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  
sinha@sus.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, which can in turn significantly improve testing tools’ performance.

CCS CONCEPTS

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

1 INTRODUCTION

The REST (REpresentational State Transfer) architectural style [10] 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 [32], 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.
The main feature - check a text with LanguageTool for possible style and grammar issues.

- name: text
type: string
in: query
description: The main feature - check a text with LanguageTool for possible style and grammar issues.

- name: language
type: string
required: false
in: formData
description: A comma-delimited list of one or more languages 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 [26]. 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, 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 automatically extracted technique that (1) leverages specially tailored natural language processing (NLP) techniques to infer different types of rules from the human-readable part of OpenAPI specifications,

(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 set of rules 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 common [24]). To mitigate this issue, NLPtoREST includes a validation step 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 a 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 an OpenAPI 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 a 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. It is worth noting that the enhanced OpenAPI specifications may also be useful in other contexts, such as to provide developers with additional documentation.
- 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 [30] with the NLPtoREST tool and experiment data that can be used for replicating and extending our work.
Table 1: Types of rules identified from natural-language descriptions in our preliminary study.

<table>
<thead>
<tr>
<th>REST service</th>
<th>API endpoint(s)</th>
<th>No. of Params</th>
<th>Param type/fmt</th>
<th>Param cons</th>
<th>Param cons</th>
<th>Oper cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bing Web Search</td>
<td>Search</td>
<td>21</td>
<td>19</td>
<td>10</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>Forte</td>
<td>Create application</td>
<td>99</td>
<td>13</td>
<td>67</td>
<td>41</td>
<td>8</td>
</tr>
<tr>
<td>Foursquare</td>
<td>Search venues</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>GitHub</td>
<td>Get user repos</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Google Geocoding</td>
<td>Geo code request</td>
<td>9</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Google Maps</td>
<td>Nearby search</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>PayPal</td>
<td>Create invoice</td>
<td>114</td>
<td>12</td>
<td>43</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Stripe</td>
<td>Create coupon</td>
<td>28</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>Stripe</td>
<td>Product</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tumblr</td>
<td>Create post</td>
<td>35</td>
<td>10</td>
<td>2</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Yelp</td>
<td>Search businesses</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>350</td>
<td>62</td>
<td>132</td>
<td>91</td>
<td>47</td>
</tr>
</tbody>
</table>

3 OUR APPROACH

In this section, we first provide an overview of the whole NLPtoREST approach (§3.1), 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 have been added using OpenAPI-supported keywords (i.e., part of the core OpenAPI syntax or OpenAPI extensions [34]). 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 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)" [25]. (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 and discards rules that fail such validation.

The NLP-based Rule Extraction phase looks for potential rules by scanning for a set of search terms using a custom Word2Vec [12] model, pre-trained on 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 Name 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 Rule Validation phase checks the candidate rules generated by the previous phase against the service specification and implementation. It (1) statically analyzes combinations of rules to discard incompatible ones (Static Pruning) and (2) 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 standard [11]. 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(parameter₁, ..., parameterₙ), or a combination of the two (e.g., IF condition THEN Operator(parameter₁, ..., parameterₙ)), where Operator is one of four inter-parameter dependency types Or, OnlyOne, AllOrNone, and ZeroOrOne [25]. Our approach currently supports extraction of simpler forms of these rules where the Operator arguments are simple parameters; in the more general formulation of these operators, an argument can be a predicate [25].

3.3 NLP-based Rule Extraction
Extracting rules from 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, inter-parameter dependencies can be described in many ways, such as “you must also set X” or “X must also be specified” [26]. Therefore, simple pattern-based approaches tend to be ineffective, as our evaluation comparing NLPtoREST against one such technique [40] shows (§4.3). We instead propose a more flexible approach that leverages a custom pre-trained NLP model and constituency-parse-tree-based sentence analysis [20] and consists of three main parts:

### 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  
4. searchTerms ← getSearchTerms(\(vt\)) ▶ loop1
5. for \(w \in s\) do ▶ loop2
6. for \(st\) in searchTerms do ▶ loop3
7. if CosineSimilarity(\(w\), \(st\)) ≥ similarityThreshold then  
8. sentencesTermsMap.add(\(s\), \(vt\)) ▶ loop3
9. continue loop1
10. end if
11. end for
12. end for
13. end for
14. end procedure

vocabulary terms identification (§3.3.1), value and parameter name detection (§3.3.2), and rule generation (§3.3.3).

