Online Testing of RESTful APIs: Promises and Challenges

3 authors:

Alberto Martin-Lopez
Universidad de Sevilla
13 PUBLICATIONS **105** CITATIONS
SEE PROFILE

Sergio Segura
Universidad de Sevilla
96 PUBLICATIONS **3,412** CITATIONS
SEE PROFILE

Antonio Ruiz-Cortés
Universidad de Sevilla
322 PUBLICATIONS **7,424** CITATIONS
SEE PROFILE

Online Testing of RESTful APIs: Promises and Challenges

Alberto Martin-Lopez
alberto.martin@us.es
SCORE Lab, I3US Institute,
Universidad de Sevilla
Seville, Spain

Sergio Segura
sergiosegura@us.es
SCORE Lab, I3US Institute,
Universidad de Sevilla
Seville, Spain

Antonio Ruiz-Cortés
aruiz@us.es
SCORE Lab, I3US Institute,
Universidad de Sevilla
Seville, Spain

ABSTRACT

Online testing of web APIs—testing APIs in production—is gaining traction in industry. Platforms such as RapidAPI and Sauce Labs provide online testing and monitoring services of web APIs 24/7, typically by re-executing manually designed test cases on the target APIs on a regular basis. In parallel, research on the automated generation of test cases for RESTful APIs has seen significant advances in recent years. However, despite their promising results in the lab, it is unclear whether research tools would scale to industrial-size settings and, more importantly, how they would perform in an online testing setup, increasingly common in practice. In this paper, we report the results of an empirical study on the use of automated test case generation methods for online testing of RESTful APIs. Specifically, we used the RESTest framework to automatically generate and execute test cases in 13 industrial APIs for 15 days non-stop, resulting in over one million test cases. To scale at this level, we had to transition from a monolithic tool approach to a multi-bot architecture with over 200 bots working cooperatively in tasks like test generation and reporting. As a result, we uncovered about 390K failures, which we conservatively triaged into 254 bugs, 65 of which have been acknowledged or fixed by developers to date. Among others, we identified confirmed faults in the APIs of Amadeus, Foursquare, Yelp, and YouTube, accessed by millions of applications worldwide. More importantly, our reports have guided developers on improving their APIs, including bug fixes and documentation updates in the APIs of Amadeus and YouTube. Our results show the potential of online testing of RESTful APIs as the next must-have feature in industry, but also some of the key challenges to overcome for its full adoption in practice.

CCS CONCEPTS

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

KEYWORDS

REST, web API, bot, black-box testing, online testing

ACM Reference Format:
Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2022. Online Testing of RESTful APIs: Promises and Challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore.* ACM, New York, NY, USA, 13 pages.

1 INTRODUCTION

Web APIs allow systems to interact with each other over the network, typically using web services [84]. Most modern web APIs comply with the REST architectural style [74], being referred to as RESTful web APIs. RESTful web APIs [95] provide access to data and services by means of create, read, update, and delete (CRUD) operations over resources (e.g., a video in the YouTube API [50] or a playlist in the Spotify API [44]). RESTful APIs are ubiquitous in the modern-day society: public institutions such as the American government [3] expose their existing assets as a set of RESTful APIs; software companies such as Microsoft [29] and Netflix [30] base many of their systems communications on their RESTful APIs; even non-software companies such as Marvel [28] provide APIs for developers to build applications on top of them. The importance and pervasiveness of web APIs is also reflected on the size of popular API repositories such as ProgrammableWeb [37] and RapidAPI [38], which currently index over 24K and 30K APIs, respectively.

Checking the correct functioning of web APIs is critical. A single bug in an API may affect tens or hundreds of other services leveraging it. In this scenario, test thoroughness and automation are of utmost importance. Industry standard tools and libraries for testing RESTful APIs such as Postman [36] and REST Assured [41] automate test execution, but test cases still need to be manually implemented. Industrial efforts are also shifting toward the model of online testing, where APIs are continuously tested while in production. This is the case of platforms such as Datadog [22], RapidAPI Testing [39], and Sauce Labs [43], which offer web API testing and monitoring as a service. Under this model, APIs are tested and monitored 24/7 with pre-defined API requests and output assertions following a black-box approach. Customers may choose among different pricing plans [40] determining the number of API calls per day/month, the integration with CI/CD platforms, and the type of notifications, among others. These platforms completely automate test case execution, but their generation still requires manual work, either for the creation or amplification [71] of test cases.

Research on the automated generation of test cases for RESTful APIs has seen significant advances in recent years. Most approaches follow a black-box strategy, where test cases are automatically derived from the specification of the API under test, typically in the OpenAPI Specification (OAS) format [34]. These approaches are capable of automatically generating (sequences of) HTTP requests, sending them to the API and asserting the validity of the responses. Test data generation strategies are varied [54, 56, 57, 70, 72, 78, 85, 86, 90], while test oracles are mostly limited to checking the conformance with the API specification and the absence of API crashes. Despite their promising results, research approaches have typically been evaluated in controlled environments with a few open source APIs [55, 68, 87], a limited number of test cases [70, 72, 90], or within the boundaries of a single organization [58, 78]. Thus, there is no evidence of their applicability and generalizability to industrial settings and, more importantly, of how they would fit into the online testing model increasingly found in practice. Hence, as a matter of fact, there is a clear gap between industrial solutions, mostly concerned with test case execution, and research approaches, focused on automated test case generation, but with evaluation results limited to lab settings.

In this paper, we report the results of an empirical study on the use of automated test case generation methods for online testing of RESTful APIs. Specifically, we automatically generated and executed test cases on 13 industrial APIs during the course of 15 days non-stop. We assessed different black-box testing strategies including fuzzing [77], adaptive random testing [82], and constraint-based testing [79], among others, resulting in over one million test cases. The evaluation was conducted using RESTest [91], an open source testing framework for RESTful web APIs. However, to scale at this level, we had to transition from a monolithic tool approach to a multi-bot architecture, resulting in over 200 bots—highly cohesive and autonomous programs—working cooperatively in tasks such as test case generation, test case execution, and test case reporting. We uncovered about 390K test failures, which we automatically clustered into 4,818 potential faults. After a systematic selection and manual inspection of 586 fault clusters, we conservatively identified 254 reproducible issues in all the APIs under test, 65 of which have been acknowledged or fixed by the API developers to date [92]. The bugs detected are varied, including API crashes caused both by valid [10] and invalid API inputs [9], unexpected client errors [8, 19] and security vulnerabilities [13], among many others. Our bug reports led to fixes in the documentation and the implementation of the APIs of YouTube and Amadeus, used by millions of users worldwide. The results provide helpful insights on the fault detection capability of online testing of RESTful APIs as well as its limitations, including its high computational cost and the debugging effort required. We also report the differences observed among different testing techniques, and how these techniques complement each other for finding more and more varied bugs. Overall, our work contributes to narrowing the gap between research and practice on online testing of RESTful APIs, showing the lights and shadows of current test case generation techniques at scale.

In summary, after discussing related work (Section 2) and presenting the empirical study performed (Section 3), this paper makes the following original research and engineering contributions:

(1) An assessment on the failure and fault detection capability of online testing of RESTful APIs (Sections 4.1 and 4.2).
(2) A comparison of test data and test case generation techniques for online testing of RESTful APIs (Section 4.3).
(3) A set of problems uncovered by online testing in 13 industrial APIs, including 254 reproducible bugs (Section 5).
(4) A list of challenges for the adoption of test case generation techniques for online testing of RESTful APIs (Section 6).
(5) A publicly available implementation of a multi-bot architecture for online testing of RESTful APIs, and a dataset of over 1M test cases and 390K test failures [92].

Finally, we address threats to validity in Section 7 and conclude the paper in Section 8.

