Testing RESTful APIs: A Survey

AMID GOLMOHAMMADI and MAN ZHANG, Kristiania University College, Norway
ANDREA ARCURI, Kristiania University College and Oslo Metropolitan University, Norway

In industry, RESTful APIs are widely used to build modern Cloud Applications. Testing them is challenging, because not only do they rely on network communications, but also they deal with external services like databases. Therefore, there has been a large amount of research sprout in recent years on how to automatically verify this kind of web services. In this article, we present a comprehensive review of the current state-of-the-art in testing RESTful APIs based on the analysis of 92 scientific articles. These articles were gathered by utilizing search queries formulated around the concept of RESTful API testing on seven popular databases. We eliminated irrelevant articles based on our predefined criteria and conducted a snowballing phase to minimize the possibility of missing any relevant paper. This survey categorizes and summarizes the existing scientific work on testing RESTful APIs and discusses the current challenges in the verification of RESTful APIs. This survey clearly shows an increasing interest among researchers in this field, from 2017 onward. However, there are still a lot of open research challenges to overcome.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Search-based software engineering;**

Additional Key Words and Phrases: Survey, literature review, REST, API, testing, test case generation, fuzzing, web service

ACM Reference format:
Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 27 (November 2023), 41 pages.
https://doi.org/10.1145/3617175

1 INTRODUCTION

When building the backends for web and enterprise applications, using RESTful APIs [94] is a common choice, especially in microservices architecture [138]. Many companies use them for their backends, such as Netflix, Uber, Airbnb, eBay, Amazon, Twitter, Nike, and so on [144].

Not only RESTful APIs are used to build the backends of enterprise applications, but also many APIs are directly available on the internet, providing all different kinds of functionalities. For example, there are several thousands of Web APIs [2, 28, 29], where RESTful ones are the most common (other kinds are, for example, the old SOAP [86] and the more recent GraphQL [18]). Several

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 864972).
Authors' addresses: A. Golmohammadi and M. Zhang (coresponding author), Kristiania University College, Kirkegata 24-26, 0153 Oslo, Norway; emails: {amid.golmohammadi, man.zhang}@kristiania.no; A. Arcuri, Kristiania University College and Oslo Metropolitan University, Oslo, Norway; email: andrea.arcuri@kristiania.no.

1049-331X/2023/11-ART27 $15.00
https://doi.org/10.1145/3617175

companies provide APIs to their services on the internet using REST, such as Google,[1] Amazon,[2] Twitter,[3] Reddit,[4] and LinkedIn.[5]

However, verifying the correctness of Web APIs is quite challenging [73, 75]. Not only does a tester need to create network messages (e.g., using HTTP over TCP) toward the API, but typically there is also the need to set up the right data into the databases and possibly mock interactions with other external services [50]. Due to their wide use in industry, it is hence not surprising that there has been a lot of attention from the research community on the development of novel techniques to test REST-based applications in recent years. However, except for one survey [90] with a limited number of covered papers and research questions, we could not find any relevant literature survey that covers the state-of-the-art on this topic. Therefore, to help carry out further research endeavors on this topic and identify research gaps, in this article, we survey and categorize the current state-of-the-art of the scientific literature on testing RESTful APIs. We believe that researchers can benefit from knowing to what extent different testing kinds have been covered and which ones have been successfully applied to real-world case studies. In particular, in this article, we aim at answering the following 12 research questions from four perspectives based on a selection of 92 scientific articles:

— Publication status in testing RESTful APIs
  – **RQ1:** How many papers were published per year?
  – **RQ2:** In which venues were the papers published?
  – **RQ3:** What contributions to testing RESTful APIs have been made by the papers in this area, and what are their frequencies?
— Existing approaches in supporting automated testing of RESTful APIs
  – **RQ4:** What metrics are used to evaluate the effectiveness of the testing?
  – **RQ5:** What techniques are used for automatically testing RESTful APIs?
  – **RQ6:** What kind of testing has been automated for RESTful APIs?
  – **RQ7:** What kind of artifacts are used for conducting empirical evaluations?
— Available Tools for Testing RESTful APIs
  – **RQ8:** Which research tools are open-source?
  – **RQ9:** Which non-research tools are used/compared?
  – **RQ10:** Which features are supported by the research prototypes?
— Addressed and Open Challenges
  – **RQ11:** Which research challenges are addressed?
  – **RQ12:** Which open research challenges are identified?

The article is organized as follows: Section 2 discusses important background information, needed to better understand the rest of the article. Related work is discussed in Section 3. How the 92 articles were selected is described in Section 4. Our research questions are answered in Section 5 (publication status, RQ1–3), Section 6 (approaches, RQ4–7), Section 7 (tools, RQ8–10), and Section 8 (challenges, RQ11–12). A summarizing discussion based on our survey's results is presented in Section 9. Threats to validity are discussed in Section 10. Section 11 concludes the article.

[1] https://developers.google.com/drive/v2/reference/
[2] http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
[3] https://dev.twitter.com/rest/public
[4] https://www.reddit.com/dev/api/
[5] https://docs.microsoft.com/en-us/linkedin/

2 BACKGROUND

2.1 Hypertext Transfer Protocol and REST

HTTP (Hypertext Transfer Protocol) is an application-layer protocol for hypermedia information systems that are distributed and collaborative over a computer network. Through the extension of its request methods, error codes, and headers, this generic and stateless protocol can be used for many tasks other than hypertext, such as name servers and distributed object management systems [93].

REST (Representational State Transfer) is an architectural style that was introduced by Fielding in 2000 [94] that is followed by many modern web APIs. REST is not a protocol, as it just defines a set of guidelines for designing APIs for accessing and manipulating resources using HTTP over the network.

To reduce interaction latency, enforce security, and encapsulate legacy systems, REST stresses scalability of component interactions, generality of interfaces, autonomous deployment of components, and intermediary components. REST does not enforce any rules on how it should be implemented at the lower level. Instead, it defines a set of high-level design constraints, including separation of concerns related to client and server, statelessness, the ability to cache data, having a uniform interface between components, multilayeredness, and code-on-demand (this latter constraint is optional). It encourages users to come up with their own solutions, as it is neither a protocol nor a standard. This means that it can be implemented in a variety of ways and there is no enforcement to adopt any specific design pattern. For instance, REST does not force the application to embrace *SOLID* [139] principles such as *Dependency Inversion*. Despite it being better to adhere to those kinds of design principles to have more sustainable applications, none of the REST constraints gets violated in case a high-level module (e.g., the code that defines API endpoints) is tightly coupled to a low-level module (e.g., a module to connect to an SQL database). A REST API can be implemented with different programming languages (e.g., Python,[6] C#[7]) and frameworks (e.g., Express[8] and Ruby on Rails[9]).

In a RESTful API, any piece of information that we can think of can be referred to as a resource. A document or image, for example, can be a REST resource, as can a temporal service, a collection of other resources, or a non-virtual object (e.g., an employee) [30]. A RESTful API sends a *representation* of the resource's state to the caller when a client request is made. **JSON (Javascript Object Notation)**, HTML, and plain text are among common data formats that can be sent via HTTP. Among them, JSON is the most widely used file format [137] because, contrary to its name, it is language-independent and understandable by both humans and machines.

To specify the desired action to be taken for a certain resource, HTTP defines a number of request methods, commonly referred to as HTTP verbs. These verbs, such as `GET` and `POST`, allow multiple actions to be performed on a resource. For example, a list of employees can be retrieved by invoking the URL for employees (e.g., "`/employees`," which is appended to the base URL of the API) with `GET`. As another example, a single employee can be fetched by calling its URL (e.g., "`/employees/42`", where 42 is the identifier of the intended employee). Similarly, a new employee can be added by invoking the same URL for retrieving the list of employees with `POST` verb and including the data of the new employee in the body payload of the HTTP request. There are three other HTTP verbs that are used frequently within RESTful APIs, including `PUT`, `PATCH`, and `DELETE`.

[6] https://www.python.org/
[7] https://docs.microsoft.com/en-us/dotnet/csharp/
[8] https://expressjs.com/
[9] https://rubyonrails.org/

These verbs are utilized for updating by replacing, updating by modification, and deleting a resource, respectively.

2.2 OpenAPI

The **OpenAPI Specification (OAS)**[10] defines a common, language-independent interface to RESTful APIs [26]. It enables both humans and machines to learn about and comprehend the capabilities of the service without having access to the service's source code or network traffic analysis. A user can comprehend and use a remote service with minimal implementation logic when it is properly defined. An OpenAPI specification can then be utilized by automated tools for creating user-friendly documentation (e.g., an interactive web page) to display the API, tools for creating code to generate server stubs and client libraries in different programming languages, tools for testing, and many more use cases [25].

The OAS defines an OpenAPI document as an object that may be expressed in either JSON or YAML. Primitive data types in the OAS are based on the types supported by JSON specification [22]. Figure 1 displays a sample OpenAPI document in JSON format. It gives information to the client on how to invoke endpoints in this API, such as which URIs and which HTTP verbs must be used to invoke that specific endpoint and also what parameters might be needed and their data types. In this example (Figure 1), there are only two simple `GET` endpoints: The first one's URL is "`/employee`", which does not take any input parameters. The second one has the same URL, in addition to an input parameter of type `integer`.

OpenAPI is one of the widely used techniques to define the schema of a REST API. There are other languages that can be used to describe APIs, such as RAML[11] and APIBluePrint.[12] However, they are not as commonly used as OpenAPI, especially for the purpose of *fuzzing*. Fuzzing is the process of creating and running tests automatically with the intention of identifying flaws [153]. Finding inputs that cause inappropriate program execution as a result of improper input data processing is the major goal of fuzzing [170]. Many fuzzing tools [52, 60, 104, 113] rely on OpenAPI specification, as it outlines how to use a REST API, including the types of requests it can handle, the possible responses, and the format of the possible responses.

3 RELATED WORK

In software engineering research, there are several surveys that summarize, categorize, and analyze the current state-of-the-art on different research topics. Examples include search-based test case generation [48], search-based mutation testing [148], software-testing education [97], test case selection and prioritization using machine learning [141], software testing effort estimation [71], software robustness assessment [114], oracle problem in software testing [67], flaky tests [142], and software engineering for AI-based systems [132].

Regarding the testing of web services, some old surveys (from 2009 and 2013) analyzed the challenges of testing service-oriented architectures [73, 75]. Lack of real-world case studies is mentioned as an important problem in the latter one [73]. It is understandable that at the time of writing that survey (i.e., 2013) there were not that many open-source APIs available, as well as not many ways to find out and discover existing APIs on the internet (e.g., References [2, 28, 29]). In Reference [73] it is mentioned that 71% of the papers provide no experimental results and only 11% of them use real-world case studies. In our survey, only 14% do not perform any empirical

[10] https://swagger.io/specification/
[11] https://raml.org/
[12] https://apiblueprint.org/