3.3.1 Vocabulary Terms Identification. This step analyzes the description field of each parameter 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 [26]. 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 [30]. 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 [12] 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 [16], so as 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 \(w \in s\) (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
Our preliminary work suggests that enumerated or quoted strings constitute such terms. In the next section, we introduce a regular expression-based detector to directly extract values, together with keyword examples: \{1:en-US, 2:de-DE, 3:fr, 4:auto\}.

### 3.3.2 Value and Parameter Name Detection

The NLP-based Rule Extraction module employs two strategies to detect the values associated with the previously identified OpenAPI vocabulary terms in a sentence. First, it uses regular expressions to parse enumerated or quoted strings. If this approach does not yield results, it leverages the constituency parse tree as a fallback method.

The constituency parse tree represents the grammatical structure of a sentence in 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 “A language code like ‘en-US’, ‘de-DE’, ‘fr’ or ‘auto’” from Figure 1(a). Each terminal node in the tree corresponds to a word in the sentence, its parent node is a PoS tag, and its other ancestor nodes correspond to its constituent nodes.

Our preliminary work suggests 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 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 work.

NLP to REST employs Algorithm 2 to detect values in a sentence via regular-expression and constituency-parse-tree analyses. The algorithm first initializes a set of stopwords, which are derived from NLP to REST [17] and supplemented with OpenAPI-specific vocabulary terms (line 3). It then attempts to detect values using regular expressions (line 4) and saves them if successful (line 28). Conversely, if no values are detected using regular expressions, the algorithm analyzes the constituency parse tree for the sentence to identify all the NP (noun phrase) constituents (lines 6–7). A noun phrase is a group of words that function as a single unit within a sentence and include 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 identifies all the words whose PoS tag corresponds to a noun or a number as potential values (lines 20–24). The algorithm then filters out any stopwords from the detected values (line 30) and returns the remaining values as the final result (line 31).

In addition to identifying relevant values, NLPtoREST also needs to pinpoint parameter names within sentences, which is crucial for detecting dependencies between different parameters. To accomplish this task, NLPtoREST employs the Inter-Parameter Dependency Parser (IPDParser), as outlined in Algorithm 3. This algorithm takes 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...
Algorithm 3 Inter-parameter dependency parser

1: procedure IPDParse(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:      ifCosimSimilarity(word, pm) ≥ 0.7 then
19:        main_param ← main_param ∪ pm
20:          end if
21:    end for
22:    for word in sbar do
23:      ifCosimSimilarity(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 §3)
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

Inter-parameter constraints, parameter names, and values from the main and subordinate clauses of the sentence. To do so, it first generates a constituency parse tree for the given sentence and extracts the main subordinate clauses from this tree (lines 2–6), which forms the foundation for the extraction process. (Note that, for simplicity, the current version of IPDParse in Algorithm 3 does not handle multiple main or subordinate clauses.)

Next, IPDParse 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 [26] 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), in which it inspects each parameter name from the input operation, searching for corresponding terms in the main and subordinate clauses. This matching process uses cosine similarity based on rest2wv, similarly to 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.

Finally, IPDParse returns a tuple that encapsulates the identified relevant terms, parameters, and values for both the main and subordinate clauses (line 39), and which is used to identify 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 [30], 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, IPDParse identifies inter-parameter dependency terms, parameter names, and values for both the subordinate and main clauses, as described above. 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 would then 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, NLPtoREST includes the Rule Validation step, which 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 be checked dynamically.