2 RELATED WORK

2.1 Automated Testing of RESTful APIs

Research on the automated generation of test cases for RESTful APIs has thrived in recent years. Most approaches follow a black-box approach, where test cases are automatically derived from the API specification, typically in the OpenAPI Specification (OAS) format [34]. An OAS document represents a contract on how a RESTful API may be used, i.e., it describes the API in terms of the allowed inputs (HTTP requests) and outputs (HTTP responses), in a machine-readable format. Given an OAS document, current techniques automatically generate pseudo-random test cases—sequences of HTTP requests—and test oracles—assertions on the HTTP responses. Approaches mainly differ in the way in which they generate input data and test cases. Regarding test data generation strategies, these include using default and example values [72], fuzzing dictionaries [57], purely random inputs [56], perturbed data [86], custom data generators and data dictionaries [85, 91], data extracted from knowledge bases (e.g., DBpedia [64]) [54], and contextual data (i.e., extracted from previous API responses) [70, 78]. Test case generation strategies include random testing (parameters are assigned values randomly) [57, 70, 72, 85], adaptive random testing (aiming to diversify test cases as much as possible) [91], and constraint-based testing (parameters are assigned values such that inter-parameter dependencies [88] are satisfied) [90]. In terms of test oracles, these are mainly limited to checking the absence of API crashes (500 status codes) and the conformance with the specification. Other oracles include checking the status code [90], metamorphic relations [97] and security properties [58]. White-box techniques require access to the code of the API under test and are far less common than black-box techniques. Existing approaches employ search algorithms to maximize code coverage and fault detection [55, 96, 99].

Related approaches on the automated generation of test cases for RESTful APIs have been mostly evaluated in lab settings using a few open source APIs [55, 68, 87], a limited number of test cases (up to a few thousands) [70, 72, 90], during a limited amount of time (up to a few hours) [57, 70, 78] or within the boundaries of a single organization [58, 78]. In contrast, our work complements previous work by assessing the strengths and limitations of different state-of-the-art test case generation techniques in an online testing scenario. Specifically, we automatically generated and ran test cases on 13 industrial APIs with millions of users worldwide during 15 days non-stop. This provides a novel perspective on the potential of automated test case generation techniques for RESTful APIs in practice: we uncovered 254 *unique* bugs in 13 industrial APIs, 65 of which have been acknowledged or fixed by developers to date, and some of which have led to changes in the APIs of Amadeus and YouTube. More importantly, our work opens new promising research directions to address some of the challenges identified in areas such as debugging and testing as a service.

2.2 Online Testing

Online testing refers to the testing performed on production software [60, 63], as opposed to offline testing, which is done on a development environment, before the software is released. Online testing can be regarded both as a *passive* monitoring activity [60, 61] or as an *active* testing process [65, 80]. In the former, the system is observed under real-world usage aiming to detect inconsistencies or anomalies [59, 83]. In the latter, the system is continuously stimulated with new test cases, possibly derived from the feedback obtained from previous test results [65, 100]. In this paper, we adopt the second definition and refer to online testing as an active testing process consisting in generating and executing test cases (i.e., API requests) in the APIs in production.

Online testing platforms for web APIs are gaining popularity in industry. These platforms provide continuous testing and monitoring capabilities as a service, with different pricing plans determining features such as the test execution frequency, the automation degree and the number of users, among others. Datadog [22] is a monitoring platform for tracking an API's availability and response time by periodically executing a set of manually configured API calls. RapidAPI Testing [39] and Sauce Labs (formerly API Fortress) [43] allow to create functional tests composed of one or more HTTP requests with custom assertions in the HTTP responses. The data used in the requests can be static, contextual (i.e., obtained from a previous response) or random, based on a data category (e.g., "yellow" for the category "color"). However, test cases must be individually created (RapidAPI) or amplified based on a template (Sauce Labs). The same test cases are executed continuously at regular intervals. In the research arena, some papers address online testing of RESTful APIs (i.e., testing web APIs in production) [70, 72, 78, 86], but with a limited number of test cases or during a few hours, and typically using a single testing technique. Online testing has also been applied to service-oriented architectures (SOA) [65, 67, 80], including non-functional testing [62, 94], and other less related domains such as mobile applications [66], embedded systems [53] and software product lines [81], among others.

Compared to prior work, we report the first empirical study on the use of automated test case generation techniques for online testing of RESTful APIs in industrial-like settings. This resembles the model of testing as a service offered by popular industrial platforms. This allowed us to study the failure and fault detection capability of different black-box test case generation techniques over time (15 days), as well as some of the key challenges to overcome for their adoption in practice.

3 EMPIRICAL STUDY

In this section, we report the results of our study on online testing of RESTful APIs, including both automated generation and execution of test cases.

3.1 Research Questions

We aim to answer the following research questions:

Table 1: RESTful APIs under test. O = Operations, P = Parameters, R = Read, W = Write.

API	Category	Service under test	O	P	R	W
Amadeus [2]	Travel	Hotel search	1	27	X	
DHL [23]	Shipping	Service point search	1	9	X	
FDIC [24]	Banking	*all*	6	73	X	
Foursquare [26]	Social, mapping	Venues search	1	17	X	
LanguageTool [27]	Languages	Text proofread	1	11		X
Marvel [28]	Entertainment	Characters	6	72	X	
Ohsome [31]	Mapping	*all*	122	1,146	X	
OMDb [32]	Media	*all*	1	9	X	
RESTcountries [42]	Reference	*all*	21	42	X	
Spotify [44]	Media	Playlists	12	48	X	X
Stripe [45]	Financial	Products	5	47	X	X
Yelp [48]	Recommendations	Businesses search	1	14	X	
YouTube [49]	Media	Comments	10	148	X	X
YouTube [51]	Media	Search	1	31	X	
Total			189	1,694	13	4

RQ$_1$: *What is the failure detection capability of online testing of RESTful APIs?* We aim to investigate the number and types of failures detected over time.

RQ$_2$: *What is the fault detection capability of online testing of RESTful APIs?* Out of all the failures uncovered, we aim to find out the number and types of *unique* faults detected over time.

RQ$_3$: *What are the differences observed between different testing approaches?* We wish to compare different testing strategies in terms of API coverage and fault detection capability, aiming to understand how different approaches complement each other.

3.2 APIs Under Test

Table 1 depicts the RESTful APIs under test. For each API, we specify its name, the service tested (*"all"* if the whole API is tested), the number of operations and parameters tested (columns "O" and "P", respectively), and whether it provides read and write operations (columns "R" and "W", respectively). We test the *Search* and *Comments* services of YouTube as different APIs, as done by previous authors [90, 93], since they are subject to different testing strategies (e.g., the former is a read-only API while the latter provides write functionality). Our benchmark comprises large and complex real-world APIs tested by previous authors [54, 87, 90, 93], which includes both commercial APIs used by millions of users worldwide (e.g., Spotify and Yelp), but also industrial-size open source APIs (e.g., Ohsome[1] and LanguageTool). We add another API to our benchmark, FDIC (Federal Deposit Insurance Corporation), as a good representative of complex APIs developed by public institutions (i.e., the American government). The OAS specifications of the APIs (required to test them) were obtained from the APIs' websites or from public repositories such as APIs.guru [4]. Exceptionally, Foursquare, RESTCountries and Yelp did not have public OAS documents, so we reused them from our previous work [87, 90], ensuring they accurately reflected the API documentations.

Overall, we selected 13 APIs from varied application domains, which comprehend 189 operations and 1,694 input parameters. Most of the APIs tested provide only read operations. This is a common

[1]Ohsome developers showed interest in our preliminary results and deployed an API instance identical to the production one where we could run our experiments without restrictions.

Table 2: Taxonomy of test bots and total bots used per type, according to the inclusion criteria for our experiments.

Bot dimension	Bot ID	Bot name	Description	Inclusion criterion	Total
Write-safety	R	*Read-only*	Tests only read operations (i.e., HTTP GET requests)	The API contains read operations	55
	RW	*Read/write*	Tests both read and write operations (e.g., POST and DELETE requests)	The API contains write operations	20
Test data generation technique	FD	*Fuzzing dictionaries*	Uses type-aware fuzzing dictionaries and malformed strings for all request parameters	All APIs	14
	CDG	*Customized data generators*	Parameters are configured with customized data generators (e.g., a generator of dates in ‘`yyyy/mm/dd`’ format)	All APIs	32
	DP	*Data perturbation*	Makes a minor change to the data used in an API request to make it invalid	All APIs	14
	SD	*Semantically-related data*	Uses concepts related to the semantics of the parameters, extracted from knowledge bases like DBpedia [64]	The API contains parameters from which to extract semantically-related concepts	5
	CD	*Contextual data*	Uses values observed in previous API responses	The API responses include reusable data for subsequent requests	10
Test case generation technique	RT	*Random testing*	Parameters are assigned values randomly	All APIs	39
	CBT	*Constraint-based testing*	Constraint solvers are used to satisfy all inter-parameter dependencies of the API [88]	The API contains inter-parameter dependencies	27
	ART⋆	*Adaptive random testing*	Similar to RT or CBT, but aiming to generate as diverse test cases as possible [82]	The API contains at least one operation with more than 10 parameters	9

⋆ART bots employ RT or CBT (depending on whether the API has inter-parameter dependencies [88] or not) as a basis to generate test case candidates, from which the most diverse test cases are selected [82].

pattern in commercial APIs, where API users can only query existing data (e.g., characters from Marvel comics), and this data can only be modified by the API providers. However, we also test services whose main functionality is invoked via write operations (e.g., proofreading a text in LanguageTool), and even services which require stateful interactions (e.g., in Spotify, creating a playlist, adding songs to it, and deleting it).

