false negatives in the rules computed by NLP-based Rule Extraction. The first example is a false positive for the OhSome service, whose `GET /elements/area` operation has a parameter called `filter`. By analyzing the parameter description "Combines several attributive filters, e.g. OSM type, the geometry (simple feature) type, as well as the OSM tag; no default value", NLP-based Rule Extraction identified "OSM" as an example value for the parameter, creating the rule "`example: OSM`". This is incorrect because "OSM" is an acronym for OpenStreetMap and the parameter description explains the OSM attributes that could be provided as filters, without enumerating an actual example. Thus, there is no example to be extracted from this parameter description. To address such cases, NLP-based Rule Extraction would have to leverage additional sources of domain-specific information. For instance, in this case, the OpenAPI specification of graphhopper.com contains relevant example values for OSM type (e.g., "node", "way", and "relation").

The second example is a false negative for the FDIC service. The `GET /institutions` operation has a parameter `sort_order` with description "Indicator if ascending (ASC) or descending (DESC)". NLP-based Rule Extraction failed to identify "ASC" or "DESC" as example values. The reason is that the search terms used for vocabulary terms identification were inadequate for detecting this sentence as potentially containing a parameter-example rule. This limitation could be addressed in different ways, such as expanding the set of search terms used for vocabulary terms identification and fine-tuning the `restW2V` model with a larger training dataset.

> NLP-based Rule Extraction effectively infers rules from OpenAPI specifications with high recall (88%-100%, overall 94%). However, its precision is lower and varies more (16%-100%, overall 50%), indicating the necessity of a validation step to eliminate false positives.

4.3.2 RQ2: Effectiveness of Rule Validation. Columns 8–14 of Table 2 present data on the effectiveness of Rule Validation in terms of the six metrics computed on the validated rules. For precision, recall, and F_1 score, the table also shows the percentage difference from the respective scores before validation.

The data illustrate that Rule Validation is quite effective in improving the accuracy of NLPtoREST. Over all services, it filtered out 235 of the 314 false positives, causing the precision to increase from 50% to 79%—for an improvement of 58%. Moreover, this was accompanied by only a small reduction in recall from 94% to 91%, with nine additional false negatives over the NLP-extracted rules. Although there is usually such a trade-off between precision and recall, we note that the decrease in recall occurs mostly due to inconsistencies between the OpenAPI specification and the API implementation.

In general, validation results in an enhanced specification that is more aligned with the service implementation and is, therefore, more helpful to test generators in crafting successful requests. In some cases, validation can also help detect inconsistent or ambiguous specifications. For example, as discussed in §3.4, for the phrase "This or 'text' is required." in Figure 1(a), NLP-based Rule Extraction correctly identifies `Or(text, data)` as an inter-parameter dependency rule. However, during the validation phase, feedback from the service implementation leads to the rule being updated to `OnlyOne(text, data)`. In such cases, it is not possible to determine which of the two rules is the intended behavior of the system (i.e., whether the description in the specification is imprecise or the implementation has a bug). NLPtoREST can facilitate the detection of such inconsistencies and ambiguities.

Although rule validation is highly beneficial in improving the precision of rule extraction, it comes with some limitations. When Rule Validation fails to produce successful requests, it is unable to validate rules. For example, considering the incorrect rule `example: OSM` inferred by NLP-based Rule Extraction for the OhSome service, Rule Validation was unable to remove it because it failed to generate a

Table 3: Improvement in performance of testing tools when fed with enhanced specifications, compared to the baseline performance with original specifications.

	Freq. of 2XX		Freq. of 4XX		Freq. of 5XX	
Tool Name	*Original*	*Enhanced*	*Original*	*Enhanced*	*Original*	*Enhanced*
bBOXRT	11.5%	20.6% (+79%)	87.4%	77.9% (-11%)	1.1%	1.6% (+36%)
EvoMaster	14.3%	21.5% (+50%)	67.2%	54.5% (-23%)	18.5%	24.0% (+30%)
Morest	5.5%	8.3% (+51%)	73.4%	72.8% (-1%)	21.1%	18.9% (-10%)
RESTest	55.2%	61.6% (+12%)	34.2%	29.6% (-13%)	10.6%	8.9% (-17%)
RESTler	14.6%	11.3% (-23%)	50.4%	47.3% (-6%)	35.0%	41.4% (+18%)
RestTestGen	29.2%	32.1% (+10%)	68.4%	67.8% (-1%)	2.5%	0.1% (-96%)
Schemathesis	31.2%	36.9% (+18%)	52.5%	49.8% (-5%)	16.3%	13.3 (-18%)
Tcases	12.5%	17.7% (+42%)	46.0%	47.9% (-4%)	41.5%	40.8% (-2%)
Average	**21.8%**	**26.2% (+20%)**	**59.9%**	**56.0% (-7%)**	**18.3%**	**18.6% (+2%)**