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "Employees API",
    "version": "v1"
  },
  "paths": {
    "/employee/{id}": {
      "get": {
        "tags": [
        "Employee"
        ],
        "parameters": [
        {
          "name": "id",
          "in": "path",
          "required": true,
          "schema": {
            "type": "integer",
            "format": "int32"
          }
        }
        ],
        "responses": {
          "200": {..},
          "400": {..},
          "401": {..}
        }
      }
    },
    "/employee": {
      "get": {
        "tags": [
        "Employee"
        ],
        "responses": {
          "200": {..},
          "400": {..},
          "401": {..}
        }
      }
    }
  },
  "components": {..}
}
```

Fig. 1. An example of OpenAPI 3.0 schema.

evaluations and 16% take advantage of artificial artifacts that are developed by the researchers and are not real-world case studies. This shows a clear increase in maturity and soundness in the research efforts aimed at this topic in the past decade.

As those two surveys are old, they do not represent the current state-of-the-art, especially considering the blossoming of research activities from 2017 on (which will be discussed in more detail in Section 5.1). Moreover, the older survey [75] does not even mention REST API testing, but rather focuses on XML-based SOAP services. However, few challenges are similar and general for the different types of web services, like the possibly very high cost of testing online live-services, which was identified as one of the main challenges in Reference [75].

There is currently another short survey on the testing of RESTful APIs [90], published in the *Applied Sciences* journal in 2022. Such survey is based on 16 published articles, addressing the following research questions:

— RQ-1: What are the main challenges in generating unit tests for RESTful APIs?
— RQ-2: What are the code coverage concerns when it comes to testing RESTful APIs?
— RQ-3: What solutions are currently available to meet testing and unit test generation challenges?
— RQ-4: What support do solutions provide for authentication-enabled RESTful APIs' testing and unit test generation?

Our survey is much larger, covering 92 articles instead of just 16 (and all these 16 are included in our analyses). Furthermore, compared to those four RQs listed in Reference [90], we answer many more research questions from various perspectives (e.g., testing metrics, testing kinds, existing fuzzers and challenges) to provide a better overview of the current state-of-the-art in this domain.

4 RESEARCH METHOD

4.1 Research Questions

To investigate current research in addressing REST API testing, we conducted a survey to answer the following research questions from four perspectives:

— Publication status in testing RESTful APIs
  – **RQ1:** How many papers were published per year?
  – **RQ2:** In which venues were the papers published?
  – **RQ3:** What contributions to testing RESTful APIs have been made by the papers in this area, and what are their frequencies?
— Existing approaches in supporting automated testing of RESTful APIs
  – **RQ4:** What metrics are used to evaluate the effectiveness of the testing?
  – **RQ5:** What techniques are used for automatically testing RESTful APIs?
  – **RQ6:** What kind of testing has been automated for RESTful APIs?
  – **RQ7:** What kind of artifacts are used for conducting empirical evaluations?
— Available Tools for Testing RESTful APIs
  – **RQ8:** Which research tools are open-source?
  – **RQ9:** Which non-research tools are used/compared?
  – **RQ10:** Which features are supported by the research prototypes?
— Addressed and Open Challenges
  – **RQ11:** Which research challenges are addressed?
  – **RQ12:** Which open research challenges are identified?

4.2 Database and Search Queries

To find relevant papers, we took advantage of seven databases as listed in Table 1. These online repositories of peer-reviewed articles were selected based on their popularity and extent of relevance to software engineering research. These sources contain well-known conferences and journals in the field. Existing surveys in software engineering research (e.g., References [71, 141]) widely referenced them, as they offer a variety of authoritative publication venues in the field.

To achieve the objectives of this work and answer our research questions, we put together our search terms according to the specific format of each database. The queries were formulated around the concept of applying testing on RESTful APIs. In some databases, such as *IEEE* and *ACM*, to have more relevant results, we excluded the papers that do not contain at least one of the commonly used terms in the literature in any part of them (e.g., by using *Anywhere* in ACM) including *black box*, *white box*, *fuzzing*, *fuzzer*, *unit test*, *system test*, *end to end test,* and *integration test*. The numbers

Table 1. Selected Databases, with Search Queries and Number of Found Articles

Name	Search Query	Found Papers
IEEE	(“Document Title”:restful OR “Index Terms”:restful OR “Document Title”:rest OR “Index Terms”:rest) AND (“Document Title”:test* OR “Index Terms”:test*) AND (“Full Text .AND. Metadata”:“white box” OR “Full Text .AND. Metadata”:“black box” OR “Full Text .AND. Metadata”:fuzz* OR “Full Text .AND. Metadata”:“unit test*” OR “Full Text .AND. Metadata”:“integration testing” OR “Full Text .AND. Metadata”:“system test*” OR “Full Text .AND. Metadata”:“end to end test*”)	85
ACM	(Title:(restful) OR Keywords:(restful) OR Title:(rest) OR Keywords:(rest)) AND (Title:(test*) OR Keyword:(test*)) AND (Anywhere:(white box) OR Anywhere:(black box) OR Anywhere:(fuzz*) OR Anywhere:(“unit test”) OR Anywhere:(“unit tests”) OR Anywhere:(“unit testing”) OR Anywhere:(“integration test”) OR Anywhere:(“integration tests”) OR Anywhere:(“integration testing”) OR Anywhere:(“system test”) OR Anywhere:(“system tests”) OR Anywhere:(“system testing”) OR Anywhere:(“end to end test”) OR Anywhere:(“end to end tests”) OR Anywhere:(“end to end testing”))	18
ScienceDirect	Title, abstract, keywords: (restful OR “rest api” OR “rest apis” OR “rest service” OR “rest services” OR “rest web services”) AND (test OR testing OR tests) Subjects: Computer Science	17
Wiley	(test*) AND (restful OR “REST API” OR “REST APIs” OR “REST service” OR “REST services” OR “REST web service” OR “REST web services”) Subjects: Computer Science	2
Web of Science	(TI=(restful) OR AK=(restful) OR TI=(“rest api”*) OR AK=(“rest api”*) OR TI=(“rest service”*) OR AK=(“rest service”*) OR TI=(“rest web service”*) OR AK=(“rest web service”*)) AND (TI=(test*) OR AK=(test*))	12
MIT Libraries	Abstract contains: (restful OR “rest api*” OR “rest service*” OR “rest web service*”) AND (test*) AND Any field contains: “black box” OR “white box” OR “unit test*” OR “integration test*” OR “system test*” OR “end to end test*” OR fuzz*	19
Springer	restful AND test AND api Articles within Computer Science and Software Engineering/Programming and Operating Systems	90

of papers found as the result of search queries are shown in Table 1. The search was conducted on March 1, 2022.

Each repository had its own limitations to conduct advanced searches. For example, unlike IEEE and ACM, it did not seem to be possible to formulate an advanced search query in Springer. At the time of writing this survey, there was no feature to limit the results based on different sections of the paper (e.g., Title or Abstract) in Springer. Therefore, the only option we had was to do a broad search query and limit it by subject (i.e., *Computer Science* and then *Software Engineering/Programming and Operating Systems*) and type of the study (i.e., *Article*). In *ScienceDirect*, there was a limitation for Boolean operators. It did not allow more than eight Boolean operators, so we were not able to include terms such as *SBST*. Another impediment with this repository was that it did not support wildcards.

4.3 Paper Selection Criteria

Table 2 introduces the inclusion criteria we used to select papers. In general, we chose papers that are written in English and that are related to testing in the domain of REST APIs. However, to be included, an article did not have to be exclusively in the domain of REST. It could be generally about web services, cloud services, and so on, which might encompass RESTful APIs as well. We excluded

Table 2. Inclusion and Exclusion Criteria

	Criterion
1	The paper is in English
2	The paper is related to testing
3	The paper is about REST. It does not have to be exclusively about REST.
4	The paper is not a thesis.
5	The paper is not a survey or a systematic literature review.

theses (e.g., MSc and PhD). We also excluded existing surveys, as those are rather discussed in our Related Work (Section 3).

4.4 Snowballing

The idea behind conducting this phase was to employ a hybrid search strategy to reduce the chance of missing important relevant papers. To do so, we conducted forward and backward snowballing on June 14, 2022. Forward and backward snowballing refer to checking the reference list and citations of a paper, respectively. To find citations of a paper, we took advantage of *Google Scholar*.[13] By studying and evaluating them based on the defined inclusion criteria, we finally gathered a total of 92 papers, including 40 that were initially found by search and also 52 new ones by conducting snowballing.

There are several reasons why the number of papers found in this step is considerably larger than that of found during the initial search. First, some of the papers were published after March 1, 2022, which is the date we conducted the initial search. This group includes 16 papers. Second, 15 papers were only available in sources that were not among those selected for the initial search (see Table 1). These sources include *arXiv*, *Penn State*, *KSIResearch*, *Open Journals*, *unam.mx*, *Semantic Scholar*, *sistedes.es*, and *Macrothink Institute*. The third reason could be that the approach might not be specific to REST, and the term REST might not be present in either the title, abstract, or keywords. However, during the snowballing phase, if we found papers where the proposed approach was evaluated on REST APIs, then we included them in this survey. In addition, there might exist some bugs in the search services provided by the article databases. For instance, by taking a look at the paper cited in Reference [171], we found out that this should have appeared in results of the initial search, as it fits our search query, but surprisingly did not. Then, we reported such issues to the service (i.e., ACM), and they have been confirmed and fixed.

4.5 Data Extraction

Table 3 contains the types of extracted data for each research question. We designed a spreadsheet in Google Sheets to put together collected data from the selected papers. This included name of the paper, a unique given ID, and information related to answering research questions. Before conducting the data extraction, we prepared a list of possible categories and relevant keywords to search for (e.g., *black-box* and *white-box*) and refined our selection while investigating each paper. However, as we started studying the papers, we discovered some other possible values. For example, regarding RQ5, we found out that not all the papers use either black-box or white-box techniques, and also they can both be supported at the same time (e.g., Reference [53]).

To conduct the data extraction, the papers were divided between two of the authors. Each of them extracted the data based on the types shown in Table 3. Then, the results were double-checked by the other author. In case of any discrepancy, the issue was discussed and settled by the third author.

[13] http://scholar.google.com/

Table 3. Type of Data Extracted per Research Question

RQs	Type of Extracted Data
RQ1	The number of papers per year
RQ2	The number of papers per venue
RQ3	(1) Types of contribution of the papers, and (2) The number of papers per contribution type
RQ4	Existing metrics that have been applied for guiding/evaluating REST API testing
RQ5	Existing testing techniques for REST APIs
RQ6	Types of testing that REST API testing approaches support
RQ7	(1) Types of artifacts that have been used for conducting evaluation of REST API testing and (2) Locations and availability of the artifacts
RQ8	(1) REST API testing approaches that have the open-source prototype and (2) Replicability of experiments conducted with the prototypes
RQ9	Existing non-research tools that have been applied for REST API testing
RQ10	Features that have been supported by research prototypes of REST API testing
RQ11	Research questions and hypotheses that have been studied in REST APIs testing
RQ12	Open challenges that have been identified in testing REST APIs

5 STATUS OF PUBLICATIONS IN REST API TESTING

To study the trends of research in REST API testing, in this section, we report our investigation on existing studies of REST API testing with academic publications. These findings are categorized in terms of time (RQ1), venues (RQ2), and main contributions (RQ3).

5.1 RQ1: How many papers were published per year?

Figure 2 illustrates the number of papers published from 2009 to 2022. It shows that there has been an upward trend in the number of papers in the domain of RESTful APIs testing during recent years.

There was not a considerable number of studies before 2017, as the number has fluctuated between 0 and 2. However, the quantity of published papers has increased dramatically from 2017 onward. In 2018, 8 papers were published, and this number further increased and reached to the highest, 24, in 2021. At the time of finalizing the selected papers for this survey (June 2022), only half of the year 2022 has passed. This is the reason why the number of papers published in 2022 is lower than that of 2021.

RQ1: *Since 2017, there has been a dramatic increase in the number of scientific studies on testing REST APIs.*

5.2 RQ2: In which venues were the papers published?

Papers selected for this survey have been published in a variety of venues. Table 4 provides information about the number of papers being published in each venue and its type. A venue can be of type conference, workshop, journal, or other open-access repositories. The table is sorted in descending order based on the number of papers. To keep the table short, we include the full name of a venue only if: (1) more than one paper is published in the venue *or* (2) a journal article is published by one of the main publishers in the field of software engineering (i.e., ACM, Elsevier, IEEE, Springer, Wiley). For other venues, we categorized them into *Other Conferences* and *Other Journals* based on the venue type.

Based on Table 4, 92 studies of REST API testing have been published in 63 different venues (i.e., 43 conferences, 14 journals, 5 workshops, and 1 open access repository), which have covered well-known top SE venues, such as TOSEM, TSE, EMSE, and ICSE. Regarding conference

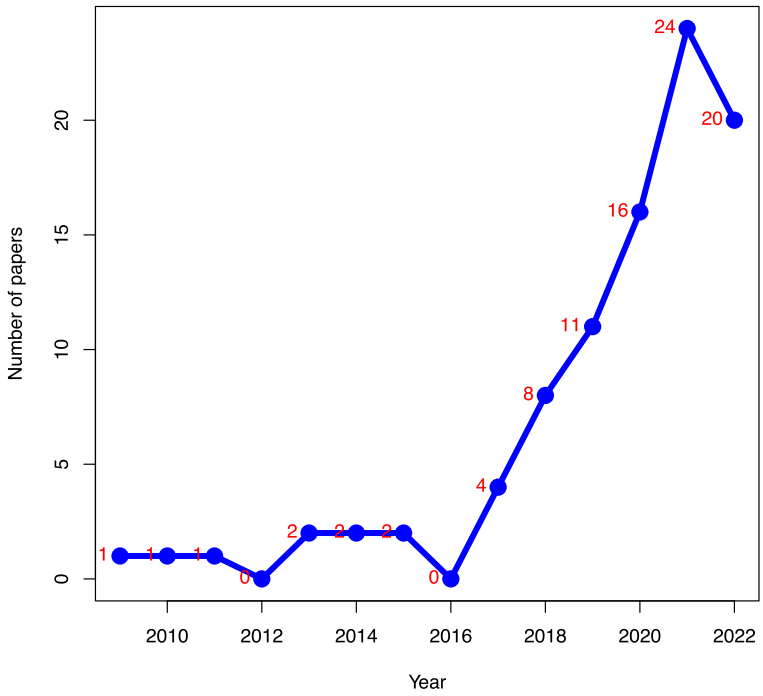

Fig. 2. Number of selected papers (y-axis) per year (x-axis).

publications, there are in total 61 papers. On top, with 38 papers, there is *Other Conferences*, which aggregates papers from conferences with a single publication. The most common venues are ICSE (*ACM/IEEE International Conference of Software Engineering*) and ICST (*IEEE International Conference on Software Testing, Verification and Validation*), with 7 and 6 papers. In addition, 20 articles have been published in journals, and the most frequent journal is *TOSEM*. It is a peer-reviewed journal published by ACM, which stands for *ACM Transactions on Software Engineering and Methodology*. Besides, there also exist 6 papers that have been submitted to a non-peer reviewed open access repository, i.e., arXiv. As of the time of writing, 3 of these papers have been published in peer-reviewed venues since the collection of our data paper [110, 118, 173]. Article [110] was published at ISSTA'22 [111], article [118] at ICSE'22 [119], and article [173] in TOSEM'23 [174]. Moreover, there are 5 papers that have been presented at 5 different workshops.

RQ2: *From 2009 to 2022, scientific work on testing RESTful APIs was published in 63 different venues, where ICSE/ICST and TOSEM were the most frequent conferences and journal, respectively.*

5.3 RQ3: What contributions to testing RESTful APIs have been made by the papers in this area, and what are their frequencies?

To better study what contributions have been made in existing studies, we defined 5 different categories to classify main contributions of selected papers, as follows:

— **New automated approach and its extension**
This category includes papers that propose a new tool, algorithm, framework, or method for automated testing of REST APIs and its integrated extensions for enhancing the automated approach. For example, regarding automated test case generation for REST APIs, Arcuri

Table 4. Number of Papers per Venue and Their Type

	Venue	Number of Papers	Venue Type
1	*Other Conferences*	**38**	Conference
2	*Other Journals*	**8**	Journal
3	*ICSE*	**7**	Conference
4	*arXiv*	**6**	Open-access Repository
5	*ICST*	**6**	Conference
6	*Other Workshops*	**5**	Workshop
7	*TOSEM*	**5**	Journal
8	*QRS*	**4**	Conference
9	*IEEE TSE*	**3**	Journal
10	*ESEC/FSE*	**2**	Conference
11	*GECCO*	**2**	Conference
12	*ISSTA*	**2**	Conference
13	*Applied Soft Computing*	**1**	Journal
14	*EMSE*	**1**	Journal
15	*IEEE Software*	**1**	Journal
16	*IEEE TSC*	**1**	Journal

proposed the **Many Independent Objective (MIO)** algorithm specific to white-box system level test generation [51] and enabled **search-based software testing (SBST)** of REST APIs with MIO [52], implemented as an open-source fuzzer, named EvoMaster. To further improve the SBST of REST APIs, SQL handling [55] and testability transformation [57] were integrated into EvoMaster. However, such extension techniques can be used outside of REST APIs, i.e., applied for testing of other domains. Another example for test data generation is *ARTE*, which is an approach presented in Reference [156] to automatically extract realistic data inputs for REST APIs from knowledge bases (e.g., DBpedia) by taking advantage of a number of techniques, including natural language processing, search-based, and knowledge extraction. To make ARTE fully automated, it is integrated into RESTest [131].