3.3 Experimental Setup

In what follows, we describe the tooling used and how it was configured, and we explain several considerations to answer the research questions. All the experiments were performed in a single virtual machine equipped with 512GB RAM, 32 CPU cores and 2TB HDD running Red Hat Enterprise Linux 8, Java 8 and NodeJS 16.

3.3.1 Testing Ecosystem. For the generation and execution of test cases we used RESTest [91], an open source framework integrating varied test case generation strategies, a key requirement for our study. After some preliminary experiments, we soon realized that the monolithic approach of the tool—combining test case generation, execution and reporting in the same process—would hinder the development and deployment of the application at the desired scale. Inspired by the WES platform of Facebook [52], we conceived a novel architecture decoupling the key parts of the testing process into highly cohesive and autonomous programs called *bots*, illustrated in Figure 1. Bots can be independently developed and deployed using different technologies. We distinguish two types of bots: input and output bots. *Input bots* support the generation and execution of test cases. *Output bots* are responsible for analyzing and leveraging the test outputs. Bots are started, stopped and monitored automatically by a *controller* component, and they can optionally interact with each other, e.g., by triggering the update of test reports.

We devise two types of input bots: *test bots*, which generate and execute test cases, and *garbage collectors*, which delete resources created by test bots. Garbage collectors are implemented in JavaScript and test bots are implemented using RESTest. As illustrated in Table 2, test bots are classified along three dimensions, namely: (1) *write-safety*, which concerns the types of operations invoked by the bot; (2) *test data generation technique*, which refers to the input

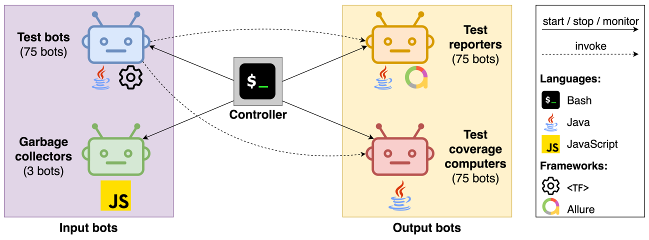


Figure 1: Testing ecosystem architecture.

data used in the API requests; and (3) *test case generation technique*, which relates to how API requests (i.e., test cases) are built. Test bots are created by combining these three dimensions in any way. As an example, *R-FD-RT* represents a *read-only* (R) bot that performs fuzzing [57], i.e., *random testing* (RT) with *fuzzing dictionaries* (FD). In total, we have 2 (write-safety) × 5 (test data generation technique) × 3 (test case generation technique) + 1 (garbage collector) = 31 types of input bots. Regarding output bots, we used both *test reporters* (TR), which generate graphical test reports with the Allure test reporting framework [1], and *test coverage computers* (TCC), which compute the test coverage achieved by test bots according to standard coverage criteria for RESTful web APIs [89].

3.3.2 Bots Selection Criteria. We selected a subset of bots for testing each API based on their potential applicability, for example, read/write bots are only applicable to APIs with write operations. Table 2 illustrates the inclusion criteria for selecting test bots in the APIs under test, and how many bots of each type were selected. According to these criteria, we selected 75 test bots, most of which are *read-only* (55 bots), use *customized data generators* (32), and apply *random testing* (39), whereas *read/write* bots (20), bots that use *semantically-related data* (5), and bots that apply *adaptive random testing* (9) are the least common. We may remark that we do not aim at doing a rigorous comparison of testing techniques. Instead, we aim to assess how different test generation strategies perform in an online scenario and how they complement each other. Table 4 shows the full list of test bots used in our experiment. Besides test bots, we configured three garbage collectors for the APIs of Spotify, Stripe and YouTube, and one test reporter and test coverage computer per test bot. As illustrated in Figure 1, we configured a total of 75 (test bots) + 3 (garbage collectors) + 75 (test reporters) + 75 (test coverage computers) = 228 bots.

3.3.3 *Bots Configuration.*

All test bots were configured to continuously generate and execute test cases during 15 days (378 hours). Bots were configured to comply with the quotas imposed by the APIs. Quotas describe the limitations of use for a fixed period of time [76] (e.g., 5K requests/day in Yelp [48]). To respect quotas, test bots work by iterations. In every iteration, they generate a set of test cases, execute them, and sleep for a given time. We configured all test bots of the same API to generate the same number of API calls per iteration, however, the sleep time between iterations was adjusted according to the test data generation technique used by the bot. FD *(fuzzing dictionaries)* and DP *(data perturbation)* test bots sleep four times longer than the rest, therefore they generate four times less API calls. This is because these bots are not expected to generate valid test cases that achieve a high API coverage, and so they are assigned a lower test budget [54, 68]. The specific configuration used for each bot (i.e., test cases per iteration and sleep time) is available in the supplementary material of the paper [92].

Test reporters and test coverage computers are invoked after every test bot iteration, continuously updating test reports and coverage. Garbage collectors are executed after test bots are stopped, thus deleting all resources created during the experiments.

3.3.4 *Test Oracles.*

To detect failures, we consider six test oracles, namely: (1) 5XX status codes (server errors) returned when sending a valid request (*F_5XX_V*); (2) 5XX status codes returned when sending an invalid request (*F_5XX_I*); (3) 2XX status codes (successful responses) with requests violating some parameter constraint (*F_2XX_P*); (4) 2XX status codes with requests violating some inter-parameter dependency [88] (*F_2XX_D*); (5) 400 status codes (client errors) with valid requests (*F_400*); and (6) disconformities with the OAS specification of the API (*F_OAS*). *F_5XX_V* and *F_5XX_I* failures are due to server errors, and so they can reveal API crashes (500 status codes) or temporary outages (503 status codes), among others. *F_2XX_P* and *F_2XX_D* failures reveal inconsistencies in the form of invalid API inputs (which expect a 4XX client error response) wrongly handled as valid (since they obtain a 2XX successful response). For instance, if a required parameter is not included in an API call, it should *not* obtain a 2XX status code (*F_2XX_P*). Likewise, if two mutually exclusive parameters are both included in an API call, it should *not* obtain a 2XX status code (*F_2XX_D*). *F_400* failures reveal unexpected client errors (400 status code) with API calls that are *valid*, therefore they should obtain a 2XX successful response. Lastly, *F_OAS* failures reveal conformance errors, for example, an API response not including a property defined as "required" in the API specification.

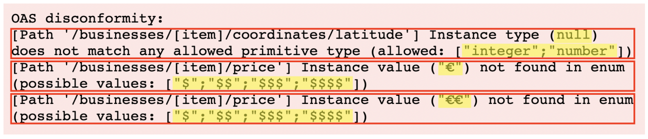
It is worth clarifying that not all test oracles are available to all test bots, and therefore different bots uncover different types of failures. In particular, disconformities with the API specification (*F_OAS*) and 5XX status codes (*F_5XX_V* and *F_5XX_I*) can be uncovered by any bot. On the other hand, to detect an unexpected client error (*F_400*), the API request must be valid, i.e., it must use valid input data. Only CDG bots can uncover this type of failures, since they always use valid data and therefore expect successful API responses. While SD and CD bots (i.e., semantic and contextual data bots) also have potential to uncover these faults, they are reported as "warnings", since they may use invalid data unintentionally, for instance, when extracting country names instead of country codes from a knowledge base [54]. Regarding *F_2XX_P* and *F_2XX_D* faults, they can only be detected by DP bots, since they purposely manipulate the input data used to make it invalid (by violating parameter constraints or inter-parameter dependencies), so they expect client error responses.