3.4.1 Static Pruning. Given a set of rules associated with an API operation, the static-pruning phase discards syntactically incompatible rule combinations, that is, 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 [30].

- If a parameter is required (required: true), the same parameter cannot appear in the inter-parameter dependency rules or, OnlyOne, ZeroOrOne, or All10rNone, as these rules would imply that the parameter is not actually required.
- 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.

When an incompatibility in a combination is identified, we 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 a deployed instance of 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 currently 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. It is worth noting, however, that there is a tension between eliminating these cases and possibly discarding valid rule combinations that may lead to the identification of actual failures in the services under test. We plan to explore this tradeoff in future work. 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 the combination 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-optional 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 a 4XX return code. Therefore, the rule would not be discarded during the above described dynamic checking and would 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 further validate the each potentially valid 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 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 successful, it 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 validating rules, fine tuning can also attempt to repair inter-parameter dependency rules or, OnlyOne, All10rNone, and ZeroOrOne. 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 status codes of the responses obtained in the four cases, fine tuning computes a new rule that better complies with the API implementation. In this particular case, the new rule would be OnlyOne(text, data) because requests (2) and (3) are accepted by the API, while requests (1) and (4) result in errors. In particular, the rejection of request (1) indicates that the two parameters cannot be used simultaneously, contrary to what an or rule states.

In summary, our approach considers as validated those rules that belong 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 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 NLtoREST compare with related NLP-based rule-extraction tools?
RQ4: Can the enhanced specification generated by NLtoREST 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 F1 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 (§2). 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 they reached a consensus.

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

To answer RQ3, we compared NLtoREST with RestCT [39], a test generation tool that uses pattern matching to extract inter-parameter dependencies from text descriptions. Specifically, we compared 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 NLtoREST. 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 expected the enhanced specifications generated by NLtoREST to help the testing tools generate a higher number of successful requests (2XX status codes) and a higher number of requests that trigger server errors (5XX status codes), while generating fewer invalid requests (4XX status codes). Consequently, we expected 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: the first one evaluating online testing of REST APIs [28], and the second one comparing automated REST API testing tools [18].

To the best of our knowledge, the first study includes the largest number of commercial REST APIs in the literature, whereas the second one contains the largest number of open-source REST APIs, with 33 REST services in total.

We 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 services but tend to struggle with larger ones [18]. Following the approach of Martin-Lopez, Segura, and Ruiz-Cortés [28], 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 with less than 10k LoC. We also excluded services that we had used to build the NLP model for NLtoREST, so as to avoid potential overfitting. This filtering resulted in a benchmark of 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 problems. Unfortunately, we found that one service, Amadeus v2, had an authentication issue. We contacted the Amadeus 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 (i.e., they limited the number of allowed requests per hour), which caused more than half of the requests in our testing sessions to be blocked. We therefore had to exclude DHL and Marvel and replace YouTube with YouTube Mock (a locally-hosted version of the YouTube service included in one of the benchmark we considered [28]).

After this process, our final benchmark consisted of nine REST services: Federal Deposit Insurance Corporation (FDIC), LanguageTool, OhSome, Open Movie Database (OMDb), REST Countries, Genome Nexus, OCVN, Spotify, and YouTube Mock.

4.2.2 REST API Testing Tools. To select the tools for our study, we evaluated state-of-the-art black-box testing tools for REST APIs that generate tests based on OpenAPI specifications. Specifically, we first selected the top seven performers from the tools we studied in our previous work [18]. We then considered two additional tools, Mostest [23] and RestCT [39], which were developed recently. However, we had to exclude RestCT [39] due to compatibility issues with most of the APIs in our benchmark. (We notified the developers of RestCT about these issues; unfortunately, they were still in the process of addressing them at the time of our experimentation, so we could not include the tool.) Our final set of eight tools includes EvoMasterBB [2], bBOXRT [22], Mostest [23], RESTest [27], REStier [5], RestTestGen [9], Schemathesis [15], and Teases [19].