— **New analysis approach and potential solution**
Studies that, instead of proposing a new approach for directly achieving automated testing, help to analyze testing related aspects (such as coverage metrics) and identify potential solutions for testing of REST APIs, fall into this category. For instance, Marculescu et al. [123] studied faults in RESTful APIs selected from the EMB repository [10] and proposed a taxonomy of the faults identified in the REST API with EvoMaster [123]. Martin-Lopez et al. [129] defined a set of coverage metrics based on API schema in the context of black-box testing of REST APIs. Katt and Prasher [70] identified potential security threats in REST APIs and proposed corresponding quantitative security assurance metrics. In addition, Soni et al. [149] proposed a framework for allowing mocking external dependencies of Java REST APIs. However, this mocking solution is for unit testing, and it could be potentially adapted to automated testing approaches for enabling additional handling of external services.

— **Empirical study on various automated approaches**
This group of papers focuses on comparing existing approaches by conducting empirical evaluations. For instance, the study conducted by Corradiani et al. [83] compared automated black box approaches for testing REST APIs through an empirical comparison. There also exist studies for comparing existing fuzzers for REST APIs [110] and analyzing open problems [173] with the fuzzers.

Table 5. Types of Paper Contributions and Their Numbers

Contribution Type	#	Papers
NA: New automated approach and its extension	61	[49, 52–57, 59–61, 63, 64, 66, 68, 74, 77, 78, 85, 89, 92, 98, 100, 101, 104, 107, 109, 112, 113, 115–118, 120–122, 126, 130, 131, 134, 136, 140, 143, 145, 147, 152, 154–157, 160, 161, 163, 166, 167, 169, 171, 172, 175–178]
NP: New analysis approach and potential solution	17	[70, 76, 80, 81, 108, 123, 125, 127–129, 133, 149, 151, 158, 159, 162, 168]
TD: Tool implementation or demonstration	7	[58, 62, 72, 84, 150, 164, 165]
PI: Proposal or idea	4	[69, 105, 106, 124]
EA: Empirical study on various automated approaches	3	[83, 110, 173]

— **Tool implementation or demonstration**
Papers that mainly focus on implementation of a testing tool, or show how a tool can be used, fall into this category. For instance, RESTATS is introduced in Reference [84], which is a tool to compute coverage metrics for black-box testing of REST APIs. This tool adopts the coverage metrics proposed by Martin-Lopez et al. [129].

— **Proposal or idea**
These papers do not include any conducted study. Instead, they suggest new ideas or plans to carry out research on the testing of REST APIs. For instance, Martin-Lopez in Reference [124] planned to develop a framework for specification-driven testing that would automatically create complex test cases for Web APIs and also intelligent programs (called "bots") capable of generating a large number of inputs.

The data provided in Table 5 shows the number of selected papers based on their main contribution. "New automated approach and its extension" has the highest number with 61, which is equal to 66% of the papers. "New analysis approach and potential solution" is the second-largest category and comprises almost one-fifth of the papers (i.e., 18%) by 17. The three other categories are much smaller (e.g., 3 for "Empirical study on various automated approaches").

We also investigated how the types of papers have evolved over the years. Figure 3 shows the number of papers of each contribution type in chronological order. In such a figure it can be seen that all types have increased in frequency, despite experiencing some fluctuations in some cases. The most significant rise is seen in "New automated approach and its extension," which indicates that there has been an avid interest in automating the process of REST API testing.

More details on the different categories and type contributions are presented and analyzed in the following research questions:

RQ3: *5 different contributions were found from the main contributions of the selected 92 papers. 66% of the papers propose a new automated approach and its extension for testing RESTful APIs, while 18% of the papers are new analysis approach and potential solution of REST API testing.*

6 APPROACHES TO TESTING REST API

One of the main objectives of this survey is to identify existing testing approaches that have been developed to be automated in the context of REST APIs.

Figure 4 represents a high-level abstraction of testing of REST APIs. A RESTful API typically exposes a schema about how to access the web service. The schema could be represented in various ways (e.g., JSON, XML, formal model), and one popular technique to define the schema is OpenAPI, as discussed in Section 2.2. As the schema defines the structure of the resources handled by SUT and the available actions to access these resources [26, 177], it is often used as an input to test the REST API, e.g., define metrics as test criteria, automatically generated tests (referred to as the

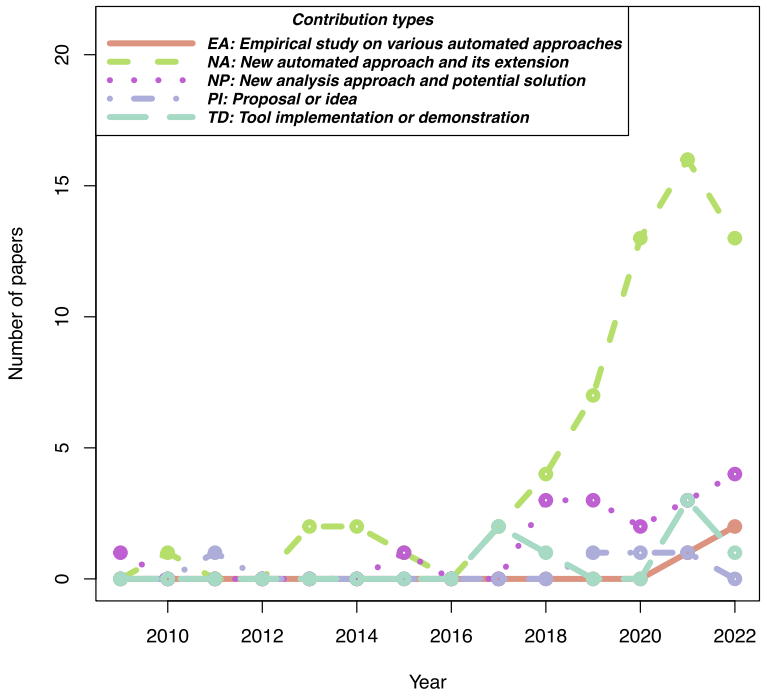

Fig. 3. Number of papers based on contribution type in a chronological order.

term *fuzzing* [99]). The test generation is typically guided by heuristics aimed at optimizing those metrics. Such metrics could be linked with behaviors of the SUT and data produced by the SUT offline or at runtime. A test for a REST API can be regarded as a sequence of HTTP requests, e.g., a sequence of `POST /foo` and `GET /foo/42`, as shown in Figure 4. To perform actions on the SUT, each request has concrete values (referred to as test data, e.g., 42) for its parameters, such as *path parameters*, *query parameters*, and *body payload*, if specified in the schema. In addition, endpoints of REST APIs might be restricted with *authentications*, and the REST API could also connect to *databases* and *external web services* (as shown in Figure 4). How to configure the authentication and handle such external services are also part of REST API testing.

To study the existing approaches of REST API testing, we designed RQs 4–7 and reported the results of our investigation on metrics, which include test criteria and heuristics (Section 6.1), techniques (Section 6.2), kinds of testing that have been applied (Section 6.3), and available artifacts used in the empirical evaluations (Section 6.4).

6.1 RQ4: What metrics are used to evaluate the effectiveness of the testing?

With this survey, based on the selected papers, we classified metrics into three types, i.e., *coverage* criteria, *fault detection*, and *performance*, as shown in Figure 5, and statistics of each type are shown in Figure 6. These are discussed next in more detail.

6.1.1 Coverage Criteria.

(1) schema-related coverage.

API schemas are the main inputs that guide how to access the SUT and define what responses should be returned. For instance, as an example of the schema defined with the OpenAPI (Figure 1), an endpoint could be defined under a `URI path` with an HTTP verb (e.g., `POST`, `GET`), input

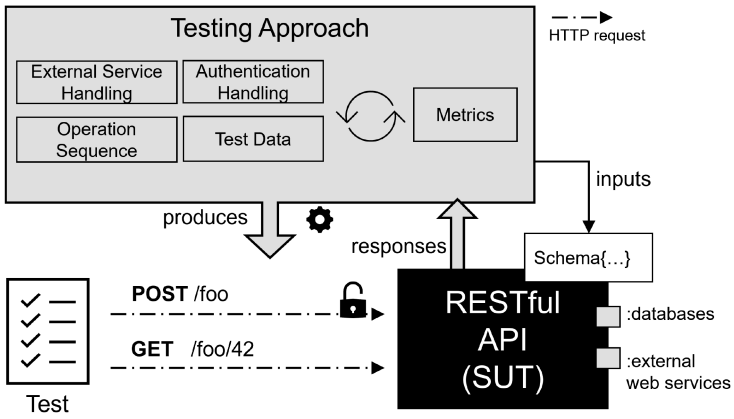

Fig. 4. High-level view of REST API testing.

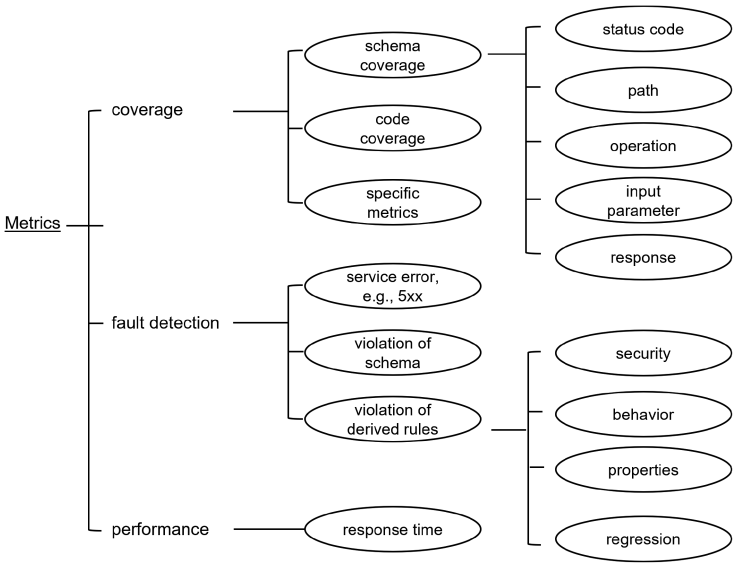

Fig. 5. Metrics employed for REST API Testing in the literature.

parameters (e.g., `Path Parameter`), and possible responses (e.g., status code, response body). Then, there exist several black-box coverage metrics that are defined based on those elements [49, 52–57, 59, 60, 63, 68, 85, 89, 104, 107, 109, 113, 118, 126, 130, 131, 145, 152, 155–157, 160, 166, 167, 171, 172, 176, 177]:

— *HTTP status code* could reflect a result of processing the request in the SUT, such as 2xx often represents a successful request. A set of testing criteria has been defined to compute a coverage of status codes that are returned during testing for each endpoint [49, 52–57, 59, 60, 63, 68, 85, 104, 107, 109, 113, 118, 126, 130, 131, 145, 152, 156, 157, 160, 166, 171, 172, 176, 177].
— *path* provides the information to access the API, i.e., the full URI to access an endpoint could be constructed by *base path* plus *path*. For instance, Martin-Lopez et al. [129] defined *path coverage*, which assesses a number of paths accessed by the generated tests out of the total available paths in the schema. Banias et al. [63] evaluated *paths tested* by considering success

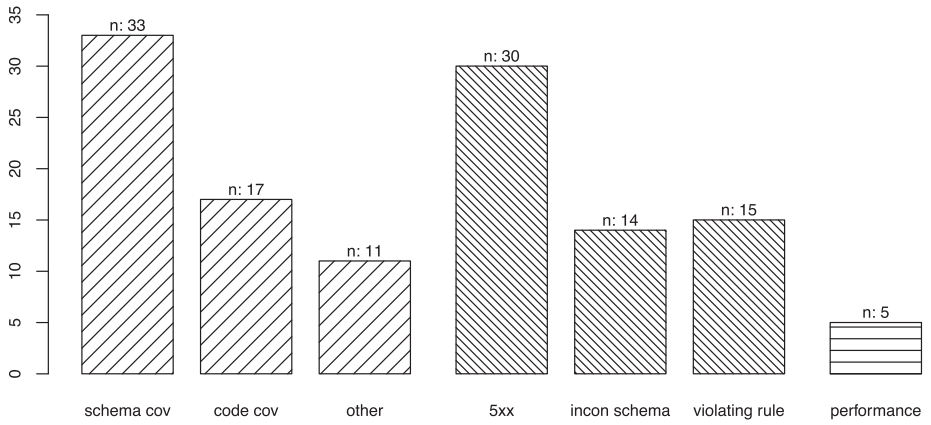

Fig. 6. Statistics of metrics employed in automated testing of REST API.

and failure responses received by requests to the path. Ed-douibi et al. [89] reported *endpoint coverage*, and an endpoint is considered as covered only if all of its operations are covered.

— *operations* are exposed to make requests (i.e., HTTP verb with path) for performing actions on the services. As discussed, a test is regarded as a sequence of the operations. To evaluate REST API testing approaches, Banias et al. reported a number of operations tested in Reference [63].

— *input parameter* is the information required to set when making a request. The parameter could be different with various types and constraints (e.g., *required*, *minimum*). Then, the input parameter-based metrics are defined to assess if various values for the parameters have been examined during testing (e.g., each Boolean parameter should had been evaluated with both values `true` and `false`). The generation could also be guided by the metrics; for example, Banias et al. [63] defined various configurations to generate tests with consideration of the required property of the parameters.

— *response* defines a list of possible responses to return per operation. Metrics relating to responses are used to examine whether various responses have been obtained (e.g., for an enumeration element in a returned body payload, coverage metrics would check if every single item in the enumeration has been returned at least once). Response body property coverage is reported to assess API schema-based REST API testing approach in Reference [63].

In addition, Martin-Lopez et al. [129] proposed 10 coverage metrics based on the schema that have been applied for assessing REST API testing approaches in Reference [63]. The 10 coverage metrics enable assessments of generated tests in fuzzing REST APIs with different inputs and outputs, such as *parameter coverage, content-type coverage,* and *response body properties coverage*. The metrics were also enabled in fuzzers, such as HsuanFuzz [155] and RESTest [131]. Restats [84] is a test coverage tool for assessing given tests based on the coverage metrics. In an empirical study [83], such schema-related coverage metrics were employed to compare black-box fuzzers for REST APIs.

(2) code coverage.

Code coverage [49, 52, 54–57, 59, 115, 118, 126, 145, 152, 155, 171, 172, 176, 177] is one typical white-box criterion for evaluating testing approaches. It was applied as well to evaluate REST API fuzzers for both black-box [59, 60] and white-box [53, 55, 171–173, 177]. Among different

code coverage criteria, line/statement coverage is one of the most common and supported ones in industry-strength coverage tools (e.g., JaCoCo for Java programs).

Code coverage was also used as an evaluation metric to assess the black-box REST API testing approaches [60, 115, 118, 155]. For instance, in Reference [60], Atlidakis et al. reported accumulated code coverage achieved along with the number of requests to be executed. To collect the code coverage, `TracePoint` hook was configured in Ruby classes to trace the execution in this study. However, black-box testing of REST API could be performed on remote public APIs that are closed-source software. The code coverage in this context is applied only for evaluation purposes.

Code coverage could also be employed as a criteria to automate fuzzing, which is considered as white-box testing. EvoMaster enables runtime code coverage (with heuristics to maximize it) automatically during fuzzing with code instrumentation. However, such automated collection of runtime code coverage is specific to the programming language, and EvoMaster has enabled it for JVM (e.g., Java and Kotlin) [52] and JavaScript [175]. In addition, Pythia [59] employs the metrics of code coverage to produce tests. But, to enable such code coverage collection, it needs a pre-manual step to extract code information (such as block location) of the SUT.

(3) specialized metrics.

Besides schema-based and code coverage, there also exist other specialized metrics, more specific (and possibly less general) to the proposed approaches [68, 77, 78, 98, 117, 118, 120, 130, 143, 156, 169].

For instance, in RESTler [60], a *grammar* is derived based on the API schema for driving following test sequence generation, e.g., selecting next HTTP request based on the derived producer-consumer dependencies among the endpoints. A specific metric employed in this approach is based on the *grammar*, i.e., *grammar coverage*. Martin-Lopez et al. [127, 128, 130] defined *inter-parameter dependencies* that represent constraints referring to two or more input parameters. Such dependencies are also used in RestCT to generate test data [166]. Alonso et al. [156] enabled test data generation with realistic inputs using natural language processing and knowledge extraction techniques. The performance of the data generation was evaluated from a percentage of *valid* API calls (i.e., 2xx status code) and *valid* inputs (i.e., *Syntactically valid* and *Semantically valid*). In Reference [100], Godefroid et al. introduced the *error type* metric, which is a pair of error code and error message. The metric was used to guide data fuzzing of REST API [100], i.e., maximizing *error type coverage*.

Chakrabarti and Rodriquez [77] defined **POST Class Graph (PCG)** to represent resources and connected relationships among them, then tests could be produced by maximizing coverage of the graph. UML state machine was applied in a model-based testing of REST APIs [143]. Metrics specific to model such as state coverage and transition coverage are used to guide test generation.

Lin et al. [115] constructed tree-based graph to analyze resource and resource dependency based on the API schema and responses. Then, test cases could be generated based on the graph with tree traversal algorithms.

To analyze security issues existing in REST API, Cheh and Chen [78] analyzed levels of *sensitivity* of data fields and API calls and also defined *exposure level* that calculates a degree of such data fields and API calls exposed to potential attacks.

6.1.2 Fault Detection.

(1) service error.

The status code 5xx has been applied to identify potential faults in REST API testing [49, 52–57, 59, 60, 63, 66, 85, 89, 104, 107, 113, 118, 126, 130, 131, 145, 152, 155, 156, 160, 166, 171, 172, 176, 177]. With HTTP, the status code 5xx indicates errors caused by the server, and the request could not be

processed until the server has been fixed. For instance, the 500 status code is generic to represent that an internal server error occurs when performing the given request. The status code 503 is more specific, stating that the service is unavailable, e.g., due to down for maintenance or the server is overloaded.

In most HTTP frameworks, when there is a crash in the business logic due to some faults (e.g., an unhandled null-pointer exception), the entire server does not crash. In these cases, the server would still reply to the incoming HTTP request, responding with a status code of 500. Therefore, 500 status codes in the responses can be used as an oracle to detect faults in RESTful APIs [104]. However, not all 500 status codes are related to software faults. For example, if the API is connecting to a database, and the database is currently down, then the server would not be able to complete the request. In such a case, returning a 500 status code would be correct, although no software fault in the API is involved.

(2) violation of schema.

The API schema (such as OpenAPI) defines the response syntax for each operation [53–57, 85, 89, 113, 126, 152, 160, 171, 172, 177], e.g., status code and response body. Actual responses should be always consistent with the syntax specified in the schema. Thus, any inconsistency between the actual response and the syntax could be regarded as faults in the REST API. For instance, Viglianisi et al. defined such oracles in RestTestGen [160] by using an OpenAPI library[14] to identify the mismatched responses. In QuickRest [107], Karlsson et al. formulated such consistency as properties.

(3) violation of defined rules.

Service errors (based on status code) and violations of schema (based on OpenAPI) are general oracles for fault finding in the context of REST API. Besides, there also exist some oracles to identify faults based on rules that characterize the REST APIs in terms of *security*, *behavior*, *properties,* and *regression* [61, 64, 66, 78, 92, 104, 107, 108, 113, 147, 156, 163, 169] (see Figure 5).

Security. As web services, security is critical for REST APIs. To enable test oracle relating to security, Atlidakis et al. [61] proposed a set of rules that formalize desirable security-related properties of the REST APIs. Any violation of the rules is identified as potential security-related bugs in the SUT. The rules are mainly defined based on assessing accessibility of resources, such as *use-after-free rule: If a resource has been deleted, then it must not be accessible anymore.*