3.3.5 *Evaluation Metrics.*

We used the following metrics: (1) total number of failures and faults detected, overall and classified by type (i.e., according to the oracle violated); (2) failure and fault detection ratio (FDR and FtDR, respectively); (3) average percentage of faults detected (APFD); and (4) API test coverage. The APFD measures the weighted average of the percentage of faults detected over the life of the test suite, and it is a good representative of how fast faults are found [73]. The API test coverage is measured according to standard black-box coverage criteria for RESTful web APIs [89], implemented in tools like RESTest [91], Restats [69] and OpenAPI specification coverage [35], and previously used to compare black-box testing techniques [54, 68]. These criteria measure the degree to which a set of API requests and responses cover the elements defined in the API specification. API requests cover input elements such as operations and parameter values, while API responses cover output elements such as status codes and response body properties. As a difference compared to the original test coverage approach [89], we consider input criteria to be covered *only* if the API request covering the elements obtained a successful response, i.e., the API request was valid. For instance, if an API request obtains a 400 status code, we consider such code as "covered", but the parameter values used in the request are "not covered", since they were rejected by the API (client error). We follow this approach because invalid API requests do not generally exercise the core functionality of the API.

3.3.6 *Classifying Failures into Faults.*

We followed a two-step semi-automated approach to classify failures into unique faults: first, failures were automatically grouped into fault clusters; then, fault clusters were either confirmed (if they represented unique faults) or discarded, manually. A fault cluster represents a unique potentially incorrect behavior in the API (e.g., a server error when setting a parameter with certain value), and it comprehends all the failures revealing that issue. We consider that two failures belong to the same fault cluster, and therefore they are caused by the same potential fault, if: (1) they violate the same test oracle, (2) they occur in the same API operation (e.g., `GET /search`), (3) they have the same status code (e.g., 404) and content-type (e.g., `application/json`), and (4) they are similar enough. Similarity between failures was measured using the normalized Levenshtein distance (NLD) [102] to compare HTTP requests, HTTP responses and error logs. If the distance is lower than certain threshold, failures are considered similar. The only exception was *F_OAS* failures, whose error logs are composed of structured error messages containing dynamic values (see example in Figure 2). These failures are considered similar if, after replacing dynamic values with wild cards and removing duplicate error messages, the resulting sanitized error logs are the same. Table 3 summarizes the heuristics and thresholds used.



```
OAS disconformity:
[Path '/businesses/[item]/coordinates/latitude'] Instance type (null)
does not match any allowed primitive type (allowed: ["integer";"number"])
[Path '/businesses/[item]/price'] Instance value ("€") not found in enum
(possible values: ["$";"$$";"$$$";"$$$$"])
[Path '/businesses/[item]/price'] Instance value ("€€") not found in enum
(possible values: ["$";"$$";"$$$";"$$$$"])
```

Figure 2: *F_OAS* error log. Red boxes depict error messages. Yellow highlighting depicts dynamic values.

Table 3: Heuristics to determine similar failures.

Type	Failures f_1 and f_2 are similar if:
F_5XX_V	$NLD(response_{f_1}, response_{f_2}) < 0.5$
F_5XX_I	$NLD(response_{f_1}, response_{f_2}) < 0.5 \ \& \ NLD(error\_log_{f_1}, error\_log_{f_2}) < 0.5$
F_2XX_P	$NLD(error\_log_{f_1}, error\_log_{f_2}) < 0.5$
F_2XX_D	$NLD(request_{f_1}, request_{f_2}) < 0.1$
F_400	$NLD(response_{f_1}, response_{f_2}) < 0.5 \ \& \ NLD(request_{f_1}, request_{f_2}) < 0.25$
F_OAS	$unique(wilcard(error\_log_{f_1})) == unique(wilcard(error\_log_{f_2}))$

4 RESULTS

Next, we describe the results of our study and how they answer the target research questions.

4.1 RQ$_1$: Failure Detection Capability

Table 4 shows the number of test cases, API coverage, failures and faults obtained in every API under test and by each test bot. As illustrated, a total of 389,216 failures were uncovered, i.e., 1 every 3 test cases. The majority of failures (243,394) consist in disconformities of the responses with the API specification (*F_OAS*). In fact, this is one of the main conclusions derived from our study: *API specifications poorly reflect the actual API implementation.* We found inconsistencies in all APIs under test except in RESTCountries, but this is because the documentation of this API did not explicitly state the format of the responses, hence no inconsistencies could be found. Disconformities include status codes and content-types not listed in the specification, as well as violations in the format of the response bodies, e.g., an object missing a required property. Systematic violations of the API contracts mean a strong limitation to client applications, which could crash if they run into a scenario not described (or even forbidden) in the API specification (e.g., a non-nullable property being null). Besides OAS disconformities, we uncovered thousands of other failures related to server errors (5XX status codes), client errors (400 status codes) and wrongly returned successful responses (2XX status codes).

We studied the FDR for all APIs under test. Overall, the FDR ranged from 0.04 (9,141 failures out of 258,297 test cases) in the API of RESTCountries to 0.79 (31,811 out of 40,480) in the API of LanguageTool. We also studied the evolution of the FDR over time. This allowed us to detect different tendencies and anomalies in several APIs. Figure 3 depicts the FDR over the test suite fraction in five APIs: Amadeus, DHL, Ohsome, Spotify, and Yelp. The remaining APIs show a similar trend to the one observed in Spotify, and are not included for the sake of clarity (charts for all APIs are available in the supplementary material [92]). Spotify represents a common scenario, where the FDR remains constant, i.e., the same failures are uncovered all the time. In Amadeus, the FDR slightly declines, meaning that test cases fail less often over time. We suspect this

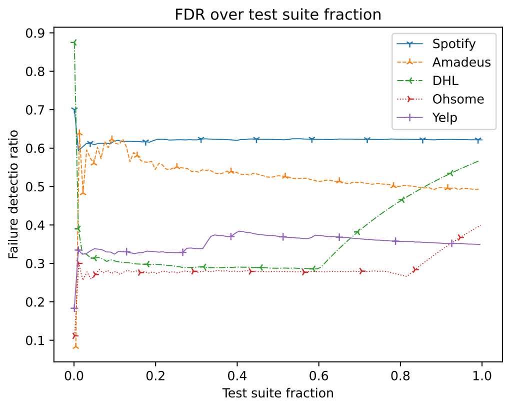


Figure 3: FDR in Spotify, Amadeus, DHL, Ohsome and Yelp.

happens because API developers are continuously fixing issues. The APIs of DHL, Ohsome and Yelp are clear anomalies. In DHL, all test cases started to fail at 60% of the overall test budget, because we were banned from the API, even though we always complied with the imposed quota. In Ohsome, a similar phenomenon was observed, but the failures were due to 500 status codes, all with the same error message: "`The timerange metadata could not be retrieved from the db`". In Yelp, the FDR spiked four times because the API only returned 500 status codes, likely due to some non-functional bug causing temporary outages.

Summary of answers to RQ$_1$

(1) Automatically generated test cases uncovered about 390K failures (1 out of every 3 test cases) in the APIs under test.
(2) Failures were revealed in all APIs, with the FDR ranging from 0.04 to 0.79.
(3) 63% of failures are due to inconsistencies between the API responses and their specifications.
(4) The FDR evolution revealed anomalies—temporary outages and API bans—in DHL, Ohsome and Yelp.

4.2 RQ$_2$: Fault Detection Capability

Our fault clustering approach yielded 4,818 potential faults (i.e., fault clusters) from the 389,216 test failures revealed. FDIC [24] and Ohsome [31] were the APIs for which most potential faults were detected, 2,871 and 1,445, respectively. We discarded these APIs when assessing the fault detection capability of our online testing setup, as we could not manually analyze all these fault clusters. We did analyze a subset of them, nonetheless, and we identified numerous bugs (more details in Section 5). Without considering FDIC and Ohsome, a total of 218,419 test failures were automatically clustered into 502 potential faults which, after manual revision, resulted in 139 reproducible (or already fixed) issues in all the APIs under test. Besides the issues extracted from the fault clusters, we spotted 8 additional issues from the reports generated by the test reporter bots (available at [33]).