valid request for the `GET /elements/area` operation that contained the parameter.

> Rule Validation filters out over 235 false positives, which is approximately 75% of the initial false positives, increasing precision by 58% (from 50% to 79%). This is accompanied by a minor 3% decrease in recall due to inconsistencies between API specifications and implementations.

4.3.3 RQ3: Comparison with State-of-the-art NLP-based Rule Extractors. We compared NLPtoREST with RestCT, a REST API test generation tool that uses pattern matching to extract inter-parameter dependencies from text descriptions. Due to compatibility issues with most of the evaluated REST APIs, RestCT was excluded from the RQ4-related experiment. However, these compatibility issues were not related to RestCT's NLP module for inter-parameter dependencies detection. Hence, we compared NLPtoREST with RestCT's NLP module by running them on the nine services in our benchmark, and counting the number of inter-parameter dependency rules each tool detected. RestCT performed poorly, as it only matches sentences with the same composition as the pattern, and it was unable to identify any inter-parameter dependencies in the benchmark. In contrast, NLPtoREST uses constituency parse tree analysis to perform flexible matching, enabling it to identify 15 out of the 19 inter-parameter dependencies that exist in the benchmark.

> NLPtoREST is more effective than RestCT, a state-of-the-art NLP-based approach for OpenAPI rule extraction; it was able to identify 15 out of 19 inter-parameter dependencies, while RestCT was unable to identify any.

4.3.4 RQ4: Improving Testing Tools Performance. Table 3 and Figure 4 present data on how the REST API testing tools perform when served with the original and the enhanced OpenAPI specifications: Table 3 shows the frequency of status codes obtained, whereas Figure 4 shows the line, branch, and method coverage achieved. (As mentioned earlier, we performed coverage measurement for the four open-source services in our benchmark.)

Table 3 shows, for each tool and the original and enhanced specifications, the ratio of the number of successful requests (2XX) to the total number of requests sent; the ratio of the number of rejected requests (4XX) to the total number of requests sent; and the ratio of requests triggering server errors (5XX) to the total number of requests sent. The results show that, on average, enhanced specifications helped increase the rate of successful requests considerably (+20%) and server error responses slightly (+2%), and decrease the rate of rejected requests moderately (-7%). For 5XX responses, we also computed the number of unique responses returned by the API. On average, the number of unique server errors increased by 4%, from 56.8 to 59.

Figure 4 depicts the comparison of branch, line, and method coverage achieved by each testing tool, using the original specifications and the enhanced specifications. The figure is divided into subplots, one for each tool, where each subplot compares the original and enhanced specifications for the three coverage types. The data is presented in the form of box plots that show the four quartiles along with the minimum, maximum, and outlier values. The mean value is shown as a black square marker, whereas the median value is depicted with an orange line. The results illustrate that the NLPtoREST-generated specifications lead to a significant improvement in coverage across all the tools, with average improvements of 103% (from 11.35% to 23.10%) for branch coverage, 50% (from 24.96% to 37.52%) for line coverage, and 52% (from 22.13% to 33.54%) for method coverage.

Overall, the results of this experiment demonstrate the effectiveness of NLPtoREST in improving the performance of REST API testing tools. The significant improvement in generating successful requests, as well as the significant increase in code coverage, suggests that the rules generated by NLPtoREST have a major impact on the performance of test generation tools.

> NLPtoREST significantly improves the performance of REST API testing tools, helping them generate more successful requests (+20%) and fewer bad requests (-7%), trigger more unique server responses (+4%), and achieve on average 103%, 52%, and 50% more branch, method, and line coverage, respectively.