Katt and Prasher [171] proposed a quantitative approach to measure the kinds of security related metrics (i.e., vulnerability, security requirement, and security assurance) for web services. To calculate the metrics, test cases are defined for validating whether the SUT meets security requirements and any kind of vulnerabilities exists. Masood and Java [133] identified various kinds of vulnerabilities that could exist in REST APIs, such as JSON Hijacking. Such vulnerabilities could be detected with both static analysis and dynamic analysis techniques. Barabanov et al. [64] proposed an approach specific to detection of **Insecure Direct Object Reference (IDOR)/Broken Object Level Authorization (BOLA)** vulnerabilities for REST APIs. The approach analyzes the OpenAPI specification by identifying its elements (such as parameters) relating to IDOR/BOLA vulnerabilities, then generates tests for verifying the API with such elements using defined security rules. Zha et al. [169] collected ***Common Sense Security Policies*** **(CSSP)**, such as *Access control*, *URL spoofing,* and private messages for Team Chat system and defined CSSP violation scenarios. Security and privacy risks can be identified if an API under the CSSP violation scenarios can still work, e.g., return a valid response. Barlas et al. [66] studied **regex-based denial of service**

[14] https://github.com/bjansen/swagger-schema-validator

(ReDoS) vulnerabilities led by handling of input sanitization in web services. The vulnerabilities are allowed to be identified by verifying consistency between client-side and server.

Behavior. Based on the provided API schema, Ed-douibi et al. [89] defined two rules to generate *nominal* test cases and *faulty* test cases. The *nominal* test cases take the inputs inferred based on examples or constraints in the schema, and the successful response is expected to return (e.g., assert that the status code is 2xx). Regarding the *faulty* test cases, it takes invalid inputs (e.g., missing required parameters, a string for a number parameter, a string violating its defined pattern), and the client error response is expected to return (e.g., assert that the status code is 4xx).

Liu et al. [117] constructed five constraints of REST guidelines with models. Then, such models could be used to verify design models of REST APIs. Any violation of the constraint models is considered as a potential defect in its architecture design.

Pinheiro et al. [143] defined UML state machines for modeling behaviors of REST APIs. The actual behavior (by executing the tests) should be consistent with the model (e.g., guard condition, invariant) as expected. Any inconsistency is recognized as potential faults of the SUT.

Properties. Most of the HTTP methods are idempotent, i.e., GET, PUT, DELETE, HEAD, OPTIONS, and TRACE. For such methods, the result of executing the method is independent of the number of repeated times, meaning that executing the method multiple times would not change the result compared to executing it once. Thus, assertions could be defined to check idempotency [157]. For example, executing multiple identical GET requests should result in the same response, and after a successful DELETE, the responses of all following identical DELETE requests should be the same. Connectedness is examined in Reference [77], which refers to accessibility among resources. For instance, assume that resource X owns resource Y; when performing a GET collection on Y referring to X, all available Y should appear in the response, otherwise the REST API is not "connected."

The REST API could apply **HATEOAS (Hypermedia as the Engine of Application State)**. Then, the response might contain hypermedia links for accessing itself or other resources. For such responses, Vassiliou-Gioles [157] defined assertions for validating availability of links in its response. In References [161, 162], Vu et al. also proposed a model-based approach that enables formalization of hypermedia behaviors of the REST API with ε-NFA, and the model could allow an identification of the faults by checking whether the SUT complies with it [163]. Fertig and Braun [92] also verified the REST APIs based on hypermedia constraints using model-based approaches.

*Metamorphic relation*s capture necessary properties that the SUT should hold with multiple executions. To enable metamorphic testing of REST APIs, there exist several works to identify such *metamorphic relation*s of the web services with abstract **Metamorphic Relation Output Patterns (MROPs)** [122, 147] (such as equivalence, disjoint) or specific properties of the API [120]. Then, faults could be detected by checking whether responses among the multiple requests conform to the identified relations.

Regression. Gazzola et al. [98] enable monitoring and tracing of the microservices to record their execution. Such recorded execution slices can be abstracted and considered as a metric for generating regression tests, i.e., verify whether the same response could be received with the same request in the further version of the SUT. Godefroid et al. [101] employed RESTler to produce tests and enabled detection of regression faults by comparing behaviors with the same tests among different versions of the REST APIs.

6.1.3 Performance Metrics. Performance-related metrics are also important for REST API testing [63, 66, 74, 92, 104], as the SUT provides services over the network. In Reference [63], Banias et al. measured the average response time of the requests generated by different strategies.

Table 6. Results of Techniques for Automated Testing of REST APIs

Black-/White-box	#	Techniques	#	Papers
Black Box	44	search	1	[116]
		model	8	[89, 92, 112, 117, 131, 143, 161, 163]
		property	8	[68, 77, 104, 107, 112, 120, 122, 147]
		others	28	[60, 61, 63, 64, 66, 74, 78, 85, 98, 100, 101, 109, 113, 115, 118, 121, 130, 134, 136, 140, 155–157, 160, 166, 167, 169, 178]
Both	1	search	1	[53]
Hybrid	1	search	1	[126]
White Box	15	search	12	[49, 52, 54–57, 145, 152, 171, 172, 176, 177]
		others	3	[59, 128, 154]
Total	61			

Note that we only consider the papers that are categorized as the type "New automated approach and its extension" (see Section 5.3).

Fertig and Braun [92] discussed potential solutions to enable performance testing with model-based approaches and existing techniques (such as Apache JMeter[15]). Schemathesis [104] defined the proprieties relating to performance metrics, i.e., identify slow response and request amplification with configured thresholds. Bucaille et al. [74] developed a testing framework that can assess and monitor performance-related properties, such as latency, by sending requests from various geographical locations using different cloud service providers.

RQ4: *Coverage criteria were the most applied metrics in REST API testing, measuring the degree to which aspects of the REST APIs are tested. Besides the traditional code coverage in white-box testing, schema-based coverage was mainly employed in black-box testing. Fault detection was the second widely applied metric that can be identified based on 5xx status codes, the given API schema, and defined rules. Performance metrics were rarely investigated in REST API testing.*

6.2 RQ5: What techniques are used for automatically testing RESTful APIs?

To answer this question, we conducted further analysis on 61 (out of 92) papers whose contributions are categorized as "new automated approach and its extension" (see Section 5.3).

6.2.1 Black-box and White-box Testing. Based on our survey, there exist two main types of testing techniques to automate testing of REST APIs, i.e., black-box and white-box. Regarding black-box testing, it enables to verify systems' behaviors with exposed endpoints, e.g., checking responses of an HTTP request. The verification and validation of REST APIs with black-box technique are mainly driven by exploiting the API specification and returned responses. Based on the results of this survey (see Table 6), 72% of the existing approaches are in the context of black-box testing.

Regarding black-box fuzzers, bBOXRT [113] aims at testing of REST APIs in terms of its robustness. EvoMaster applies search-based techniques by defining fitness function with black-box heuristics in a black-box mode [53], such as status code coverage and faulty status code (i.e., 500), to generate tests. There is also a set of property-based testing approaches that identifies properties of REST APIs as test oracle problems, such as QuickRest [107] and Schemathesis [104]. For instance, in terms of the API specification, QuickRest and Schemathesis both identify consistency with the specification as the properties. Schemathesis also explores semantic properties of REST APIs, such as `GET` should fail after an unsuccessful `POST` when they perform on the same resource. Moreover, dependencies of REST APIs are studied for REST API testing. For instance,

[15] https://jmeter.apache.org/

RESTler [60] infers and handles dependency among endpoints to generate effective tests based on API specification and runtime returned responses. RESTest generates tests by exploring inter-parameter dependencies of REST APIs. With RestTestGen [85, 160], *Operation Dependency Graph* is proposed to capture data dependencies among operations of a REST API. The graph is initially built based on its OpenAPI schema then further extended at runtime. Morest [118] formalized a property graph to construct relations of operations and object schema, and such a graph could be derived based on the API schema then dynamically updated based on responses at runtime. Within a specified time budget, test cases could be initially generated by traversing the graph, and any update of the graph would result in new tests. Furthermore, RestCT [166] is a combinational approach, integrating two phases that facilitate generating orders of operations then concertizing input parameters of operations. The orders are generated based on HTTP action semantics with greedy algorithm, e.g., for a specific resource, its `GET` operation should not appear before its `POST` and after its `DELETE`. The concrete values of input parameters could be produced in various ways, such as random, previous responses, specified examples, or inter-parameter dependencies. In the context of black-box testing, Cheh and Chen [78] enabled a semi-automatic approach to identify security issues in the API schema. Navas [136] proposed an approach to infer and validate the API schema, such as misnamed and duplicated elements in response schema.

Compared to black-box testing, white-box techniques could enable testing of internal behaviors of the REST APIs with additional metrics relating to source code, such as code coverage. For example, Pythia [59] employs code coverage heuristics to guide test generation. The code coverage could be collected by pre-configuring locations of basic blocks in its implementation with static analysis. EvoMaster is a fuzzer that has a white-box mode, and the white-box testing is enabled by code instrumentation for JVM [51, 52] and NodeJS programs [175] developed in the fuzzer. The code instrumentation allows to identify the code to cover and collect code coverage at runtime. With such information, EvoMaster defined white-box heuristics and applied search-based techniques to fuzzing REST APIs. With this survey, we found that all other white-box testing techniques are built on top of the EvoMaster platform (i.e., References [49, 52, 55, 145, 152, 154, 161, 171, 172, 176, 177]) with new algorithms and new techniques.

There also exist hybrid approaches that combine black-box and white-box [126]. For instance, Martin-Lopez et al. [126] proposed a solution by integrating two fuzzers (i.e., RESTest and EvoMaster). A motivation of having such a hybrid approach as described in the paper is that inputs of a REST API may be restricted with constraints, and the constraints are not specified in the API schema. Without the information of constraints, generating successful requests (i.e., receiving a response with 2xx status) is not trivial. However, white-box testing approach could tackle this problem by identifying the constraints with source code, such as EvoMaster [56, 57]. In Reference [126], RESTest is first employed to generate tests in the context of black-box testing, then the generated tests would be regarded as seeds that EvoMaster starts with. With the four selected REST APIs, the hybrid approach achieved the best results compared to isolated black-box and white-box solutions.

6.2.2 Search-based Testing. Search-based testing aims at solving software testing problems with metaheuristic search techniques, such as Genetic Algorithms and Swarm Algorithms. To enable the search techniques, it needs to reformulate the addressed testing problem as a search problem.

EvoMaster is a search-based fuzzer that reformulates test case generation as search problems supporting both black-box and white-box mode. The black-box mode is achieved with *Random* strategies with black-box heuristics, such as coverage of operations and status code. The white-box mode is enabled with novel techniques (e.g., code instrumentation [52], testability transformations [56, 57], and SQL handling [54, 55]), which allows to define white-box heuristics as part of

fitness function. For instance, code instrumentation enables identification of lines and branches to cover as testing targets and collecting such coverage at runtime, and testability transformations provide better guidelines to search for maximizing the testing targets. **Many independent objectives algorithm (MIO)** is the default algorithm in white-box mode of EvoMaster. MIO is an evolutionary algorithm developed specific to white-box system test generation, and the algorithm is inspired by (1+1)EA [88], which contains sampling and mutation operators. Its effectiveness to test REST APIs has been demonstrated in several papers [51, 52, 57]. To be specialized for Web API testing and REST API domain, MIO is further extended with adaptive hypermutation [171], smart sampling [52], and resource-based techniques [176, 177].

Genetic Algorithms (GAs) were also enabled for REST API testing. For instance, the Whole Test Suite [95] employed genetic algorithm to enable automated test suite generation. With the GA, the approach evolved the test generation with mutation and crossover operators, and the fitness function is defined with an overall single objective using white-box heuristics. Instead of single-objective optimization, the **Many-Objective Sorting Algorithm (MOSA)** extends NSGA-II [87] for handling test generations with respects to maximizing many testing targets as many-objective optimization. WTS and MOSA have been integrated into EvoMaster, which allows to apply them to tackle REST API testing [51]. With the EvoMaster platform, Stallenberg et al. [152] employed **Agglomerative Hierarchical Clustering (AHC)** to identify patterns of tests, then extended MOSA (named LT-MOSA) to generate tests with the patterns. Aside from test generation, Liu and Chen [116] employed genetic algorithm to optimize test data of REST APIs in the context of mutation testing.

Moreover, Swarm Algorithms were also investigated. Sahin [145] proposed a **Discrete Dynamic Artificial Bee Colony with Hyper-Scout (DABC-HS)** algorithm that was introduced to address shortcomings of a basic **Artificial Bee Colony (ABC)** for REST API testing. Furthermore, a Greedy Algorithm was applied in RestCT to produce operation sequences [166].

6.2.3 Property-based Testing. Property-based testing is an approach that validates and verifies the SUT based on identified properties that should always abide by the SUT. In the context of REST API testing, one type of properties could be identified based on schema, such as check if the responses during testing conform to the schema [104, 107]. In addition, as REST, it defines a set of guidelines on how to access resources. Then, states of resources in REST API are captured as properties of REST APIs. For instance, QuickRest [107] defined stateful properties of REST API with documented responses for producing a stateful sequence of operations. Seijas et al. [112] generated finite state machines to construct changes of states of resources, then employed Quviq QuickCheck to enable property-based testing of REST APIs. Chakrabarti and Rodriquez [77] examined the connectedness of resources in testing REST APIs.

There also exist some works for exploring semantic of REST APIs. For instance, in Schemathesis, Hatfield-Dodds and Dygalo [104] derived structural and semantic properties of REST APIs relating to constraints, security, and performances. Metamorphic testing was also enabled in REST API testing that identified metamorphic relations of the REST API based on their semantics [120, 122, 147].

6.2.4 Model-based Testing. Model-based testing is to employ models to perform testing. The models are typically constructed manually before the testing for depicting the SUT, such as states, behaviors. For instance, Vu et al. [161–163] proposed a model-based testing approach that uses an ε-NFA to construct hypermedia behavior of the REST API. Fertig and Braun [92] developed a **Domain Specific Language (DSL)** for constructing models of REST APIs, then defined a set of test templates for test generations with such models.

In Reference [117], Liu et al. proposed a model-based approach for verifying REST service architecture using **colored Petri Nets (CPN)**. Five REST feature constraints are defined with CPN models. Such constraint models allow a verification of architecture models using simulation. Pinheiro et al. [143] constructed behaviors of REST API with UML state machines.

Moreover, the models could also be used to construct the tests. Ed-douibi et al. [89] defined a metamodel for formalizing test suites of REST APIs. In the approach, the authors proposed a set of rules (e.g., infer parameter values based on examples, generate faulty test cases with invalid inputs) to generate the test suite model based on the OpenAPI. Then, the generated test suite model is further converted to execute code (e.g., JUnit) for performing the requests against the SUT.

6.2.5 Others. Graphs were constructed in several works for capturing the structures and relationships of REST APIs. Then, for example, **Breadth-First Search (BFS)** can be employed for generating tests based on such graph [60].

Gazzola et al. [98] proposed an approach called ExVivoMicroTest that facilitates regression test generation by recording service interactions at runtime. Godefroid et al. [101] proposed an approach for detecting faults due to changes made among different versions of RESTful web services. Such faults could be identified by comparing behavior of different versions with the same inputs generated by a fuzzer RESTler.