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 had 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 our experience in previous work [18], which showed that the code coverage achieved by fuzzers tends to plateau within one hour. 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 for each service.

To collect code coverage information, we used JaCoCo [35], which allowed us 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. In the table, Column 2 shows the number of rules in the ground truth for each service. 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 F1 score).
### Table 2: Effectiveness of NLP-based Rule Extraction and Rule Validation.

<table>
<thead>
<tr>
<th>REST Service</th>
<th>No. of Rules in Ground Truth</th>
<th>NLP-based Rule Extraction</th>
<th>Rule Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TP</td>
<td>FP</td>
<td>FN</td>
</tr>
<tr>
<td>FDIC</td>
<td>45</td>
<td>42</td>
<td>3</td>
</tr>
<tr>
<td>Genome Nexus</td>
<td>81</td>
<td>79</td>
<td>3</td>
</tr>
<tr>
<td>LanguageTool</td>
<td>20</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>OCVN</td>
<td>17</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>OhSome</td>
<td>14</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>OMDb</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>REST Countries</td>
<td>32</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>Spotify</td>
<td>88</td>
<td>83</td>
<td>5</td>
</tr>
<tr>
<td>Youtube</td>
<td>34</td>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>Total</td>
<td>333</td>
<td>312</td>
<td>314</td>
</tr>
</tbody>
</table>

The results demonstrate that NLP-based Rule Extraction is effective in extracting rules from the human-readable part of OpenAPI specifications, with an overall recall of 94% (i.e., missing only 21 of the 338 rules in the ground truth). This high recall rate is fairly consistent across services, ranging from a minimum of 88% (for YouTube) to a maximum of 100% (for LanguageTool and OMDb).

The precision results were not as good, with approximately half of the 626 extracted rules being false positives (i.e., 50% precision on average, across services). However, for four of the services (Genome Nexus, OCVN, OMDb, and REST Countries), the technique achieved high precision (96%, 88%, 100%, and 97%, respectively). Most importantly, as we also discuss in §4.3.2, our technique is designed to (1) initially achieve high recall, possibly at the cost of low precision, and (2) improve precision by aggressively filtering the initial set of rules.

We discuss two examples of 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 identifies “OSM” as an example value for the parameter, creating the rule "example: OSM". This is incorrect because “OSM” is an acronym for OpenStreetMap; that is, the parameter description is not providing literal examples for the OSM attributes that could be provided as filters, so no actual example can be extracted from the 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 types (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/or fine-tuning the rest2v model with a larger training dataset.

NLP-based Rule Extraction effectively infers rules from OpenAPI specifications with high recall (88%–100%, 94% on average). However, its precision is lower and varies more across services (16%–100%, 50% on average), supporting the need of a validation phase to reduce false positives.

#### 4.3.2 Q2: 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 F₁ score the table also shows, in parentheses, the percentage difference from the corresponding scores before validation.

Overall, the data show that Rule Validation is effective in improving the accuracy of NLPtoREST. Across services, Rule Validation filtered out 235 of the 314 false positives, causing the precision to increase from 50% to 79%—a 58% improvement. Moreover, this was accompanied by only a small reduction in recall from 94% to 91% (i.e., 9 additional false negatives). It is worth noting that the decrease in recall occurs mostly due to inconsistencies between OpenAPI specifications and API implementations. In general, the validation phase 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.

It is worth mentioning that, although rule validation is effective in improving the precision of rule extraction, the effectiveness of its dynamic part is limited when Rule Validation fails to produce successful requests. Consider, for instance, the incorrect rule example: 0SM inferred by NLP-based Rule Extraction for the OhSome service. Rule Validation was unable to remove this incorrect rule because it failed to generate a valid request for the GET /elements/area operation, which contained the parameter.

Rule Validation eliminates ~75% of the false positives (235 out of 314), increasing precision by 58% (from 50% to 79%), while only causing a 3% decrease in recall.