Overall, we uncovered a total of 147 faults in 12 APIs. Half of these faults (73) are disconformities between the API responses and their specifications (*F_OAS*), which is coherent with the findings

Table 4: Per-bot breakdown of test cases, coverage, failures and faults. Last column depicts unique faults, i.e., only uncovered by that bot. Rows in boldface depict bots uncovering the most faults. Rows highlighted in gray depict the total stats per API.

API	Bot	Test cases	Coverage (%)	Failures/Faults							Unique faults
				F_5XX$_V$	*F_5XX*$_I$	*F_400*	*F_2XX*$_P$	*F_2XX*$_D$	*F_OAS*	Total	
Amadeus	**R-CDG-ART**	**384**	**78.77**	**0/0**	**0/0**	**137/7**	**0/0**	**0/0**	**112/5**	**249/12**	**3**
Amadeus	R-CDG-CBT	384	76.42	0/0	0/0	203/7	0/0	0/0	53/2	256/9	0
Amadeus	R-DP-CBT	384	76.42	0/0	0/0	0/0	37/3	32/0	0/0	69/3	3
Amadeus	R-SD-CBT	384	74.53	0/0	0/0	0/0	0/0	0/0	13/2	13/2	0
Amadeus	R-CD-CBT	384	72.17	0/0	0/0	0/0	0/0	0/0	5/2	5/2	0
Amadeus	R-FD-RT	408	3.3	0/0	0/0	0/0	0/0	0/0	48/1	48/1	1
Amadeus	R-CDG-RT	384	76.42	0/0	0/0	0/0	0/0	0/0	12/2	12/2	0
Amadeus	All bots	2,712	-	0/0	0/0	340/7	37/3	32/0	243/6	652/16	-
DHL	R-FD-RT	2,280	13.7	0/0	0/0	0/0	0/0	0/0	913/2	913/2	0
DHL	R-CDG-RT	8,976	91.78	1/0	0/0	90/3	0/0	0/0	3,678/1	3,769/4	3
DHL	**R-DP-RT**	**2,280**	**97.26**	**0/0**	**15/1**	**0/0**	**516/3**	**0/0**	**912/2**	**1,443/6**	**4**
DHL	R-SD-RT	8,952	91.78	2/0	0/0	0/0	0/0	0/0	3,587/1	3,589/1	0
DHL	R-CD-RT	8,951	87.67	0/0	0/0	0/0	0/0	0/0	6,280/2	6,280/2	1
DHL	All bots	31,439	-	3/0	15/1	90/3	516/3	0/0	15,370/3	15,994/10	-
FDIC	R-CDG-ART	12,744	85.54	190/-	0/-	2,141/-	0/-	0/-	10,113/-	12,444/-	-
FDIC	R-FD-RT	3,456	85.54	0/-	139/-	0/-	0/-	0/-	140/-	279/-	-
FDIC	R-CDG-RT	12,240	85.54	188/-	0/-	1,956/-	0/-	0/-	9,125/-	11,269/-	-
FDIC	R-DP-RT	3,456	85.54	0/-	252/-	0/-	409/-	0/-	0/-	661/-	-
FDIC	All bots	31,896	-	378/-	391/-	4,097/-	409/-	0/-	19,378/-	24,653/-	-
Foursquare	R-CDG-ART	12,240	65.79	7/0	0/0	2/0	0/0	0/0	12,024/3	12,033/3	0
Foursquare	R-CDG-CBT	22,200	65.79	4/0	0/0	23/0	0/0	0/0	16,229/3	16,256/3	0
Foursquare	**R-DP-CBT**	**5,700**	**68.42**	**0/0**	**3/0**	**0/0**	**617/2**	**1,905/3**	**2,528/2**	**5,053/7**	**6**
Foursquare	R-FD-RT	5,700	63.16	0/0	1/0	0/0	0/0	0/0	1,516/2	1,517/2	0
Foursquare	R-CDG-RT	22,200	65.79	0/0	4/0	0/0	0/0	0/0	15,108/3	15,112/3	0
Foursquare	All bots	68,040	-	11/0	8/0	25/0	617/2	1,905/3	47,405/4	49,971/9	-
LanguageTool	**RW-CDG-ART**	**6,680**	**47.95**	**0/0**	**0/0**	**1,177/1**	**0/0**	**0/0**	**5,503/5**	**6,680/6**	**0**
LanguageTool	**RW-CDG-CBT**	**7,500**	**47.95**	**0/0**	**0/0**	**1,018/1**	**0/0**	**0/0**	**6,482/5**	**7,500/6**	**0**
LanguageTool	RW-DP-CBT	1,900	45.21	0/0	0/0	0/0	82/1	0/0	1,818/1	1,900/2	1
LanguageTool	RW-SD-CBT	7,500	45.21	0/0	0/0	0/0	0/0	0/0	3,126/4	3,126/4	0
LanguageTool	RW-CD-CBT	7,500	53.42	2/0	0/0	0/0	0/0	0/0	3,203/4	3,205/4	0
LanguageTool	RW-FD-RT	1,900	0	0/0	0/0	0/0	0/0	0/0	1,900/1	1,900/1	0
LanguageTool	RW-CDG-RT	7,500	47.95	0/0	2/0	0/0	0/0	0/0	7,498/5	7,500/5	0
LanguageTool	All bots	40,480	-	2/0	2/0	2,195/1	82/1	0/0	29,530/6	31,811/8	-
Marvel	R-CDG-ART	14,273	95.45	3/0	0/0	0/0	0/0	0/0	7,233/14	7,236/14	0
Marvel	R-FD-RT	3,744	23.86	0/0	3/1	0/0	0/0	0/0	590/12	593/13	11
Marvel	R-CDG-RT	14,507	95.45	4/0	0/0	0/0	0/0	0/0	5,726/14	5,730/14	0
Marvel	R-DP-RT	3,744	88.64	0/0	41/1	0/0	1,332/10	0/0	0/0	1,373/11	11
Marvel	**R-CD-RT**	**14,508**	**96.02**	**2/0**	**0/0**	**0/0**	**0/0**	**0/0**	**7,408/16**	**7,410/16**	**1**
Marvel	All bots	50,776	-	9/0	44/2	0/0	1,332/10	0/0	20,957/26	22,342/38	-
Ohsome	R-CDG-CBT	73,198	80.55	8,477/-	0/-	19,389/-	0/-	0/-	34,855/-	62,721/-	-
Ohsome	R-DP-CBT	56,120	21.45	0/-	13,100/-	0/-	283/-	6/-	0/-	13,389/-	-
Ohsome	R-FD-RT	56,730	9.98	0/-	12,815/-	0/-	0/-	0/-	0/-	12,815/-	-
Ohsome	R-CDG-RT	178,120	80.23	0/-	46,209/-	0/-	0/-	0/-	11,010/-	57,219/-	-
Ohsome	All bots	364,168	-	8,477/-	72,124/-	19,389/-	283/-	6/-	45,865/-	146,144/-	-
OMDb	R-CDG-CBT	13,388	78.26	0/0	0/0	0/0	0/0	0/0	725/1	725/1	0
OMDb	**R-DP-CBT**	**3,420**	**78.26**	**0/0**	**36/1**	**0/0**	**1,674/2**	**1,710/1**	**0/0**	**3,420/4**	**3**
OMDb	R-SD-CBT	11,518	78.26	0/0	0/0	0/0	0/0	0/0	149/1	149/1	0
OMDb	R-CD-CBT	13,388	78.26	0/0	0/0	0/0	0/0	0/0	700/2	700/2	0
OMDb	**R-FD-RT**	**3,420**	**86.96**	**0/0**	**552/1**	**0/0**	**0/0**	**0/0**	**261/3**	**813/4**	**1**
OMDb	R-CDG-RT	13,391	78.26	1/0	0/0	0/0	0/0	0/0	668/1	669/1	0
OMDb	All bots	58,525	-	1/0	588/1	0/0	1,674/2	1,710/1	2,503/3	6,476/7	-
RESTCountries	R-FD-RT	19,739	48.34	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0
RESTCountries	R-CDG-RT	72,869	82.12	224/0	0/0	0/0	0/0	0/0	0/0	224/0	0
RESTCountries	**R-DP-RT**	**19,740**	**49.01**	**1/0**	**2/0**	**0/0**	**8,478/11**	**0/0**	**0/0**	**8,481/11**	**11**
RESTCountries	R-SD-RT	73,290	77.48	218/0	0/0	0/0	0/0	0/0	0/0	218/0	0
RESTCountries	R-CD-RT	72,659	70.2	218/0	0/0	0/0	0/0	0/0	0/0	218/0	0
RESTCountries	All bots	258,297	-	661/0	2/0	0/0	8,478/11	0/0	0/0	9,141/11	-
Spotify	R-CDG-RT	22,139	27.43	1/0	0/0	0/0	0/0	0/0	16,832/8	16,833/8	0
Spotify	RW-FD-RT	5,700	17.4	0/0	0/0	0/0	0/0	0/0	662/1	662/1	0
Spotify	RW-DP-RT	5,700	20.65	0/0	0/0	0/0	58/2	0/0	475/0	533/2	2
Spotify	**RW-CD-RT**	**22,140**	**72.27**	**26/1**	**0/0**	**0/0**	**0/0**	**0/0**	**11,863/13**	**11,889/14**	**6**
Spotify	All bots	55,679	-	27/1	0/0	0/0	58/2	0/0	29,832/13	29,917/16	-
Stripe	R-CDG-CBT	11,160	17.24	0/0	0/0	0/0	0/0	0/0	773/1	773/1	0
Stripe	RW-CDG-ART	10,737	25.52	0/0	0/0	9/0	0/0	0/0	10,728/1	10,737/1	0
Stripe	RW-CDG-CBT	11,158	25.52	0/0	0/0	16/0	0/0	0/0	11,142/1	11,158/1	0
Stripe	RW-DP-CBT	2,850	17.24	0/0	0/0	0/0	0/0	4/1	0/0	4/1	1
Stripe	**RW-CD-CBT**	**11,098**	**91.03**	**0/0**	**0/0**	**0/0**	**0/0**	**0/0**	**2,616/4**	**2,616/4**	**2**
Stripe	RW-FD-RT	2,850	3.45	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0
Stripe	RW-CDG-RT	11,158	25.52	0/0	0/0	0/0	0/0	0/0	1,432/1	1,432/1	0
Stripe	All bots	61,011	-	0/0	0/0	25/0	0/0	4/1	26,691/4	26,720/5	-
Yelp	**R-CDG-ART**	**11,340**	**100**	**2,088/0**	**0/0**	**1,226/1**	**0/0**	**0/0**	**1,875/6**	**5,189/7**	**1**
Yelp	R-CDG-CBT	22,140	100	8,110/0	0/0	2,344/1	0/0	0/0	1,554/5	12,008/6	0
Yelp	R-DP-CBT	5,700	100	0/0	456/0	0/0	76/1	221/1	154/1	907/3	3
Yelp	R-FD-RT	5,700	54.55	0/0	1/1	0/0	0/0	0/0	0/0	1/1	0
Yelp	R-CDG-RT	22,200	100	0/0	3,988/0	0/0	0/0	0/0	1,299/3	5,287/3	0
Yelp	All bots	67,080	-	10,198/0	4,445/1	3,570/1	76/1	221/1	4,882/6	23,392/10	-
YouTubeComments	R-CDG-CBT	1,152	11.84	0/0	0/0	128/1	0/0	0/0	0/0	128/1	1
YouTubeComments	R-CDG-RT	1,152	11.84	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0
YouTubeComments	RW-CD-ART	1,152	49.05	0/0	0/0	0/0	0/0	0/0	46/1	46/1	0
YouTubeComments	**RW-DP-CBT**	**288**	**0**	**0/0**	**0/0**	**0/0**	**27/7**	**2/1**	**0/0**	**29/8**	**8**
YouTubeComments	RW-CD-CBT	1,152	47.57	0/0	0/0	0/0	0/0	0/0	18/1	18/1	0
YouTubeComments	RW-FD-RT	288	0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0
YouTubeComments	All bots	5,184	-	0/0	0/0	128/1	27/7	2/1	64/1	221/10	-
YouTubeSearch	**R-CDG-ART**	**1,840**	**87.5**	**0/0**	**0/0**	**456/3**	**0/0**	**0/0**	**417/1**	**873/4**	**0**
YouTubeSearch	**R-CDG-CBT**	**1,885**	**87.5**	**0/0**	**0/0**	**486/3**	**0/0**	**0/0**	**230/1**	**716/4**	**0**
YouTubeSearch	R-DP-CBT	475	85.58	0/0	0/0	0/0	165/2	1/1	0/0	166/3	3
YouTubeSearch	R-FD-RT	474	2.88	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0
YouTubeSearch	R-CDG-RT	1,885	86.54	0/0	0/0	0/0	0/0	0/0	27/1	27/1	0
YouTubeSearch	All bots	6,559	-	0/0	0/0	942/3	165/2	1/1	674/1	1,782/7	-
Total	All APIs	1,101,846	-	19,767/1	77,619/5	30,801/16	13,754/44	3,881/8	243,394/73	389,216/147	-