Takeda et al. [154] developed a test case extraction approach that aims at identifying the impacts of migrating APIs to microservices from monolithic architecture. The impacts are derived by analyzing source code in the context of white-box testing.

RQ5: *Both white-box and black-box techniques were investigated for tackling testing of REST API, and most of existing approaches (i.e., 72%) are black-box. With black-box testing, property-based and model-based testing were widely employed, which identify characteristics of the REST APIs (such as properties and behavior) to facilitate automated testing. White-box testing of REST APIs was mainly addressed by search-based techniques.*

6.3 RQ6: What kind of testing has been automated for RESTful APIs?

To find out what kinds of testing have been conducted by the papers and what are their frequencies, we extracted relevant data and analyzed them. The results of our findings are shown in Table 7. Not all the papers necessarily conduct only one testing type, and they might cover more than one of them (e.g., unit testing, integration testing, and acceptance testing are covered by Reference [69]). By studying the papers of this survey, we discovered 8 testing types in the context of REST API testing. More details about these types are discussed as follows:

— **System Testing**
System testing is a type of testing that verifies a software product's integration and completion. A system test's objective is to gauge how well the system requirements are met from beginning to end. Most of the papers (i.e., 72 out of 92) we found are either focusing solely on this type (e.g., Reference [134]) or are conducting it along with other types. As the REST is a guideline to build the web services and current techniques (e.g., OpenAPI) enable necessary information to perform system testing (e.g., make the request to REST API), it is expected that the system testing is the most-addressed problem in REST API testing.

— **Security Testing**
The fundamental objective of security testing is to determine the system's risks and assess any potential vulnerabilities so threats can be confronted and the system can continue to operate without being compromised. As an example, the study by Cheh and Chen uses the standardized OpenAPI specification as an input and suggests a semi-automatic method to

deduce different significant details regarding the security flaws in that API definition [78]. Security testing of REST APIs is of great importance. As it is mentioned in Section 1, many large enterprises rely on REST APIs. However, there have reportedly been a number of incidents involving web API security in recent years. The top three were denial of service attacks (19%), bot/scraping (20%), and vulnerabilities (54%) followed by authentication problems (46%). These flaws continue to exist until a hacker finds and takes advantage of them, which can lead to data loss, account abuse, or service interruption [47].

— **Integration Testing**
This stage examines whether collections of components function as expected by the technical system design or specification. We found some papers focusing on this kind of testing, such as the study by Vu et al. [161] that is aimed at automation of integration testing with the focus on hypermedia testing.

— **Unit Testing**
Unit testing is the process of testing a single units, small specialized section of code written by a developer. MockRest [149] is one tool focusing on unit testing by proposing a mock framework to help developers get a consistent response while the real REST API is down.

— **Regression Testing**
Regression testing is the process of ensuring that altered software continues to function as intended [91]. For example, to automatically find breaking changes across API versions, differential regression testing for REST APIs is presented in Reference [101]. In this study, 2 papers tackle specific regression testing problems for REST API. Moreover, test cases produced by a fuzzer could also be used for regression testing.

— **Robustness Testing**
Robustness testing aims to identify the extent to which a particular system or component can continue to operate properly in the presence of erroneous inputs or demanding environmental circumstances [20]. Fuzzers that aim at finding faults in the APIs might send invalid inputs on purpose to check if the API correctly returns an error message. One work focusing on robustness testing is Reference [113], which performed this type of test over REST services based on the constraint information expressed in their OpenAPI specification.

— **Architecture Design Testing**
The only paper that we found conducting this kind of testing is the study by Liu et al. [117]. This paper is aimed at enhancing system design's quality by verifying whether an API conforms to the REST architecture constraints described in Section 2.1.

— **Acceptance Testing**
Acceptance testing is a formal testing procedure used to ascertain whether a system satisfies its acceptance criteria and to give the client the option of accepting the system or not. The study by Besso et al. is the only paper that covers this type of testing by conducting it against web service choreographies [69].

As it is shown in Table 7, the frequency of papers that focus on system testing was much higher compared to those papers that necessarily require accessing to software components (e.g., integration test) or source code (e.g., unit test). As mentioned in Section 2.1, REST is a high-level design guideline, so it is understandable that most of the papers (i.e., 72) were focused on system testing. System testing is essential to ensure that the complete software system functions properly.

RQ6: *Existing studies in REST API testing covered 8 different testing types. Most of the studies (i.e., 72 out of 92) referred to system testing of REST API.*

Table 7. Different Testing Types and Their Frequencies among the Papers

Testing Type	#	Papers
System	72	[49, 52–60, 62, 63, 66, 68, 70, 74, 76, 77, 80, 83–85, 89, 92, 100, 104, 106, 107, 109, 110, 112, 113, 115, 116, 118, 120–131, 134, 136, 140, 143, 145, 147, 152, 154–157, 160, 162, 163, 165–168, 171–173, 175–178]
Security	8	[61, 64, 72, 78, 105, 108, 133, 169]
Integration	8	[69, 150, 151, 158, 159, 161, 163, 164]
Unit	5	[69, 149–151, 154]
Robustness	3	[85, 113, 160]
Regression	2	[98, 101]
Architecture	1	[117]
Acceptance	1	[69]

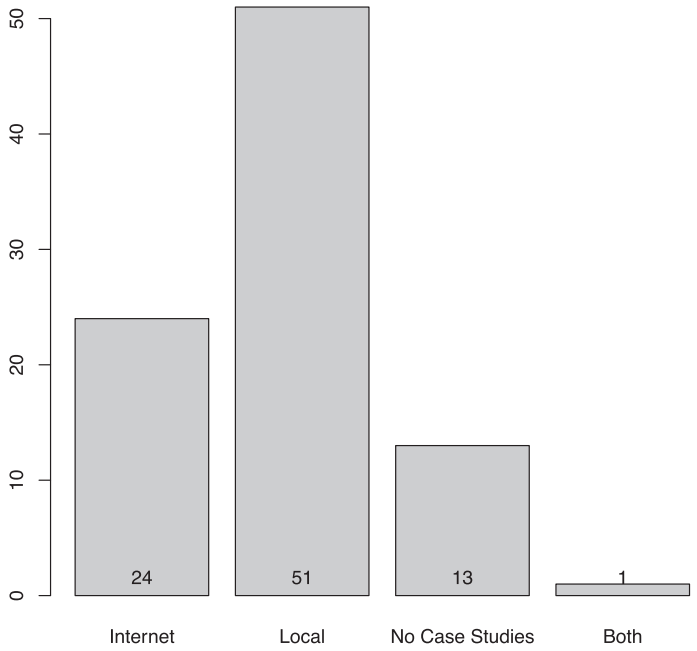

Fig. 7. Location of case studies used for empirical evaluations and their frequencies.

6.4 RQ7: What kind of artifacts are used for conducting empirical evaluations?

In this section, we investigated what kind of artifacts researchers have used as case studies to empirically evaluate the effectiveness of their proposed approaches. Sound empirical evidence is needed to demonstrate the usefulness of a novel technique. The larger and more variegated a case study is, the more likely it will be that results would generalize to other systems.

The results of our findings, including different categories of case studies and their frequency among the papers, are displayed in Figure 7. When dealing with experiments on testing Web APIs, there are two main types of artifacts: (1) APIs run on a local machine; and (2) existing APIs available on the internet (e.g., there are several thousands of REST APIs that are either free or commercial [2, 28, 29]).

For the former type (i.e., APIs run on a local machine), those are typically open-source projects, hosted on open-source repositories such as GitHub. Local open-source project is the most common

type of case studies used by 30% of the papers. For instance, case studies from *EMB* repository,[16] which includes a set of open-source APIs are utilized by 17 papers as case study [52–57, 123, 129, 145, 152, 168, 171–173, 175–177]. There are cases in which local closed-source APIs are used for this kind of experiments, but it is a less common occurrence (i.e., 7%), as it typically requires academia-industry collaborations on joined research projects [96]. There are also papers that take advantage of both open-source and closed-source APIs by running them locally. This group consists of 4% of the papers. In addition, several cases are witnessed in which some artificial example APIs are built by researchers to conduct empirical studies. These artificial artifacts are utilized by 16% of the papers.

The latter type, which includes 28% of the case studies, are typically industrial APIs providing paid services on the internet or free services from different government agencies. Some might be open-source, but those are a small minority. For this kind of APIs, researchers usually investigate only black-box techniques, since white-box techniques require access to the source code. For example, RESTTESTGEN [85, 160] used 87 RESTful APIs available on the internet for its empirical studies. However, one major drawback of using APIs on the internet is that experiments might not be *repeatable*, as APIs on the internet might change or disappear without any previous notice. Another case is Reference [136], which uses case studies hosted on the internet along with artifacts developed by the researchers.

Furthermore, a considerable number of papers did not perform any empirical evaluations: 14% of papers, such as the 4 papers, which their main contribution is of type "Proposal or idea," did not conduct any empirical evaluations. Additionally, there are papers that are not of type "Proposal or idea," but did not make use of any case studies for empirical evaluations, such as the study by Pinheiro et al. [143].

RQ7: *The most common groups of case studies are local open-source projects, used by 30% of the papers, and APIs on the Internet, used by 28% of the papers. Additionally, we found that 14% of the papers do not use any case studies.*

7 TOOLS FOR TESTING RESTFUL APIS

7.1 RQ8: Which research tools are open-source?

Typically, a scientific article does not have the space to fully describe all the low-level details of a newly presented technique. Releasing the implementation of a new research prototype as open-source offers multiple advantages. Not only does it address this limitation, but it also enables replicated studies, as re-implementing everything based solely on an article description can be a major engineering task. Furthermore, open-source tools, if maintained and properly engineered, can be used by practitioners to reduce the gap between academic research and industrial practice.

Studies that have their tool released as open-source, or use existing open-source tools for comparisons, are listed in Table 8, along with the programming language(s) the tools are implemented with. Note that, in some cases, studies extend on existing open-source tools, but no information is provided on whether the extension is available as open-source. Also, it might happen that a tool is released as open-source only *after* a scientific article is published. We have included these cases as well, but due to the lack of precise information (e.g., URL links might not be explicitly stated in the articles), we might have missed some.

Overall, it can be observed that 16 tools have either been compared or introduced by 44 papers. EVOMASTER is the tool that appears in the highest number of papers, with 19 papers either

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

Table 8. Information of Open-source Research Tools Have Been Developed for REST API Testing

	Tool	Website	#	Papers	Programming Language(s)
1	EvoMaster	[9]	19	[49, 52–58, 110, 123, 126, 145, 152, 171–173, 175–177]	Java, Kotlin, C#, TypeScript, JavaScript
2	RESTest	[35]	8	[83, 110, 125, 126, 130, 131, 156, 173]	Java
3	Restler	[36]	8	[60, 61, 66, 83, 100, 101, 110, 173]	Python, F#
4	RestTestGen	[37]	5	[83, 85, 110, 160, 173]	Java
5	bBOXRT	[5]	4	[83, 110, 113, 173]	Java
6	Schemathesis	[38]	3	[104, 110, 173]	Python
7	RestCT	[34]	2	[166, 173]	Python
8	Jsongen	[23]	2	[62, 68]	Erlang
9	RESTApiTester	[31]	1	[63]	Java
10	DeepLearningBasedTool	[8]	1	[134]	Python
11	Api Tester	[3]	1	[89]	Java
12	Restats	[33]	1	[84]	Python
13	Gadolinium	[15]	1	[74]	TypeScript, JavaScript
14	Hsuan-Fuzz	[19]	1	[155]	Go
15	Instance Identification	[21]	1	[158]	Go
16	ExVivoMicroTest	[11]	1	[98]	Python

presenting it, proposing a new technique integrated into it, or using it for comparison with other tools. The second most common tools are RESTest and RESTler, used in 8 papers.

It can be seen that Java and Python are the most common programming languages, each used by 6 open-source tools.

RQ8: *16 research tools have been released as open-source based on research outcomes of REST API testing.* EvoMaster *is the most used tool, which has been studied, extended, and compared with other tools, and* RESTest *and* RESTler *are the second most common ones. Java and Python are the most applied programming languages for developing these research tools.*

7.2 RQ9: Which non-research tools are used/compared?

REST APIs are widely used in industry, and their testing is a concrete issue that needs to be addressed, as it has practical value. Therefore, besides the work from the research community, it is not surprising to see effort from engineers and practitioners in industry to address this problem. Our goal here is not to survey all the non-research work done on the topic, but rather to analyze what academics found important and relevant enough to cite and use in their studies. Note that, with the term "non-research," we loosely mean all the software tools and libraries developed by practitioners in industry, without any published scientific article (as far as we know) describing them.

The aim of this research question is to find out what are the non-research tools and libraries that have been used in our surveyed studies. Those should be either specific to test REST APIs (e.g., API fuzzers) or can be used for testing REST APIs (e.g., code coverage tools). The tools we found are listed in Table 9. We found a larger number of non-research tools being used or compared in the papers, but not all of them were relevant to software testing or REST APIs domains. For example, *WSFuzzer*[17] is a tool used by one of the papers [121], but it is aiming at SOAP APIs, as the paper covers this type of APIs as well. With this survey, we found two kinds of non-research tools have been used in REST API testing, i.e., *library*, and *toolkit*.

There exist two libraries that have been used in testing of REST APIs. Being utilized by 12 papers, *RestAssured* [32] is the most used non-research tool, which facilitates doing HTTP calls

[17] https://sourceforge.net/projects/wsfuzzer/

Table 9. Information of Non-research Tools that Have Been Applied in REST API Testing

	Tool	Website	Description	#	Papers
1	RestAssured	[32]	DSL for writing tests in Java	12	[49, 52, 54–57, 72, 126, 130, 131, 171, 175]
2	Swagger Schema Validator	[43]	Validates JSON objects against Swagger 2	2	[85, 160]
3	Postman	[27]	Client tool for testing APIs manually	2	[126, 131]
4	Burp Suite	[6]	Commercial security testing fuzzer	2	[80, 84]
5	SoapUI	[39]	Web-service testing application	2	[74, 136]
6	APIFuzzer	[1]	Fuzzer tool based on OpenAPI	2	[104, 110]
7	Fuzz-lightyear	[12]	Stateful fuzzing framework based on OpenAPI	2	[64, 104]
8	TnT-Fuzzer	[45]	Open-source robustness testing tool	2	[80, 104]
9	SpotBugs	[40]	Static analysis tool for Java code	1	[113]
10	Dredd	[24]	Validates responses based on status codes, headers, and body payloads	1	[110]
11	Tcases	[44]	Black-box model-based testing based on OpenAPI	1	[110]
12	Autorize	[4]	Authorization enforcement detection tool (Extension for Burp Suite)	1	[64]
13	Go-fuzz	[16]	Randomized testing for Go	1	[155]
14	Got-Swag	[17]	Monkey testing for APIs based on Swagger	1	[104]
15	Cats	[7]	Fuzzer and negative testing tool based on OpenAPI	1	[104]
16	Swagger-conform	[41]	Tests if API conforms to Swagger schema	1	[104]
17	Fuzzy-swagger	[14]	Fuzzer based on Swagger	1	[104]
18	Swagger Fuzzer	[42]	Fuzzer based on Swagger	1	[104]
19	Fuzzapi	[13]	Open-source general-purpose HTTP fuzzer	1	[80]
20	ZAP	[46]	Penetration Testing Tool for Web Apps	1	[121]

against REST API and validating responses. *RestAssured* is mainly applied in tests generated by fuzzers, such as EvoMaster [52] and RESTest [126]. *Swagger Schema Validator* [43] is another library, which validates JSON objects against Swagger 2, which is utilized by response validation oracle to assess the automated test case generation [85].

There is a variety of toolkits for testing of REST APIs. For instance, *Postman* [27] is a platform for building and testing APIs, such as test execution and result validation. For REST API testing, tests can be specified as *Postman* format, and such tests can be created manually or automatically by fuzzers. For instance, EvoMaster supports *Postman* tests as seed for automated test case generation [126]. *Fuzz-lightyear* [12] is a framework for stateful fuzzing. It maintains the state between requests, allowing us to put together a request sequence, design it to simulate a malicious attack vector, and use it to notify of unexpected success. *Go-fuzz* [16] is also a framework to enable fuzzing in applications written in Go language. *Burp Suite* [6] is a commercial fuzzer for security testing that is utilized by 2 papers. *TnT-Fuzzer* [45] is an open-source tool for testing robustness that is taken advantage of for evaluation by 2 papers [80, 104]. *SoapUI* [39] is an application to perform automated end-to-end tests on a variety of web-services, including REST APIs to test their performance and security. *Tcases* [44] is another tool that performs black-box model-based testing based on OpenAPI specifications. *SpotBugs* [40] is a tool that enables static analysis of codes written in Java. *Dredd* [24] is an API testing tool that conducts sample-value-based testing technique and validates responses based on status codes, headers, and body payloads. *Autorize* [4] is another tool

Table 10. Features that Have Been Supported by Open-source Research Prototypes

Item	#	Tool(s)
Input		
OpenAPI	12	RestTestGen, RESTAPITester, EvoMaster, bBOXRT, RESTest, Restler, API Tester, Restats, Gadolinium, RestCT, Hsuan-Fuzz, Schemathesis
I/O Traces	2	Restats, ExVivoMicroTest