#### 4.3.3 Q3: 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. Note that, due to compatibility issues with most of the evaluated REST APIs, RestCT was excluded from the RQ4-related experiment. However, these issues did not
Table 3: Improvement in performance of the testing tools considered when fed with enhanced specifications, compared to their baseline performance with original specifications.

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Freq. of 2XX</th>
<th></th>
<th>Freq. of 4XX</th>
<th></th>
<th>Freq. of 5XX</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Enhanced</td>
<td>Original</td>
<td>Enhanced</td>
<td>Original</td>
<td>Enhanced</td>
</tr>
<tr>
<td>bBOXRT</td>
<td>11.5%</td>
<td>20.6% (+79%)</td>
<td>67.4%</td>
<td>77.9% (+15%)</td>
<td>1.1%</td>
<td>1.6% (+56%)</td>
</tr>
<tr>
<td>EvoMaster</td>
<td>14.3%</td>
<td>21.5% (+50%)</td>
<td>67.2%</td>
<td>54.5% (-23%)</td>
<td>18.5%</td>
<td>24.0% (+30%)</td>
</tr>
<tr>
<td>Mostest</td>
<td>5.5%</td>
<td>8.3% (+51%)</td>
<td>73.4%</td>
<td>72.8% (-1%)</td>
<td>21.1%</td>
<td>18.9% (-10%)</td>
</tr>
<tr>
<td>RESTest</td>
<td>55.2%</td>
<td>61.6% (+12%)</td>
<td>34.2%</td>
<td>29.6% (-13%)</td>
<td>10.6%</td>
<td>8.9% (-17%)</td>
</tr>
<tr>
<td>RESTier</td>
<td>14.6%</td>
<td>11.3% (-23%)</td>
<td>56.4%</td>
<td>47.3% (-6%)</td>
<td>35.0%</td>
<td>41.4% (+18%)</td>
</tr>
<tr>
<td>RestTestGen</td>
<td>29.2%</td>
<td>32.1% (+10%)</td>
<td>68.4%</td>
<td>67.8% (-1%)</td>
<td>2.5%</td>
<td>0.1% (-96%)</td>
</tr>
<tr>
<td>Schemathesis</td>
<td>31.2%</td>
<td>36.9% (+18%)</td>
<td>52.5%</td>
<td>49.8% (-5%)</td>
<td>16.3%</td>
<td>13.3% (-18%)</td>
</tr>
<tr>
<td>Traces</td>
<td>12.5%</td>
<td>17.7% (+42%)</td>
<td>46.0%</td>
<td>47.9% (+1%)</td>
<td>41.5%</td>
<td>40.8% (-2%)</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>21.8%</strong></td>
<td><strong>26.2% (+20%)</strong></td>
<td><strong>59.9%</strong></td>
<td><strong>56.0% (-7%)</strong></td>
<td><strong>18.3%</strong></td>
<td><strong>16.6% (+2%)</strong></td>
</tr>
</tbody>
</table>

Overall, these results demonstrate the usefulness of the enhanced specifications generated by NLPtoREST in improving the performance of REST API testing tools. The increase in both the number of successful requests and code coverage suggest that the rules generated by NLPtoREST can have a significant impact on the performance of test generation tools.
results presented in [18] and more recent publications. Additionally, 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 [18] and default settings for all RESTful services during the experiments.

4.4.2 Internal Validity. A first internal threat to the validity of our evaluation is that the training set for the restW2V model might not be representative, which could impact the accuracy of the model. To mitigate this threat, we tried 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.

Another possible internal threat is the manual creation of the ground truth, as the individual interpretation of the specifications by the students involved could have affected the results. To address this, the students were asked to reach a consensus on any discrepancies, and they were not made 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 errors in the implementation of the NLPtoREST technique. To address this threat, we thoroughly examined and tested our code and made it publicly accessible to allow for review (as well as extensions) by others.