obtained in terms of failure detection capability (RQ_1). About one third of the faults (52) are caused by invalid API inputs handled as valid (F_2XX_P and F_2XX_D). This type of faults are pervasive: APIs are sometimes designed to return successful responses even when receiving invalid inputs (e.g., by ignoring them). For instance, in the Stripe API [45], when using two mutually exclusive parameters, the API ignores one of them instead of properly returning a 400 (“`Bad Request`”) status code. This was confirmed by the Stripe API developers. Invalid inputs do, however, cause API crashes in some cases, due to poor input validation. We found 5 bugs of this type (F_5XX_I) in 4 APIs (DHL, Marvel, OMDb and Yelp). Server errors on valid inputs (F_5XX_V) are a much more severe fault, since they may be caused by misconfigurations or data corruption, among others. We found one of these in the Spotify API, unveiled when reordering or replacing the tracks of a playlist. Lastly, we uncovered 16 client errors obtained with valid API inputs (*F_400*). These faults generally reveal inconsistencies between the documentation and the implementation of an API. For instance, in the DHL API, the country code ‘`KV`’ (Kosovo) is rejected by the API (400 status code), although it should be supported according to the API documentation [23].

The FtDR ranged from 0.04×10^{-3} (11 faults out of 258,297 test cases) in the API of RESTCountries to 5.8×10^{-3} (16 out of 2,712) in the API of Amadeus. An overview of when and how often faults are found is shown in Figure 4, which shows the APFD for every API under test. For all APIs, two thirds of the test budget (i.e., the test cases generated in 10 days) was enough to uncover all faults. LanguageTool, RESTCountries and Stripe exposed all their faults within the first day of testing. These faults were related to disconformities with the API specifications (*F_OAS*) and wrong handling of invalid inputs (F_2XX_P and F_2XX_D). On the other hand, Amadeus, DHL, Marvel and YouTubeComments required over one week to uncover all faults. More varied issues were found in these cases, as explained in Section 5.

Summary of answers to RQ_2

(1) Automatically generated test cases found 147 faults in 11 APIs (65 of which confirmed or fixed by developers).
(2) Half of the faults (73) are caused by OAS disconformities.
(3) About one third of the faults (52) are caused by invalid API inputs handled as valid.
(4) Only 6 faults are due to internal server errors (5XX status codes).
(5) 87% of faults (128) were found in the first 3 days of testing, with 10 days being enough to uncover all faults.

4.3 RQ_3: Comparison of Testing Approaches

All testing approaches uncovered faults, even after 3 days of testing, but we found several differences among the test data and test case generation techniques under evaluation, discussed below. As previously mentioned, the results for the APIs of FDIC and Ohsome are discussed later (Section 5.1) due to the number of failures revealed. Thus, the results reported next refer to the 11 remaining APIs.