4.4 Threats to Validity

Our study may suffer from both external and internal threats to validity.

In terms of external threats, the services and testing tools included in our evaluation might not be representative. To mitigate this threat, we selected nine services based on two existing comprehensive studies on REST API testing, one focusing on commercial REST APIs [27] and the other on open-source REST APIs [17]. We selected eight testing tools based on the results presented in [17] and more recent publications. The settings used for the testing tools and RESTful services might also not be optimal. However, we attempted to minimize the impact of hardware and software configurations on the results by using the same settings as a previous study [17] and default settings for all RESTful services during the experiments.

In terms of internal threats, the training set for the `restW2V` model might not be representative, which could impact the accuracy of the model. To mitigate this threat, we aimed to make the training dataset as large and diverse as possible, by using 1,875,607 text sets taken from 4,064 REST API specifications.

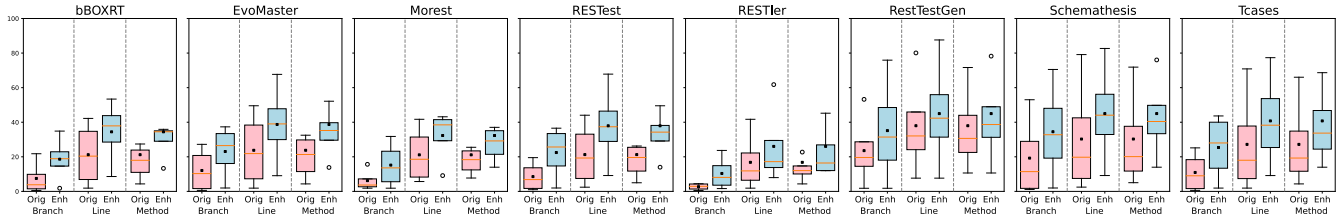

Figure 4: Comparison of code coverage results for original (Org) and enhanced (Enh) specifications, with respect to branch, line, and method coverage.

Another internal threat is the manual ground truth generation. The individual interpretation of OpenAPI specifications by the students involved could have influenced the ground truth. To address this, the students were asked to reach a consensus on any discrepancies, and they were not aware of the specific rules elicited in our preliminary experiment, thus reducing the risk of bias.

Lastly, the experiment results reported in this paper might also be influenced by the specific implementation of the NLPtoREST approach. To address this threat, we thoroughly examined our code to minimize the risk of mistakes and made it publicly accessible to allow for review and extension by others.

5 RELATED WORK

Our work, NLPtoREST, intersects with research on enhancing OpenAPI specifications and test generation, utilizing natural language processing (NLP).

RestCT [40] is a close work that generates automated REST tests by inferring inter-parameter dependencies through pattern matching. It uses a combinatorial testing approach to handle complex APIs and employs a popular covering array generation tool, ACTS, to generate constrained covering arrays. However, it relies on a set of hard-coded patterns, while NLPtoREST adopts a more flexible approach, leveraging the same catalog of constraints [25] but with a broader NLP-based extraction method.

ARTE [1], another tool in this field, generates test inputs for web APIs by querying the DBpedia knowledge base using parameter names. The parameter names are identified via processing of the parameter descriptions. As opposed to NLPtoREST, ARTE does not take into account the format of the values, such as the default, minimum, and maximum values, that occur in the textual descriptions. Consequently, the input values it generates are much more limited for use as OpenAPI example rules than NLPtoREST's generated example values. Hence, ARTE has both different objectives and different processes from NLPtoREST for extracting information from OpenAPI textual descriptions.

Documentation that exists in web pages is another source of information for inference of OpenAPI specifications [7, 41]. These methods involve web crawling to identify the relevant documentation and extraction of relevant information such as base URL, path templates, and HTTP methods. While these methods also aim to generate OpenAPI specifications from documentation, they differ from NLPtoREST in the approach used to analyze the text. Rather than using NLP techniques to analyze human-readable text descriptions, these methods extract the necessary information from web documentation via methods such as clustering. As a result, these methods might not be able to capture the same level of detail and nuance that NLPtoREST can provide, such as parameter constraints and example values.