API Call	1	DeepLearningBasedTool
Path dependencies	1	Hsuan-Fuzz
JSON schema	1	Jsongen
Output		
JUnit test case	4	RestTestGen, EvoMaster, RESTest, API Tester
Unspecified test case	4	RESTAPITester, bBOXRT, Restler, Hsuan-Fuzz
Python test case	2	Schemathesis, ExVivoMicroTest
xUnit test case	1	EvoMaster
Jest test case	1	EvoMaster
Validity of API call	1	DeepLearningBasedTool
Test coverage metrics	1	Restats
Test coverage level	1	Restats
OpenAPI	1	Gadolinium
Visualized data (charts)	1	Gadolinium
Test results (statistics.csv and Swagger)	1	RestCT
QuickCheck generator	1	Jsongen
Authentication Support		
	8	RestTestGen, EvoMaster, RESTest, Restler, API Tester, Hsuan-Fuzz, Instance Identification, Jsongen
Database Reset Support		
	1	EvoMaster
Automated Code Instrumentation		
	1	EvoMaster

focusing on security that detects authorization enforcement within a REST API. *ZAP* [46] is a web application scanner tool focusing on security testing by performing penetration testing. The rest of the tools are fuzzers that depend on OpenAPI/Swagger schema.

RQ9: *20 non-research tools were identified from the selected 92 papers for REST API testing, in which RestAssured was the most used/compared by 12 papers.*

7.3 RQ10: Which features are supported by the research prototypes?

To get better understanding of the current state-of-the-art, we investigated what features are supported by the research prototypes. This can also help practitioners to evaluate these prototypes. To answer this research question, we collected features supported by the 16 open-source tools that we identified (see Section 7.1). Based on the collected features, we then divided them into five main categories, as shown in Table 10. The identification of features is based on what is reported in the published papers of these tools and their online documentation (if any is available). At times, the needed information is either unclear or missing. For example, a tool could state that it generates "test cases," but no information seems to be provided on the language and format of these output test cases in the documentation.

OpenAPI is supported by most of the tools (i.e., 12 out of 16) as a supported format of REST API schema. The tools take the OpenAPI specification as an input to identify endpoints of the REST

API. It has also been taken advantage of alongside other information such as HTTP logs in Restats and visualized data (i.e., charts) in Gadolinium [74].

Regarding what the released prototypes output, test cases are the most common one and, among them, *JUnit* is the most employed one. The capability to test REST APIs that need the client to be authenticated/authorized is a challenge that is handled by 8 of the tools. Another supported feature is clearing the database after each test run, which resets the database state. This feature is of great importance, as each test case has to be independent from each other. This feature is only supported by EvoMaster. EvoMaster is also the only tool that supports *Automated Code Instrumentation* as it conducts white-box testing and needs to insert probes into the source code of the SUT to collect code coverage during test case generation.

The features we detected are not only limited to Table 10. There are also a few other usability features in the different tools. For example, RESTest enables users to specify only subsets of operations to test. In other words, it allows to exclude some endpoints from being tested.

To be able to be used by practitioners, extremely important features for a tool are its *usability* and the quality of its *documentation*. It does not really matter if a tool can achieve high code coverage and fault detection if it is too cumbersome for anyone other than its authors to use. However, these metrics are hard to measure in an objective manner and would likely require controlled empirical studies with human subjects (ideally, practitioners in industry). In our survey, we have not found any such empirical study. The closest we found is studies on the comparisons of tool performance (e.g., References [83, 110, 173]), where authors commented on how difficult or easy it was to use and compare other tools from different researchers. There are different levels of usability and documentation quality among these tools that have been reported, where Schemathesis [104] clearly stands out among the most user-friendly and industry-strength tools.

RQ10: *There is a variety of features supported by the released prototypes.We listed them based on five categories. OpenAPI schema as inputs and JUnit test cases as outputs have been the most common ones.*

8 RESEARCH CHALLENGES

In this section, we discuss challenges presented in existing studies of REST API testing. These could be addressed challenges that have been posed by the research questions in the papers (RQ11) or issues that are still open for future work (RQ12).

8.1 RQ11: Which research challenges are addressed?

We studied the papers to infer the challenges they have addressed. Then, we wrote them down, grouped them, and searched for the most common ones that appear in at least more than one article. To determine the main challenges for each paper, we opted to consider only those challenges mentioned in the abstract section. The extracted main challenges were reviewed by two of us authors, where the third author discussed and resolved any discrepancies or disagreements, if there were any.

Most articles deal with presenting a novel technique (usually a fuzzer) aiming at fault finding. Here, however, we rather discuss work that focuses on specific identified challenges of testing REST APIs and not their testing in general. This can provide a better understanding of different specific aspects of testing REST APIs.

In Table 11, we have included the research challenges that have been addressed by at least one paper. *Handling resource dependency* was the most repeated research challenge among the analyzed articles, focused on in 8 articles. This challenge refers to dependencies among resources as they typically exist in the SUTs, and *dependency identification* is used to detect such dependencies based

Table 11. Common Research Challenges Addressed by the Papers

Challenge	#	Papers
Handling resource dependencies	8	[60, 77, 115, 118, 152, 164, 176, 177]
Inferring inter-parameter dependencies	6	[80, 125, 127, 128, 130, 166]
Oracle problem	5	[61, 64, 120, 122, 147]
Handling database	3	[54, 55, 172]
White-box heuristics	3	[56, 57, 175]
Mocking	2	[70, 149]
Defining coverage criteria	2	[84, 129]
Instance identification	2	[158, 159]

on *REST API Schema*, *Accessed SQL Tables,* and *Fitness Feedback* [177]. For example, Stallenberg et al. [152] have formed a model that captures, replicates, and preserves dependency patterns of API calls in new test cases, as breaking them could impede the effectiveness of the test-case generation process.

Inferring inter-parameter dependencies was the second most frequently addressed challenge (examined in 6 articles). It refers to the restrictions that web services frequently impose on how two or more input parameters can be combined to create valid calls to the service. It is frequent that the use of one parameter necessitates or impedes the use of another parameter or set of parameters. For example, Reference [127] has introduced a domain-specific language for the formal specification of dependencies, called **Inter-parameter Dependency Language (IDL)**, and a tool suite for the automated analysis of IDL.

Another common challenge is *Oracle problem*. Fuzzers can identify faults based on 500 HTTP status code and mismatches of the responses with the given API schema. Research has been carried out to define further automated oracles to be able to detect more faults. This could be based on security rules (e.g., References [61, 64]) or metamorphic relations. When a test execution's expected outcome is complex or unclear, metamorphic testing offers a solution that solves the oracle problem [79, 146]. Instead of examining the results of a single program execution, metamorphic testing examines whether many instances of the program being tested satisfy particular requirements known as metamorphic relations. For example, take into account the following metamorphic relation in Spotify: *Regardless of the size of the pagination, two searches for albums with the same query should return the same number of total results* [147].

The rest of challenges shown in Table 11 are less common, as they were addressed by two to three articles each. *Handling Databases* is a challenge that was addressed, as it has proven that taking database's state into account when generating tests will result in higher code coverage and finding new faults [54, 55].

White-box heuristics aim at improving code coverage results. For example, a common issue in SBST is the *flag problem* [65], where the branch distance is not able to provide any gradient. To solve this problem, one method is to use so-called Testability Transformations to change the SUT's source code in a way that enhances the fitness function [103]. These techniques can be used also to derive specific information for REST APIs, for example, detecting the use of query parameters not specified in the OpenAPI schema. The papers that have addressed this issue such as References [56, 57] transform the code of SUT to improve the fitness function during the search.

Mocking is another addressed challenge by two of the papers [70, 149], which is aimed at providing reliable response for the services that might not be accessible or down temporarily. *Instance Identification* is a problem in the context of micro-service testing that software testers face, as they might not know which concrete instance of service is being invoked. Vassiliou-Gioles

Table 12. Common Open Challenges

Challenge	#	Papers
Tool support	30	[49, 52, 63, 64, 66, 74, 80, 83, 89, 104, 105, 110, 112, 113, 118, 121, 122, 128–131, 133, 136, 143, 155–157, 160, 165, 175]
Having more REST case-studies	14	[60, 61, 72, 107, 117, 120, 126, 129, 134, 160, 171, 172, 176, 177]
Security testing	10	[60, 66, 85, 92, 98, 105, 108, 112, 155, 160]
Resource handling	4	[110, 172, 176, 177]
Classifying test results	4	[53, 83, 113, 155]
Handling external services	4	[49, 53, 80, 173]
White-box heuristics	4	[49, 56, 57, 126]
Database handling	4	[49, 52, 54, 55]
Automated oracles	4	[62, 68, 161, 162]
Performance testing	3	[92, 121, 150]
Non-functional testing	2	[130, 131]
Load testing	1	[63]
Metamorphic testing	1	[131]
Underspecified schemas	1	[173]

has suggested adding a micro-service **instance identification (IID)** header field to HTTP requests and responses to make them more testable [158, 159].

How to measure the effectiveness of black-box test generators is a challenge addressed in References [84, 129], where new black-box criteria besides counting detected faults have been defined (i.e., *Defining coverage criteria*). This is particularly important when testing remote APIs for which code coverage metrics cannot be used.

RQ11: *The most common addressed challenges include handling resource and inter-parameter dependencies, as well as defining new automated oracles.*

8.2 RQ12: Which open research challenges are identified?

Most of the papers we studied have mentioned some future objectives or challenges to be addressed. Among the selected papers, there were 73 papers that have explicitly mentioned at least one open challenge for future work. This is usually stated in the *Conclusion* section of these articles or in specific *Future Work* section. We collected all of these objectives and identified the most common challenges among them, as shown in Table 12. Note that, in most cases, a typical challenge is to design better techniques to obtain better code coverage and fault finding results. Here, we rather focus on more specific challenges related to testing RESTful APIs.

Based on Table 12, the most common objective left as future work is *Tool support*, which is mentioned in 30 papers. Research in this domain is based on tool prototypes. Due to the complexity of handling RESTful APIs, these tools require major engineering effort. An early prototype can provide the base to experiments with some research ideas, which can already be of use and be publishable. Work is then left to improve these tools to be able to be applicable on more case studies and to be more user-friendly for practitioners. For example, in Reference [89] the authors proposed an approach that supports OpenAPI v2 and plan to enable its support for OpenAPI v3 as well. Another example, supporting authentication is needed to enable testing endpoints that need the client to be authenticated, which was specified as needed future work in Reference [160]. In these cases, not all techniques presented in the literature support authentication or the full specs of the OpenAPI standard. Still, even with partial support, it is possible to provide and evaluate novel techniques.

Being mentioned in 14 papers, *Having more REST case-studies* to apply the new approach to a higher number of REST APIs for empirical evaluation was the second most frequent open challenge. This is a general issue related to *Threats to External Validity*, which applies to most empirical studies in the software engineering research literature. However, one peculiarity here is on the kind of systems used for experimentation. On the one hand, APIs on the internet pose few issues, including difficulty in replicating studies (APIs can change at any time) and inability to collect code coverage metrics. On the other hand, running APIs on local machines for experimentation has non-trivial setup costs, for example, on how to set up databases and authentication information. It can take a significant amount of time to find and set up a large number of REST APIs for experimentation. Furthermore, experiments on system test generation for Web APIs are time-consuming, as each test case evaluation requires to execute HTTP calls over a network. Given a fixed amount of time to run experiment (e.g., a machine with experiments left running for a week), this reduces the number of APIs that can be used for experimentation.

Many of the articles discussed in this article present a new approach for generating test cases that can find faults based on functional properties (e.g., API should not return a 500 HTTP status code or a response not matching the constraints of the schema). A common line of future work mentioned in these articles is to extend such work to consider other types of testing, in particular *Security testing* (10 papers), as well as others such as *Performance testing* and *Load testing* (and other unspecified *Non-functional testing*).

The other types of open challenges are mentioned less often, in up to 4 articles. For example, there are specific properties of REST APIs (e.g., the idempotency of HTTP verbs) that could be used as *Automated oracles* to be able to find more faults. Related, it will also be important to properly analyze the test results (i.e., *Classifying test results*) to be able to automatically check if the obtained responses represent actual faults and classify their importance/criticality.

Regarding *Handling external services*, it is possible for a RESTful API to rely on communications with other RESTful APIs. Whether the testing mode is either black-box or white-box, dealing with external services makes testing these APIs very difficult. In the black-box mode, there would be no control of the external services. Interactions with these external services would be based on both their implementation and current status. As a result, the chance of being flaky increases. Handling external services is a challenge in white-box mode as well. Even if a developer had complete control over all of those services, it might be difficult to set up and run numerous separate web services (each of which can utilize its own database) for usage by the SUT during testing.

Another issue is when dealing with *Database handling*. Databases are very common in RESTful APIs, and the interactions of the APIs with these databases do impact the level of code coverage and fault-finding the fuzzers can reach. Some basic techniques have been presented to handle SQL databases, but more needs to be done, especially to handle other kinds of databases, like NoSQL databases such as MongoDB.

When the content of databases cannot be analyzed (e.g., in black-box testing), it is important to detect relations between operations, such as the need to create a resource with a `POST` request first before being able to test its `GET` endpoint. Several articles have addressed this issue (e.g., recall Section 8.1). Still, a major issue regarding *Resource handling* is how to deal with schemas that do not follow the guidelines of REST (e.g., resources not structured hierarchically), as it is much harder there to infer resource relations among the different endpoints.

Based on our review, only one tool supports automated white-box testing of RESTful APIs (i.e., EvoMaster). As the achieved code coverage still needs to be improved, several code-level issues have been identified and categorized, which will need to be addressed to achieve higher code coverage, e.g., with new *White-box heuristics*. One possible venue where white-box heuristics can

be useful is to deal with *Underspecified schemas*, i.e., when there are constraints on the data and operations of the API, but those are not specified in the schema.

RQ12: *We have identified a variety of open challenges discussed in the analyzed articles. There is still a lot of research work that is needed to be carried out in this domain.*

9 DISCUSSION

This survey provides a clear picture of the current state-of-the-art in the testing of REST APIs. There has been a sharp growth in the number of studies carried out in the domain of REST API testing, as explained in Section 5 (**RQ1**). Scientific articles on this topic have been published in many different venues, where top venues such as TOSEM, ICSE, and ICST are among the most common (**RQ2**). This shows a clear growing maturity in this line of research and its interest in the software engineering research community.