All test data generation techniques except semantically-related data (SD) found unique bugs or contributed to improve the coverage in all APIs under test. Regarding coverage, customized data generators (CDG) and contextual data (CD) obtained the best results in most APIs (8 out of 11). This was expected, since valid data tends

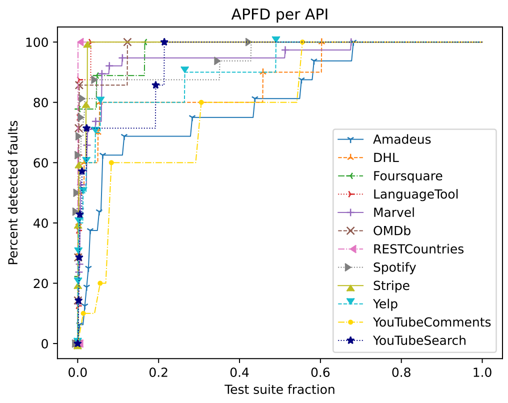


Figure 4: APFD per API. Markers denote faults.

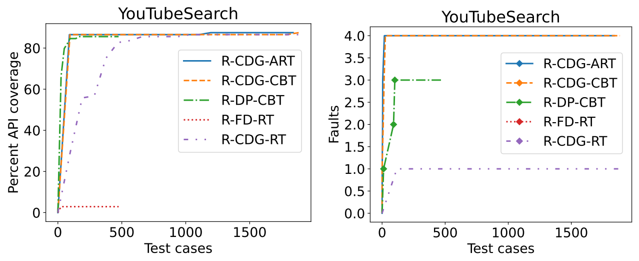


Figure 5: Coverage and faults over test cases classified per bot for the YouTubeSearch API. Markers denote unique faults.

to cover more input elements (e.g., parameter values) and to obtain successful API responses, thus covering more output elements (e.g., response body properties). Invalid data—data perturbation (DP) and fuzzing dictionaries (FD)—achieved the highest coverage in 3 APIs, because they covered client and server errors (4XX and 5XX status codes) that valid data could not cover. In terms of faults found, valid and invalid data generation uncovered 76 and 81 faults, respectively (10 of these faults were found by both approaches). In fact, the bugs uncovered by DP are often unique (i.e., not uncovered by other techniques). Figure 5 (right-hand side) illustrates this phenomenon in the API of YouTubeSearch (charts of all APIs available in the supplementary material [92]), where all bugs found by the DP bot were unique. Overall, DP found the largest amount of unique bugs, 55 in total, followed by CDG with 27. Different techniques uncover different types of bugs. For instance, F_2XX_P and F_2XX_D faults can only be detected by DP bots, since they purposely manipulate the input data used to make it invalid. Similarly, *F_400* faults can only be detected by CDG bots, since they use manually set valid data which should never obtain 400 status codes.

When using the same test data generation technique, different test case generation strategies achieved similar coverage. However, constraint-based testing (CBT) was more cost-effective than random testing (RT). For instance, in YouTubeSearch (left-hand side of Figure 5), CBT required just about 50 test cases to achieve the same coverage as RT with about 700. Regarding faults found, RT, CBT and adaptive random testing (ART) found 63, 31 and 4 unique bugs, respectively. However, in the APIs where it was evaluated, CBT found more bugs than RT. At the same time, ART did not yield any improvement over RT or CBT, except in Amadeus, where 3 more bugs were found, compared to CBT. In particular, these faults belong to the *F_OAS* category, therefore the diversification of API requests may have led to more diverse API responses, where more varied OAS disconformities were found.

Summary of answers to **RQ$_3$**

(1) Valid and invalid test data generation strategies (i.e., positive and negative testing) are clearly complementary, as they uncovered a similar number of unique bugs.
(2) All test data generation strategies contributed to detect unique faults or improve the coverage in some API, being data perturbation the most effective one. The only exception was semantic data.
(3) All test case generation strategies contributed to detect unique faults or improve the coverage in some API, being constraint-based testing the most effective one.
(4) Test case diversity in terms of API parameters and input values does not seem to help to find new bugs or achieve higher API coverage.

5 DISCUSSION

In this section, we discuss the results of FDIC and Ohsome, where thousands of potential faults were identified, as well as some of the issues detected in all APIs and the overall cost of our experiments.

5.1 FDIC and Ohsome APIs

In the APIs of FDIC and Ohsome, a total of 24,653 and 146,144 test failures were classified into 2,871 and 1,445 potential faults (i.e., fault clusters), respectively. In FDIC, 2,686 of these clusters (94% of the total) are related to disconformities with the OAS specification. After a quick analysis, we conjecture that most of these clusters represent, indeed, *unique* faults. This is because the specification of this API is so complex and its responses are so varied that every test case can potentially uncover new inconsistencies in the OAS specification. In Ohsome, the same types of faults seem to occur in all its 122 operations. This explains the high number of fault clusters found. Although it is possible that a single fault manifest in multiple operations, this cannot be known in a black-box testing setup, therefore we adopt the same criteria as previous authors, where faults in different operations are considered unique [55, 70, 86].

We analyzed a subset of fault clusters in FDIC and Ohsome to grasp a better understanding of how effectively our fault clustering approach found real-world bugs in these APIs, and what shape they have. We created the subset as follows: we sorted fault clusters over time and we selected 42 clusters (mean number of clusters found in the other APIs under test) equidistant to each other. For example, in Ohsome, out of the 1,445 faults clusters found, we analyzed those at positions $[1, 36, 71, ..., 1375, 1410, 1445]$. In FDIC, all fault clusters were unique faults. We also spotted 33 additional bugs in the graphical test reports [33], conforming a total of 75 unique bugs in this API. In Ohsome, 28 out of 42 fault clusters were unique faults. Additionally, we spotted 3 faults related to 500 status codes in the test reports, making a total of 31 unique bugs. In both cases, more faults could have been manually found in the test reports, for instance, faults affecting other API operations in Ohsome, or more disconformities with the OAS specification in FDIC.

5.2 Issues Detected

Testing APIs in production can be helpful to uncover problems that escaped offline testing. In fact, online testing allowed us to detect not only functional bugs, but also problems related to non-functional requirements of the APIs under test. We detail these first, and then we delve into some of the functional bugs found.

5.2.1 *Non-Functional Requirements.*

We found problems related to security, reliability, availability and SLA compliance.

Security. When using malformed input data, we found 500 status codes showing the stack trace of the exception thrown in the OMDb API [13], and 400 status codes with the error message “`Threat Detected`” in the DHL API [7]. Disclosing this information to a potential attacker is dangerous, since they could exploit vulnerabilities revealed in the stack trace (e.g., outdated libraries) or find an input compromising the integrity of the system.

Reliability and availability. We detected temporary outages in the form of 5XX status codes in 9 APIs (DHL, Foursquare, LanguageTool, Marvel, Ohsome, OMDb, RESTCountries, Spotify and Yelp). Outages are generally caused by limitations of the servers, and they can affect the overall user experience, especially when they are prolonged in time (e.g., as in Yelp and Ohsome).

SLA compliance. In the ninth day of testing, we were banned from the DHL API, even though we always complied with the specified quota limitations. SLA-aware API specifications such as SLA4OAI [75] could help detect more SLA violations like this in a more systematic way.

5.2.2 *Functional Bugs.*

We identified 254 reproducible bugs in all APIs under test—209 extracted from 586 fault clusters, 44 found in the test reports [33], and 1 found by the garbage collector bot of Spotify (when unfollowing playlists, the bot would start obtaining server errors if requests were sent quickly, e.g., every 5 milliseconds [16]). This is a conservative approximation, since more bugs could have been found with a more sophisticated and precise fault clustering approach (Section 3.3.6), or with a more thorough manual analysis of the potential faults found in FDIC and Ohsome. To date, 65 bugs from 7 APIs (Amadeus, DHL, Foursquare, OMDb, RESTCountries, Yelp and YouTube) have been acknowledged by developers, reproduced by other users or already fixed. Next, we describe some types of bugs uncovered in our experiments (all bugs are documented in the supplementary material [92]).

Errors due to invalid data introduced by API clients. In the Amadeus API, hotel objects must contain two properties, `phone` and `fax`, both of which must match the regular expression `‘[0-9]{2}’`. We found response bodies whose hotel objects contained invalid phone numbers (e.g., `‘+1 2 9’`) [5] and missing fax numbers [6]. Client applications may not be able to parse such API responses, or even crash. Amadeus developers confirmed that: (1) these errors are caused by hotel providers themselves, as they may introduce invalid hotel data; and (2) they would update the API specification to reflect this new scenario. Issues like these can hardly be discovered with offline testing, since they deal with data from the real-world systems.