Other research [4–6, 28, 32, 37, 38] uses NLP for test generation from program specifications. While these methods are effective for their specific contexts, they do not focus on consuming, generating, or enhancing OpenAPI specifications. Moreover, they do not necessarily tackle the complex task of inferring inter-parameter dependencies from natural language descriptions. In contrast, NLPtoREST goes beyond simple constraint inference and aims to interpret more nuanced and complex semantic information, such as inter-parameter dependencies, from the unconstrained natural language parts of the OpenAPI specification to enhance it. This approach makes NLPtoREST uniquely positioned to offer more comprehensive and refined enhancements to OpenAPI specifications.

6 CONCLUSION AND FUTURE WORK

We presented NLPtoREST, a novel technique and tool for extracting information from human-readable parts of OpenAPI specifications to improve the performance of REST API testing tools. Our empirical evaluation showed high recall for NLP-based rule extraction, high precision for rule validation, and considerable improvement in the successful request generation rate and code coverage for eight REST API test generators when using the enhanced specification generated by NLPtoREST. Additionally, we showed that NLPtoREST significantly outperforms a state-of-the-art REST API testing tool that infers inter-parameter dependencies using pattern matching. In future work, we will explore the potential application of generative AI models [12, 30, 35] to further refine and expand the capabilities of NLPtoREST. Also, we will look for other possible extensions such as chat bot generation from API specifications [36]. Additionally, we will extend our evaluation to include more services and test generation tools, investigate the use of alternative NLP models and training datasets, and explore ways of extending our technique, e.g., via analysis of server messages and custom parse trees, to improve the accuracy of rule extraction and increase the types of rules that can be extracted.

ACKNOWLEDGMENT

This research has been partly supported by the European Union's Horizon Europe research and innovation programme under grant agreement No 101070238.

REFERENCES

[1] J. C. Alonso, A. Martin-Lopez, S. Segura, J. Garcia, and A. Ruiz-Cortes. 2023. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* 49, 01 (jan 2023), 348–363. https://doi.org/10.1109/TSE.2022.3150618

[2] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1, Article 3 (jan 2019), 37 pages. https://doi.org/10.1145/3293455

[3] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 748–758. https://doi.org/10.1109/ICSE.2019.00083

[4] Mourad Badri, Linda Badri, and Marius Naha. 2004. A Use Case Driven Testing Process: Towards a Formal Approach Based on UML Collaboration Diagrams. In *Formal Approaches to Software Testing*, Alexandre Petrenko and Andreas Ulrich (Eds.). Springer Berlin Heidelberg, Berlin, Germany, 223–235. https://doi.org/10.1007/978-3-540-24617-6_16

[5] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezze, and Sergio Delgado Castellanos. 2018. Translating Code Comments to Procedure Specifications. In *Proceedings of the 2018 International Symposium on Software Testing and Analysis (ISSTA 2018)* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 242–253. https://doi.org/10.1145/3213846.3213872

[6] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezze. 2022. Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 19, 11 pages. https://doi.org/10.1145/3551349.3556961

[7] Hanyang Cao, Jean-Rémy Falleri, and Xavier Blanc. 2017. Automated Generation of REST API Specification from Plain HTML Documentation. In *Service-Oriented Computing*, Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol (Eds.). Springer International Publishing, New York, NY, USA, 453–461. https://doi.org/10.1007/978-3-319-69035-3_32

[8] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2022. Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability* 32 (01 2022). https://doi.org/10.1002/stvr.1808

[9] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-Based Software Architectures*. Ph. D. Dissertation. University of California, Irvine.

[10] The Linux Foundation. 2022. OpenAPI specification. https://spec.openapis.org/oas/v3.1.0.

[11] Yoav Goldberg and Omer Levy. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. arXiv:1402.3722 [cs.CL]

[12] Google. 2023. Google Bard. https://bard.google.com/

[13] Nitin Hardeniya, Jacob Perkins, Deepti Chopra, Nisheeth Joshi, and Iti Mathur. 2016. *Natural language processing: python and NLTK*. Packt Publishing Ltd, Birmingham, UK.