Different topics have been investigated, and different types of studies have been carried out (**RQ3**). However, two important topics seem missing from our survey: *manual testing* and *experience reports*. In our survey, there is no article that studied the use of manual testing for REST APIs. We did not have any specific reason for not including such kind of study. None of our search queries defined in Table 1 are either filtering them out, nor taking in only papers related to automated testing. We simply did not find any such kind of study in the literature. We have some hypotheses to explain this phenomenon. A possible major issue is the lack of case studies. To study manual testing for REST APIs, there is a need of several APIs with existing, manually written test cases. Although REST APIs are widely used in industry, they are less common in open-source repositories, which can explain why collecting a proper case study for such kind of analysis has not happened so far. This is also supported by our finding in **RQ7** (types of artifacts used in empirical evaluations) and in **RQ12** (open challenges). Where, for example, Mostafa and Wang used 5,000 open-source projects to study current practices in the use of mock objects in manual tests [135], doing something like this for REST APIs does not yet seem possible.

Another important missing topic in our survey is experience reports. Many fuzzers have been developed and evaluated in the lab, but no study in industry with industrial partners applying such tools and integrating them in their development processes has been reported yet. This seems to point out that we are not there yet in the technology transfer from academic research to industrial practice. However, the increase in research output in the past few years (**RQ1**) might imply that we are not so far from such important goal. These two missing topics clearly point to an important missing gap in the research literature that will need future work to cover.

In Section 6, we have surveyed how techniques have been evaluated (**RQ4**), what techniques have been used to test REST APIs (**RQ5**), the type of testing addressed (**RQ6**), and which case studies were used in the empirical studies (**RQ7**). This can provide useful information for researchers, by showing what kind of research has been done so far, and help bootstrap new research by showing how new techniques can be evaluated and compared.

Besides for researchers, there can also be benefits for practitioners. In Section 7, we have surveyed the current existing tools for testing RESTful APIs. In particular, we looked at open-source projects coming from the research community (**RQ8**), which non-academic tools have been used in research studies (**RQ9**), and which features are supported by research tools (**RQ10**). Prototypes developed for publishing research papers are not necessarily mature enough to be used by practitioners to test their APIs. As we are the authors of EvoMaster [52], we are too biased to comments on its usability and impact for practitioners. However, it is clear that there are existing fuzzers that are already of use for practitioners (e.g., Schemathesis [104]). Several studies on real-world APIs

have shown the feasibility and potential usefulness of these tools (**RQ7**). Although more still needs to be done, practitioners today can already directly benefit from this line of research.

To drive and steer new research effort on this important engineering domain, in Section 8, we have analyzed which research challenges have been addressed by the surveyed work (**RQ11**) and which open challenges have been identified (**RQ12**). Those latter provide a clear road-map for future research, as we found many gaps in this field. For example, it is highlighted that the primary future work objective, mentioned in 30 papers, is the development of tool support for handling RESTful APIs. These tools, currently in the prototype stage, require substantial engineering effort. Early prototypes offer a foundation for experimentation and potential publication. However, further improvements are needed to enhance the tools' applicability across different case studies and make them more user-friendly for practitioners. Furthermore, while based on Section 6.4 only 14% of the studies did not perform any experimental evaluations, which is much lower than the number reported by Reference [73] in 2013, according to Table 12, one of the most common open challenges is *Having more REST case-studies*. This indicates that, while a lot has improved in terms of experimental evaluations, researchers in this domain believe that there is still need for more solid empirical results to prove the value of their innovative techniques. Existing fuzzers have already been used to find thousands of real faults in existing APIs. However, one clear topic that is often mentioned for future work is *security testing*. The research community has just started to look into it, and there is much more there that can be done with potential impact for industrial practice. A very recent (published half-year *after* our data collection) good example is Reference [82], on the study of mass assignment vulnerabilities (at the time of this writing, the proceedings of ICSE 2023 where such work was accepted/presented are not published yet).

10 THREATS TO VALIDITY

As for any survey, there is the validity threat that some important and relevant articles have been missed from our analysis. We used the most popular search databases to find all relevant articles and followed both a forward and backward snowballing procedure to find any other missing relevant articles. However, as this procedure was done manually, human mistakes are possible. To reduce such a risk, three authors were involved in the procedures to reach a final agreement by all, e.g., only papers that all of us three authors agreed on to exclude were excluded. However, there is always the possibility that other researchers might have come to some different selections.

Extracting data from the articles required manual effort and expertise, and it might be prone to human mistakes. To reduce such validity threat, each selected paper was checked by at least two of us authors. We are the authors of 16 articles out of the 92 in our survey. Although we are confident that we were able to properly analyze our own work, there is always the risk that we might have misunderstood the text of some articles written by other researchers.

11 CONCLUSION

In this survey, we have collected and analyzed 92 articles on the topic of testing RESTful APIs. For this analysis, we can see that there has been an exponential increase in interest on this topic in the research community, starting from 2017 (Section 5). Various techniques have been evaluated, including both black-box and white-box techniques (Section 6). Furthermore, besides scientific articles, several prototypes have been released as open-source projects, with empirical investigation carried out on many real-word APIs, finding real faults in them. This shows potential usefulness of this line of research for practitioners in industry (Section 7). Different scientific challenges have been addressed, while others still need to be solved (Section 8).

RESTful APIs are widely used in industry. Research work on this topic has strong potential to have significant impact on industrial practice.

This survey provides a detailed snapshot of the current state-of-the-art in the research literature on testing RESTful APIs. This is a growing field, where this survey can provide a useful starting point to drive new research directions on this important topic.

A replication package with all the raw data from this survey can be found on Zenodo [102].

REFERENCES

[1] APIFuzzer – HTTP API Testing Framework. Retrieved from https://github.com/KissPeter/APIFuzzer. Accessed August 28, 2023.
[2] APIs.guru. Retrieved from https://apis.guru/. Accessed August 28, 2023.
[3] ApiTester. Retrieved from https://github.com/opendata-for-all/api-tester. Accessed August 28, 2023.
[4] Autorize: Automatic authorization enforcement detection extension for Burp Suite. Retrieved from https://github.com/portswigger/autorize. Accessed August 28, 2023.
[5] bBOXRT. Retrieved from https://git.dei.uc.pt/cnl/bBOXRT. Accessed August 28, 2023.
[6] Burp Suite. Retrieved from https://portswigger.net/burp. Accessed August 28, 2023.
[7] Cats: REST API Fuzzer and negative testing tool for OpenAPI endpoints. Retrieved from https://github.com/Endava/cats. Accessed August 28, 2023.
[8] Deep Learning-Based Prediction of Test Input Validity for RESTful APIs Tool. Retrieved from https://anonymous.4open.science/r/8954c607-8d6c-4348-a23c-d57c920cdc22. Accessed August 28, 2023.
[9] EvoMaster. Retrieved from https://github.com/EMResearch/EvoMaster. Accessed August 28, 2023.
[10] EvoMaster Benchmark (EMB). Retrieved from https://github.com/EMResearch/EMB. Accessed August 28, 2023.
[11] ExVivoMicroTest. Retrieved from https://gitlab.com/learnERC/exvivomicrotest. Accessed August 28, 2023.
[12] Fuzz-lightyear: Stateful fuzzing framework. Retrieved from https://github.com/Yelp/fuzz-lightyear. Accessed August 28, 2023.
[13] Fuzzapi. Retrieved from https://github.com/Fuzzapi/fuzzapi. Accessed August 28, 2023.
[14] Fuzzy-swagger. Retrieved from https://github.com/Lothiraldan/swagger-fuzzer. Accessed August 28, 2023.
[15] Gadolinium. Retrieved from https://github.com/opendata-for-all/gadolinium. Accessed August 28, 2023.
[16] Go-fuzz: Randomized testing for Go. Retrieved from https://github.com/dvyukov/go-fuzz. Accessed August 28, 2023.
[17] Got-Swag. Retrieved from https://github.com/freenet-public/got-swag. Accessed August 28, 2023.
[18] GraphQL Foundation. Retrieved from https://graphql.org/foundation/. Accessed August 28, 2023.
[19] Hsuan-Fuzz. Retrieved from https://github.com/iasthc/hsuan-fuzz. Accessed August 28, 2023.
[20] IEEE Standard Glossary of Software Engineering Terminology, IEEE Standard 610.12-1990, Dec. 1990. [Online; Available:] Retrieved from http://standards.ieee.org. Accessed August 28, 2023.
[21] Instance Identification. Retrieved from https://github.com/theovassiliou/instanceidentification. Accessed August 28, 2023.
[22] JSON Schema Specification Wright Draft 00. Retrieved from https://datatracker.ietf.org/doc/html/draft-wright-json-schema-00. Accessed August 28, 2023.
[23] JSONGen. Retrieved from https://github.com/fredlund/jsongen. Accessed August 28, 2023.
[24] Language-agnostic HTTP API Testing Tool. Retrieved from https://github.com/apiaryio/dredd. Accessed August 28, 2023.
[25] Open API Specification. Retrieved from https://swagger.io/specification/. Accessed August 28, 2023.
[26] OpenAPI/Swagger. Retrieved from https://swagger.io/. Accessed August 28, 2023.
[27] Postman: API platform for building and using APIs. Retrieved from https://www.getpostman.com/. Accessed August 28, 2023.
[28] Programmable Web. Retrieved from https://www.programmableweb.com/. Accessed August 28, 2023.
[29] RapidAPI. Retrieved from https://rapidapi.com/. Accessed August 28, 2023.
[30] REST API Tutorial. Retrieved from https://restfulapi.net/. Accessed August 28, 2023.
[31] RESTApiTester. Retrieved from https://github.com/RobertGyalai/RESTApiTester. Accessed August 28, 2023.
[32] RestAssured. Retrieved from https://github.com/rest-assured/rest-assured. Accessed August 28, 2023.
[33] Restats. Retrieved from https://github.com/SeUniVr/restats. Accessed August 28, 2023.
[34] RestCT. Retrieved from https://github.com/GIST-NJU/RestCT. Accessed August 28, 2023.
[35] RESTest. Retrieved from https://github.com/isa-group/RESTest. Accessed August 28, 2023.
[36] RESTler. Retrieved from https://github.com/microsoft/restler-fuzzer. Accessed August 28, 2023.
[37] RestTestGen. Retrieved from https://github.com/SeUniVr/RestTestGen. Accessed August 28, 2023.
[38] Schemathesis: Property-based testing for API schemas. Retrieved from https://schemathesis.readthedocs.io/. Accessed August 28, 2023.

[39] SoapUI: Accelerating API Quality Through Testing. Retrieved from https://www.soapui.org/. Accessed August 28, 2023.
[40] SpotBugs: Find bugs in Java program. Retrieved from https://github.com/spotbugs/spotbugs. Accessed August 28, 2023.
[41] Swagger-conform. Retrieved from https://github.com/olipratt/swagger-conformance. Accessed August 28, 2023.
[42] Swagger Fuzzer. Retrieved from https://github.com/Lothiraldan/swagger-fuzzer. Accessed August 28, 2023.
[43] Swagger Schema Validator. Retrieved from https://github.com/bjansen/swagger-schema-validator. Accessed August 28, 2023.
[44] Tcases for OpenAPI: From REST-ful to Test-ful. Retrieved from https://github.com/Cornutum/tcases/tree/master/tcases-openapi. Accessed August 28, 2023.
[45] TnT-Fuzzer. Retrieved from https://github.com/Teebytes/TnT-Fuzzer. Accessed August 28, 2023.
[46] ZAP: OWASP Zed Attack Proxy. Retrieved from https://www.zaproxy.org/. Accessed August 28, 2023.
[47] Salt.security. 2021. API Security Trends. Retrieved from https://salt.security/api-security-trends. Accessed August 28, 2023.
[48] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Trans. Softw. Eng.* 36, 6 (2010), 742–762.
[49] Andrea Arcuri. 2017. RESTful API automated test case generation. In *IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*. IEEE, 9–20.
[50] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: We need usable automated test generation. *Empir. Softw. Eng.* 23, 4 (2018), 1959–1981.
[51] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Inf. Softw. Technol.* 104 (2018), 195–206.
[52] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Trans. Softw. Eng. Methodol.* 28, 1 (2019), 3.
[53] Andrea Arcuri. 2020. Automated black-and white-box testing of RESTful APIs with EvoMaster. *IEEE Softw.* 38, 3 (2020), 72–78.
[54] Andrea Arcuri and Juan P. Galeotti. 2019. SQL data generation to enhance search-based system testing. In *Genetic and Evolutionary Computation Conference (GECCO'19)*. Association for Computing Machinery, New York, NY, 1390–1398. DOI: https://doi.org/10.1145/3321707.3321732
[55] Andrea Arcuri and Juan P. Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Trans. Softw. Eng. Methodol.* 29, 4 (2020), 1–31.
[56] Andrea Arcuri and Juan P. Galeotti. 2020. Testability transformations for existing APIs. In *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST'20)*. IEEE, 153–163.
[57] Andrea Arcuri and Juan P. Galeotti. 2021. Enhancing search-based testing with testability transformations for existing APIs. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (2021), 1–34.
[58] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A search-based system test generation tool. *J. Open Source Softw.* 6, 57 (2021), 2153.
[59] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: Grammar-based fuzzing of rest APIs with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498* (2020).
[60] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API fuzzing. In *ACM/IEEE International Conference on Software Engineering (ICSE'19)*. 748–758.
[61] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking security properties of cloud service rest APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'20)*. IEEE, 387–397.
[62] Ignacio Ballesteros, Luis Eduardo Bueso de Barrio, Lars-Ake Fredlund, and Julio Marino. 2018. Tool demonstration: Testing JSON web services using JSONGEN. *biblioteca.sistedes.es* (2018). [Online; Available:] Retrieved from https://biblioteca.sistedes.es/submissions/descargas/2018/PROLE/2018-PROLE-025.pdf. Accessed August 28, 2023.
[63] Ovidiu Banias, Diana Florea, Robert Gyalai, and Daniel-Ioan Curiac. 2021. Automated specification-based testing of REST APIs. *Sensors* 21, 16 (2021), 5375.
[64] Alexander Barabanov, Denis Dergunov, Denis Makrushin, and Aleksey Teplov. 2022. Automatic detection of access control vulnerabilities via API specification processing. *arXiv preprint arXiv:2201.10833* (2022).
[65] A. Baresel and H. Sthamer. 2003. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation Conference (GECCO'03)*. 2442–2454.
[66] Efe Barlas, Xin Du, and James C. Davis. 2022. Exploiting input sanitization for regex denial of service. In *IEEE/ACM 44th International Conference on Software Engineering (ICSE'22)*. 883–895. DOI: https://doi.org/10.1145/3510003.3510047

[67] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Trans. Softw. Eng.* 41, 5 (2015), 507–525.
[68] Clara Benac Earle, Lars-Åke Fredlund, Ángel Herranz, and Julio Mariño. 2014. Jsongen: A QuickCheck based library for testing JSON web services. In *13th ACM SIGPLAN Workshop on Erlang*. 33–41.
[69] Felipe M. Besson, Pedro M. B. Leal, Fabio Kon, Alfredo Goldman, and Dejan Milojicic. 2011. Towards automated testing of web service choreographies. In *6th International Workshop on Automation of Software Test*. 109–110.
[70] Thilini Bhagya, Jens Dietrich, and Hans Guesgen. 2019. Generating mock skeletons for lightweight web-service testing. In *26th Asia-Pacific Software Engineering Conference (APSEC'19)*. IEEE, 181–188.
[71] Ilona Bluemke and Agnieszka Malanowska. 2021. Software testing effort estimation and related problems: A systematic literature review. *ACM Comput. Surv.* 54, 3 (2021), 1–38.
[72] Gloria Bondel, Josef Kamysek, Markus Kraft, and Florian Matthes. 2021. Design and implementation of a test tool for PSD2 compliant interfaces. In *International Conference on Enterprise Information Systems (ICEIS'21)*. 249–256.
[73] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. 2013. Testing and verification in service-oriented architecture: A survey. *Softw. Test., Verific. Reliab.* 23, 4 (2013), 261–313.
[74] Steven Bucaille, Javier Luis Cánovas Izquierdo, Hamza Ed-Douibi, and Jordi Cabot. 2020. An OpenAPI-based testing framework to monitor non-functional properties of rest APIs. In *International Conference on Web Engineering*. Springer, 533–537.
[75] Gerardo Canfora and Massimiliano Di Penta. 2009. Service-oriented architectures testing: A survey. In *Software Engineering*. Springer, 78–105.
[76] Sujit Kumar Chakrabarti and Prashant Kumar. 2009. Test-the-rest: An approach to testing restful web-services. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns (COMPUTATIONWORLD'09)*. IEEE, 302–308.