5 RELATED WORK
In general, our work is related to research on enhancing OpenAPI specifications and test generation via NLP techniques.

RestCT [40] 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, general, and ultimately effective approach, as shown in our evaluation (see §4.3.3).

ARTE [1] generates test inputs for web APIs by querying the DBpedia knowledge base [5] using parameter names identified by processing parameter descriptions. First, ARTE is more limited in scope than NLPtoREST, as it only generates example input values. Second, ARTE does not take into account the constraints provided in the human-readable part of the specifications, such as default, minimum, and maximum values for parameters. Consequently, even the example input values it identifies are not as good as the OpenAPI example rules that NLPtoREST can generate.

Some existing techniques [8, 41] aim to extract OpenAPI specifications or parts thereof from semi-structured web API documentation pages using a machine-learning based approach. While these methods also aim to generate OpenAPI specifications from documentation, they differ from NLPtoREST in the information they target, the approach they use to extract the information, and the information they extract. As a result, these approaches are mostly orthogonal to our technique.

Other research [4, 6, 7, 29, 33, 37, 38] uses NLP techniques to generate tests from program specifications. While these methods can be effective in generating tests, they operate differently from NLPtoREST and do not aim to enhance OpenAPI specifications.

6 CONCLUSION AND FUTURE WORK
We presented NLPtoREST, a novel approach that (1) extracts information from the human-readable parts of OpenAPI specifications, (2) uses the extracted information to enhance the specifications, and (3) feeds the enhanced specifications to (specification-based) REST API testing tools to improve their performance. Our empirical evaluation show that NLPtoREST is effective and accurate in generating enhanced specifications and that these enhanced specifications can indeed benefit testing tools by allowing them to generate larger numbers of valid requests, trigger more unique server responses, and achieve higher coverage.

There are several possible directions for future work, in addition to performing additional experiments with more services and test generation tools to confirm our current findings. One first direction involves investigating the use of alternative NLP techniques, NLP models, and training datasets for these models. We will also explore the possibility of analyzing server messages, that is, textual responses received by REST servers after they process requests. We believe that such messages contain relevant information that can be used to derive additional rules. As we discussed in the paper, our rule validation approach currently filters out rules that result in 5XX errors. These rules or a subset thereof, however, are likely to be valid and useful to detect errors in the REST APIs under test. We will therefore investigate alternative validation approaches that preserve, in part or completely, rules resulting in 5XX errors; and we will study the tradeoffs between having a higher number of spurious rules and detecting additional REST API errors. An additional future research direction consists of using our validation approach to help developers detect inconsistent or ambiguous specifications. When there is an inconsistency between a rule and the corresponding implementation or specification, NLPtoREST could present that information to the developers as a possible problem with the implementation or the specification, instead of simply discarding the rule as it currently does. A final future work direction is the investigation of how generative AI techniques (e.g., [13, 31, 36]) could help refine and expand the capabilities of NLPtoREST.

ACKNOWLEDGMENT
We thank the developers of the testing tools we used in our empirical evaluation (i.e., EvoMasterBB, hBOXRT, Morest, RESTest, RESTler, RestTestGen, Schemathesis, Tcases, and RestCT) for making their tools available. A special thank goes to the developers of RestCT for offering to investigate and attempting to fix the issues we encountered when using their tool. This research was partially supported by NSF, under grant CCF-0725202, DARPA, under contract N66001-21-C-4024, DOE, under contract DE-FOA-0002460, the European Union’s Horizon Europe research and innovation programme, under grant 101070238, the Italian Ministry of University and Research, under the PNRR programme for the Interconnected Nord-Est Innovation Ecosystem (iNEST), the Italian Ministry of University and Research, under agreement 40-G-14702-3 for the PON programme for Research and Innovation (Action IV.6), and gifts from Facebook, Google, IBM Research, and Microsoft Research. This publication reflects the views only of the authors, and the sponsoring agencies cannot be held responsible for such views and any use which may be made of the information contained therein.