[14] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving Semantics-Aware Fuzzers from Web API Schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 345–346. https://doi.org/10.1145/3510454.3528637

[15] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. 2016. FastText.zip: Compressing text classification models. arXiv:1612.03651 [cs.CL]

[16] Daniel Jurafsky and James H. Martin. 2021. Speech and Language Processing: Constituency Parsing. https://web.stanford.edu/~jurafsky/slp3/13.pdf.

[17] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. '22. Automated test generation for REST APIs: no time to rest yet. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, New York, NY, USA, 289–301. https://doi.org/10.1145/3533767.3534401

[18] Kerry Kimbrough. 2023. Tcases. https://github.com/Cornutum/tcases

[19] Dan Klein and Christopher D. Manning. 2003. Accurate Unlexicalized Parsing. In *Proceedings of the 41st annual meeting of the association for computational linguistics*. Association for Computational Linguistics, Edinburgh, Scotland, 423–430.

[20] LanguageTool. 2023. LanguageTool REST API. https://languagetool.org/proofreading-api.

[21] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A Black Box Tool for Robustness Testing of REST Services. *IEEE Access* 9 (2021), 24738–24754. https://doi.org/10.1109/ACCESS.2021.3056505

[22] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: Model-Based RESTful API Testing with Execution Feedback. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1406–1417. https://doi.org/10.1145/3510003.3510133

[23] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the Faults Found in REST APIs by Automated Test Generation. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 41 (mar 2022), 43 pages. https://doi.org/10.1145/3491038

[24] Alberto Martin-Lopez, Sergio Segura, Carlos Müller, and Antonio Ruiz-Cortés. 2022. Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. *IEEE Transactions on Services Computing* 15, 4 (2022), 2342–2355. https://doi.org/10.1109/TSC.2021.3050610

[25] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A Catalogue of Inter-Parameter Dependencies in RESTful Web APIs. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings* (Toulouse, France). Springer-Verlag, Berlin, Heidelberg, 399–414. https://doi.org/10.1007/978-3-030-33702-5_31

[26] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 682–685. https://doi.org/10.1145/3460319.3469082

[27] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2022. Online Testing of RESTful APIs: Promises and Challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE '22)*. Association for Computing Machinery, New York, NY, USA, 408–420. https://doi.org/10.1145/3540250.3549144

[28] Manish Motwani and Yuriy Brun. 2019. Automatically Generating Precise Oracles from Structured Natural Language Specifications. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 188–199. https://doi.org/10.1109/ICSE.2019.00035

[29] Myeongsoo Kim and Davide Corradini. 2023. Experiment infrastructure and data for NLPtoREST. https://github.com/codingsoo/nlp2rest.

[30] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[31] OpenAPI. 2023. OpenAPI standard. https://www.openapis.org.

[32] Johannes Ryser and Martin Glinz. 1999. A scenario-based approach to validating and testing software systems using statecharts. https://doi.org/10.5167/uzh-205008

[33] SmartBear Software. 2023. OpenAPI Extensions. https://swagger.io/docs/specification/openapi-extensions/.

[34] EclEmma Team. 2023. JaCoCo. https://www.eclemma.org/jacoco/.

[35] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[36] Mandana Vaziri, Louis Mandel, Avraham Shinnar, Jérôme Siméon, and Martin Hirzel. 2017. Generating Chat Bots from Web API Specifications. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)* (Vancouver, BC, Canada) *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 44–57. https://doi.org/10.1145/3133850.3133864

[37] Chunhui Wang, Fabrizio Pastore, and Lionel Briand. 2018. Automated Generation of Constraints from Use Case Specifications to Support System Testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Piscataway, NJ, USA, 23–33. https://doi.org/10.1109/ICST.2018.00013

[38] Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2022. Automatic Generation of Acceptance Test Cases From Use Case Specifications: An NLP-Based Approach. *IEEE Transactions on Software Engineering* 48, 2 (2022), 585–616. https://doi.org/10.1109/TSE.2020.2998503

[39] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/3510003.3510151

[40] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/3510003.3510151

[41] Jinqiu Yang, Erik Wittern, Annie T. T. Ying, Julian Dolby, and Lin Tan. 2018. Towards Extracting Web API Specifications from Documentation. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 454–464. https://doi.org/10.1145/3196398.3196411

Received 2023-02-16; accepted 2023-05-03