[77] Sujit Kumar Chakrabarti and Reswin Rodriquez. 2010. Connectedness testing of restful web-services. In *3rd India Software Engineering Conference*. ACM, 143–152.
[78] Carmen Cheh and Binbin Chen. 2021. Analyzing OpenAPI specifications for security design issues. In *IEEE Secure Development Conference (SecDev'21)*. IEEE, 15–22.
[79] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: A new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).
[80] Yixiong Chen, Yang Yang, Zhanyao Lei, Mingyuan Xia, and Zhengwei Qi. 2021. Bootstrapping automated testing for RESTful web services. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 46–66.
[81] Emilia Cioroaica, Said Daoudagh, and Eda Marchetti. 2022. Predictive simulation for building trust within service-based ecosystems. In *IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops'22)*. IEEE, 34–37.
[82] Davide Corradini, Michele Pasqua, and Mariano Ceccato. 2023. Automated black-box testing of mass assignment vulnerabilities in RESTful APIs. *arXiv preprint arXiv:2301.01261* (2023).
[83] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Empirical comparison of black-box test case generation tools for RESTful APIs. In *IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM'21)*. IEEE, 226–236.
[84] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Restats: A test coverage tool for RESTful APIs. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'21)*. IEEE, 594–598.
[85] 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. *Softw. Test., Verific. Reliab* 32, 5 (2022), e1808.
[86] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. 2002. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Comput.* 6, 2 (2002), 86–93.
[87] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. A. M. T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolut. Comput.* 6, 2 (2002), 182–197.
[88] S. Droste, T. Jansen, and I. Wegener. 2002. On the analysis of the (1+1) evolutionary algorithm. *Theor. Comput. Sci.* 276 (2002), 51–81.
[89] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic generation of test cases for REST APIs: A specification-based approach. In *IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC'18)*. 181–190.
[90] Adeel Ehsan, Mohammed Ahmad M. E. Abuhaliqa, Cagatay Catal, and Deepti Mishra. 2022. RESTful API testing methodologies: Rationale, challenges, and solution directions. *Appl. Sci.* 12, 9 (2022), 4369.

[91] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical evaluations of regression test selection techniques: A systematic review. In *2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 22–31.
[92] Tobias Fertig and Peter Braun. 2015. Model-driven testing of RESTful APIs. In *24th International Conference on World Wide Web*. ACM, 1497–1502.
[93] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. 1999. Hypertext Transfer Protocol–HTTP/1.1. (1999). [Online; Available:] Retrieved from https://www.rfc-editor.org/rfc/rfc2616?data1=dwnsb4B&data2=abmurltv2b. Accessed August 28, 2023.
[94] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine.
[95] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Trans. Softw. Eng.* 39, 2 (2013), 276–291.
[96] Vahid Garousi, Dietmar Pfahl, João M. Fernandes, Michael Felderer, Mika V. Mäntylä, David Shepherd, Andrea Arcuri, Ahmet Coşkunçay, and Bedir Tekinerdogan. 2019. Characterizing industry-academia collaborations in software engineering: Evidence from 101 projects. *Empir. Softw. Eng.* 24, 4 (2019), 2540–2602.
[97] Vahid Garousi, Austen Rainer, Per Lauvås Jr, and Andrea Arcuri. 2020. Software-testing education: A systematic literature mapping. *J. Syst. Softw.* 165 (2020), 110570.
[98] Luca Gazzola, Maayan Goldstein, Leonardo Mariani, Marco Mobilio, Itai Segall, Alessandro Tundo, and Luca Ussi. 2022. ExVivoMicroTest: ExVivo testing of microservices. *J. Softw.: Evolut. Process* 35, 4 (2022), e2452.
[99] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.
[100] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. ACM, 725–736.
[101] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 312–323.
[102] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Replication Package for Testing RESTful APIs: A Survey (Version 4). DOI: https://doi.org/10.5281/zenodo.8018888
[103] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. 2004. Testability transformation. *IEEE Trans. Softw. Eng.* 30, 1 (2004), 3–16.
[104] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web API schemas. In *IEEE/ACM 44th International Conference on Software Engineering (ICSE'22)*. IEEE, 345–346.
[105] Muhammad Idris, Iwan Syarif, and Idris Winarno. 2021. Development of vulnerable web application based on OWASP API security risks. In *International Electronics Symposium (IES'21)*. IEEE, 190–194.
[106] Stefan Karlsson. 2019. Exploratory test agents for stateful software systems. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1164–1167.
[107] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. 2020. QuickREST: Property-based test generation of OpenAPI described RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'20)*. IEEE.
[108] Basel Katt and Nishu Prasher. 2018. Quantitative security assurance metrics: REST API case studies. In *12th European Conference on Software Architecture*. 1–7.
[109] Yaroslav Khortiuk, Galyna Kondratenko, Ievgen Sidenko, and Yuriy Kondratenko. 2020. Increasing reliability of programming interfaces based on fuzz testing. In *IEEE 11th International Conference on Dependable Systems, Services and Technologies (DESSERT'20)*. IEEE, 272–277.
[110] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated test generation for REST APIs: No time to rest yet. *arXiv preprint arXiv:2204.08348* (2022).
[111] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated test generation for REST APIs: No time to rest yet. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*. Association for Computing Machinery, New York, NY, 289–301. DOI: https://doi.org/10.1145/3533767.3534401
[112] Pablo Lamela Seijas, Huiqing Li, and Simon Thompson. 2013. Towards property-based testing of RESTful web services. In *12th ACM SIGPLAN Workshop on Erlang*. ACM, 77–78.
[113] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A black box tool for robustness testing of REST services. *IEEE Access* 9 (2021), 24738–24754.
[114] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A systematic review on software robustness assessment. *ACM Comput. Surv.* 54, 4 (2021), 1–65.
[115] Jiaxian Lin, Tianyu Li, Yang Chen, Guangsheng Wei, Jiadong Lin, Sen Zhang, and Hui Xu. 2022. foREST: A tree-based approach for fuzzing RESTful APIs. *arXiv preprint arXiv:2203.02906* (2022).
[116] Jing Liu and Wenjie Chen. 2017. Optimized test data generation for RESTful web service. In *24th Asia-Pacific Software Engineering Conference (APSEC'17)*. IEEE, 683–688.

[117] Jing Liu, Zhen-Tian Liu, and Yu-Qiang Zhao. 2018. CPN model based standard feature verification method for REST service architecture. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Springer, 688–707.
[118] 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. *arXiv preprint arXiv:2204.12148* (2022).
[119] 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 *ACM/IEEE International Conference on Software Engineering (ICSE'22)*.
[120] Gang Luo, Xi Zheng, Huai Liu, Rongbin Xu, Dinesh Nagumothu, Ranjith Janapareddi, Er Zhuang, and Xiao Liu. 2019. Verification of microservices using metamorphic testing. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 138–152.
[121] Riyadh Mahmood, Jay Pennington, Danny Tsang, Tan Tran, and Andrea Bogle. 2022. A framework for automated API fuzzing at enterprise scale. In *IEEE Conference on Software Testing, Verification and Validation (ICST'22)*. IEEE, 377–388.
[122] Vishnu Manikantan, Neeraj Menon, V. S. Vishnupriyan, Ansamma John, and N. K. Anantha Padmanabhan. 2022. Software tool to perform metamorphic testing on RESTful web APIs. In *Innovations in Computer Science and Engineering*. Springer, 355–362.
[123] 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 (2022), 1–43.
[124] Alberto Martin-Lopez. 2020. AI-driven web API testing. In *ACM/IEEE 42nd International Conference on Software Engineering*. 202–205.
[125] Alberto Martin-Lopez. 2021. ICSE: G: Automated management of inter-parameter dependencies in web APIs. In *ACM SRC Grand Finals*.
[126] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-box and white-box test case generation for RESTful APIs: Enemies or allies? In *IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE'21)*. IEEE, 231–241.
[127] Alberto Martin-Lopez, Sergio Segura, Carlos Muller, and Antonio Ruiz-Cortés. 2021. Specification and automated analysis of inter-parameter dependencies in web APIs. *IEEE Trans. Serv. Comput.* 15, 4 (2021).
[128] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A catalogue of inter-parameter dependencies in RESTful web APIs. In *International Conference on Service-oriented Computing*. Springer, 399–414.
[129] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. Test coverage criteria for RESTful web APIs. In *10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 15–21.
[130] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-box constraint-based testing of RESTful web APIs. In *International Conference on Service-oriented Computing*.
[131] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated black-box testing of RESTful web APIs. In *ACM International Symposium on Software Testing and Analysis (ISSTA'21)*. ACM, 682–685.
[132] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. 2022. Software engineering for AI-based systems: A survey. *ACM Trans. Softw. Eng. Methodol.* 31, 2 (2022), 1–59.
[133] Adnan Masood and Jim Java. 2015. Static analysis for web service security—Tools & techniques for a secure development life cycle. In *IEEE International Symposium on Technologies for Homeland Security (HST'15)*. IEEE, 1–6.
[134] A. Giuliano Mirabella, Alberto Martin-Lopez, Sergio Segura, Luis Valencia-Cabrera, and Antonio Ruiz-Cortés. 2021. Deep learning-based prediction of test input validity for RESTful APIs. In *IEEE/ACM 3rd International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest'21)*. IEEE, 9–16.
[135] Shaikh Mostafa and Xiaoyin Wang. 2014. An empirical study on the usage of mocking frameworks in software testing. In *14th International Conference on Quality Software*. IEEE, 127–132.
[136] Alvaro Navas Baltasar, Pedro Capelastegui de la Concha, Francisco Huertas Ferrer, Pablo Alonso Rodríguez, and Juan Carlos Dueñas Lopez. 2014. REST service testing based on inferred XML schemas. *Netw. Protoc. Algor.* 6, 2 (2014), 6–21.
[137] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. 2018. An analysis of public REST web service APIs. *IEEE Trans. Serv. Comput.* 14, 4 (2018).
[138] Sam Newman. 2015. *Building Microservices*. O'Reilly Media, Inc.
[139] Samuel Oloruntoba. 2021. SOLID: The First 5 Principles of Object Oriented Design. Retrieved from https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
[140] Mitchell Olsthoorn. 2022. More effective test case generation with multiple tribes of AI. In *44th International Conference on Software Engineering Companion*.

[141] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. 2022. Test case selection and prioritization using machine learning: A systematic literature review. *Empir. Softw. Eng.* 27, 2 (2022), 1–43.
[142] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (2021), 1–74.
[143] Pedro Victor Pontes Pinheiro, Andre Takeshi Endo, and Adenilso Simao. 2013. Model-based testing of RESTful web services using UML protocol state machines. In *Brazilian Workshop on Systematic and Automated Software Testing*.
[144] R. V. Rajesh. 2016. *Spring Microservices*. Packt Publishing Ltd.
[145] Omur Sahin and Bahriye Akay. 2021. A discrete dynamic artificial bee colony with hyper-scout for RESTful web service API test suite generation. *Appl. Soft Comput.* 104 (2021), 107246.
[146] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Trans. Softw. Eng.* 42, 9 (2016), 805–824.
[147] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2017. Metamorphic testing of RESTful web APIs. *IEEE Trans. Softw. Eng.* 44, 11 (2017).
[148] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sérgio Lopes de Souza. 2017. A systematic review on search based mutation testing. *Inf. Softw. Technol.* 81 (2017), 19–35.
[149] Anshu Soni, Virender Ranga, and Sandeep Jadhav. 2020. MockRest—A generic approach for automated mock framework for REST APIs generation. In *Inventive Communication and Computational Technologies*. Springer, 237–255.
[150] Stelios Sotiriadis, Andrus Lehmets, Euripides G. M. Petrakis, and Nik Bessis. 2017. Unit and integration testing of modular cloud services. In *IEEE 31st International Conference on Advanced Information Networking and Applications (AINA'17)*. IEEE, 1116–1123.
[151] Stelios Sotiriadis, Andrus Lehmets, Euripides G. M. Petrakis, and Nik Bessis. 2018. Testing cloud services using the TestCast tool. In *Information Technology—New Generations*. Springer, 819–824.
[152] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2021. Improving test case generation for REST APIs through hierarchical clustering. *arXiv preprint arXiv:2109.06655* (2021).
[153] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education.
[154] Tomohiro Takeda, Masakazu Takahashi, Tsuyoshi Yumoto, Satoshi Masuda, Tohru Matsuodani, and Kazuhiko Tsuda. 2019. Applying change impact analysis test to migration test case extraction based on IDAU and graph analysis techniques. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'19)*. IEEE, 131–139.
[155] Chung-Hsuan Tsai, Shi-Chun Tsai, and Shih-Kun Huang. 2021. REST API fuzzing by coverage level guided blackbox testing. In *IEEE 21st International Conference on Software Quality, Reliability and Security (QRS'21)*. IEEE, 291–300.
[156] Juan Carlos Alonso Valenzuela, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated generation of realistic test inputs for web APIs. *IEEE Trans. Softw. Eng.* 49, 1 (2022).
[157] Theofanis Vassiliou-Gioles. 2020. A simple, lightweight framework for testing RESTful services with TTCN-3. In *IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C'20)*. IEEE, 498–505.
[158] Theofanis Vassiliou-Gioles. 2021. Quality assurance of micro-services-when to trust your micro-service test results? In *IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C'21)*. IEEE, 01–06.
[159] Theofanis Vassiliou-Gioles. 2022. Solving the instance identification problem in micro-service testing. In *IFIP International Conference on Testing Software and Systems*. Springer, 189–195.
[160] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated black-box testing of RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'20)*. IEEE.
[161] Henry Vu, Tobias Fertig, and Peter Braun. 2018. Automation of integration testing of RESTful hypermedia systems: A model-driven approach. In *International Conference on Web Information Systems and Technologies (WEBIST'18)*. 404–411.
[162] Henry Vu, Tobias Fertig, and Peter Braun. 2018. Verification of hypermedia characteristic of restful finite-state machines. In *the Web Conference*. 1881–1886.
[163] Henry Vu, Tobias Fertig, and Peter Braun. 2019. Model-driven integration testing of hypermedia systems. *J. Web Eng.* 18, 4 (2019), 301–480.
[164] Hu Wenhui, Huang Yu, Liu Xueyang, and Xu Chen. 2017. Study on REST API test model supporting web service integration. In *IEEE 3rd International Conference on Big Data Security on Cloud (BIGDATASECURITY), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*. IEEE, 133–138.
[165] Behailu Getachew Wolde and Abiot Sinamo Boltana. 2021. REST API composition for effectively testing the cloud. *J. Appl. Res. Technol.* 19, 6 (2021), 676–693.
[166] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial testing of RESTful APIs. In *ACM/IEEE International Conference on Software Engineering (ICSE'22)*.

[167] Koji Yamamoto. 2021. Efficient penetration of API sequences to test stateful RESTful services. In *IEEE International Conference on Web Services (ICWS'21)*. IEEE, 734–740.
[168] Koji Yamamoto, Takao Nakagawa, Shogo Tokui, and Kazuki Munakata. 2020. Call sequence list distiller for practical stateful API testing. In *International Conference on Software Engineering and Knowledge Engineering (SEKE'20)*. 342–346.
[169] Mingming Zha. 2022. Hazard integrated: Understanding security risks in app extensions to team chat systems. In *Network and Distributed Systems Security (NDSS'22) Symposium*.
[170] Bin Zhang, Jiaxi Ye, Chao Feng, and Chaojing Tang. 2017. S2F: Discover hard-to-reach vulnerabilities by semi-symbolic fuzz testing. In *13th International Conference on Computational Intelligence and Security (CIS'17)*. IEEE, 548–552.
[171] Man Zhang and Andrea Arcuri. 2021. Adaptive hypermutation for search-based system test generation: A study on REST APIs with EvoMaster. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (2021).
[172] Man Zhang and Andrea Arcuri. 2021. Enhancing resource-based test case generation for RESTful APIs with SQL handling. In *International Symposium on Search-based Software Engineering*. Springer, 103–117.
[173] Man Zhang and Andrea Arcuri. 2022. Open problems in fuzzing RESTful APIs: A comparison of tools. *arXiv preprint arXiv:2205.05325* (2022).
[174] Man Zhang and Andrea Arcuri. 2023. Open problems in fuzzing RESTful APIs: A comparison of tools. *ACM Trans. Softw. Eng. Methodol.* (May 2023). DOI: https://doi.org/10.1145/3597205
[175] Man Zhang, Asma Belhadi, and Andrea Arcuri. 2022. JavaScript instrumentation for search-based software testing: A study with RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST'22)*. IEEE.
[176] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based test case generation for RESTful web services. In *Genetic and Evolutionary Computation Conference*. 1426–1434.
[177] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2021. Resource and dependency based test case generation for RESTful Web services. *Empir. Softw. Eng.* 26, 4 (2021), 1–61.
[178] Chunhui Zhao, Zhili Wang, and Xiao Zhang. 2020. Semantic-oriented automatic generation method of REST interface test cases. In *4th International Symposium on Computer Science and Intelligent Control*. 1–6.

Received 30 December 2022; revised 9 June 2023; accepted 24 July 2023