Inconsistencies between API implementation and API documentation. Industrial APIs are complex and evolve rapidly, and so does their documentation. In some cases, the implementation and documentation of an API may not be in sync. For instance, the YouTube search API documentation [51] does not state that parameters `location` and `channelType` are mutually exclusive, although when using both in an API request, a 400 status code is obtained [19]. Similarly, the documentation of the YouTube comments API [49] states that parameters `id` and `maxResults` are mutually exclusive, although when using both in an API request, a successful response is obtained [18]. YouTube developers acknowledged both issues, the former as *incomplete documentation* (they updated the API documentation accordingly) [21], and the latter as an *implementation defect* (they introduced a bug fix) [20]. A continuous online testing setup can rapidly detect these inconsistencies.

Internal server errors with valid input data. API crashes caused by invalid requests—for instance, using an out-of-range value for an `enum` parameter in Marvel [11], or using a negative number for the `limit` parameter in DHL [9]—are relatively common and easy to fix, namely, by doing proper input validation. However, server errors on valid inputs represent a more serious fault. For instance, in the Ohsome API, a valid request obtained a 500 status code with the error message "`No message available`" [12]. Similarly, in the FDIC API, multiple valid requests obtained server errors with the message "`Cannot read property 'map' of undefined`" [10]. Debugging these faults is hard, especially due to the little information provided in the error messages.

Beyond all these bugs, we uncovered dozens of other types of issues related to OAS disconformities [17], inconsistencies between the status codes and response bodies [15], unparseable JSON responses [14] and unexpected client errors [8], among many others.

5.3 Cost of our Online Testing Setup

Our testing ecosystem is heavily based on RESTest. This greatly influenced the manual work required to deploy all test bots, as well as the overall consumption of computational resources.

In terms of manual work, we had to configure 75 test bots (RESTest instances), which involved writing both test configuration files and data dictionaries (mainly used by CDG bots). For the test configuration files, we wrote about 26K lines of code (LOCs), although about 95% of them were duplicated, i.e., copied and pasted. This high percentage reveals that the data format used for configuration files in RESTest should be improved necessarily, to make it less verbose. Despite this limitation, half of the bots required just 2 or less manually written LOCs, since the configuration of different bots of the same API can be reused. Regarding data dictionaries, we used 50 dictionaries containing 52,084 values, about 1K values per dictionary. Note that data dictionaries are generally not manually written, but rather extracted from the API documentation (e.g., Foursquare categories [25] and Stripe tax codes [46]) or other general-purpose knowledge bases (e.g., country codes in Wikipedia [47]).

In terms of computational resources, the RAM and disk consumption kept increasing over the 15 days of online testing. Preliminary experiments show that RAM usage could be reduced by restarting bots regularly (e.g., every day), to avoid potential memory leaks. We also measured the computational cost per bot. Although we did not find noticeable differences across different bots of the same API, we did find evident differences between different APIs. For instance, the bots testing FDIC and RESTCountries used much more RAM (up to 20GB) and took up much more disk space (up to 250GB) than the rest. This is due to the shape and size of the responses obtained in these APIs, some of which took hundreds of MBs. Handling such responses (e.g., parsing and analyzing them in the search for faults) implies a non-negligible overhead.

6 CHALLENGES

We identify the following challenges for the adoption of test case generation techniques for online testing of RESTful APIs.

Challenge #1: *Automated fault identification.* Automatically determining the root causes of the many failures found (about 390K in our work) is challenging. Our fault clustering approach helped us identify 139 unique bugs out of 502 potential faults, but this also means that 363 potential faults were misclassified (i.e., duplicated, non-reproducible or not actual faults), and 8 bugs that we spotted in the test reports were missed in the fault clustering. Faults are misclassified due to the heterogeneity of error logs, requests, and especially responses. The thresholds used may not be equally effective for all failures. More sophisticated fault identification approaches are desirable, e.g., using dynamic thresholds according to the type of failure and API, applying non-supervised machine learning techniques to improve the precision of the automatic classification, or employing delta debugging to isolate failure-inducing inputs [103].

Challenge #2: *Effective human interaction.* While bots work mostly autonomously, they can benefit greatly from some human interaction. For instance, fault clusters could be formed by output bots, and then checked by a human to improve the accuracy of the classification and to avoid the formation of duplicated clusters. Similarly, bots could report some failures as "potential faults" and request human input to confirm or discard them. This could be addressed with active learning algorithms [98], for example, integrating human input as a part of the testing and debugging process.

Challenge #3: *Optimal selection of testing strategies.* Testers may choose different testing strategies according to several factors such as the types of bugs that they wish to uncover (e.g., crashes, regressions or inconsistencies between the API and its documentation), or the characteristics of the API under test (e.g., size, output format and quota limitations). Automatically determining the most appropriate testing techniques based on these and other factors is an open problem.

Challenge #4: *Optimal test execution scheduling.* The test execution schedule determines when, how many, and in what order tests should be executed. This was manually configured in our experiments. Ideally, however, the schedule should be automatically computed to optimize the available quota (based on the API pricing plans), the time and economic budget (e.g., the infrastructure costs), and the testing strategies used (e.g., prioritizing those strategies that are more cost-effective).

Challenge #5: *Optimization of computational resources.* This is both an engineering and a research challenge. Test bots should be optimized to use the least resources possible, especially avoiding memory leaks. From a research point of view, sophisticated techniques for test regression, selection and prioritization could help in devising more effective test suites or reusing existing ones (e.g., with metamorphic testing [97]), thus saving resources by avoiding the creation and execution of test cases that are not likely to uncover new faults.

7 THREATS TO VALIDITY

Our work is subject to a number of validity threats, discussed below.

Internal validity. Threats to the internal validity relate to those factors that may affect the results of our evaluation. Faults in the implementation of our testing ecosystem might compromise the validity of our conclusions. Even though the tools used (e.g., RESTest [91] and Allure [1]) are thoroughly tested, it cannot be guaranteed that they are free of bugs. To mitigate this threat, and to enable replicability of our results, we provide a supplementary package containing the source code and the implementation of all tools used in our testing ecosystem (e.g., the bots and the controller component), as well as scripts to fully replicate the results reported in this paper [92].

The use of RESTest as the selected test case generation tool may have influenced the results obtained in our experiments, especially depending on how the evaluated techniques (e.g., data perturbation and adaptive random testing) were implemented in the framework. However, this does not invalidate our results; on the contrary, it supports the potential of our multi-bot architecture to integrate other testing techniques and tools (e.g., RESTler [57] and RestCT [101]).

To mitigate possible errors of our fault clustering approach (Section 3.3.6), we manually reviewed all fault clusters generated, and reported all identified faults to the API developers, so as to count on their confirmation when possible. We adopt a conservative approximation and report only faults that have been manually confirmed. Despite possible limitations of our fault clustering approach, it automatically classified 218,419 failures into 502 potential faults in 11 APIs. From these, we identified 139 actual unique bugs, 65 of which have been acknowledged or fixed by API developers to date.

External validity. Threats to the external validity might affect the generalizability of our findings. The main threat in this regard is the selection of case studies. We tested 13 industrial APIs comprising 189 operations in total, although this might not be a sufficiently representative sample. To minimize this threat, we selected highly popular APIs with millions of users worlwide, from different application domains (e.g., media, financial and social), and with diverse characteristics and sizes.

8 CONCLUSIONS

In this paper, we assessed the potential of automated black-box test case generation approaches for online testing of RESTful APIs, resembling the model of testing as a service increasingly found in industry. To this end, we devised a multi-bot architecture allowing us to generate and execute test cases with varied strategies during 15 days non-stop in 13 highly popular APIs. The results are promising, with over 200 bugs found, both functional and non-functional, some of which have led developers to take actions including fixes and documentation updates in the APIs of Amadeus and YouTube. This provides an encouraging vision on the future of testing of web APIs as a service, where platforms could offer a rich catalog of bots providing diverse automated test case generation capabilities under different pricing plans. Several challenges stay in the way though. In this regard, our work paves the way for new promising research directions to overcome some of the problems identified, including the need for automated debugging mechanisms, human-in-the-loop models, and optimal test execution scheduling strategies.

ACKNOWLEDGMENTS

This work has been partially supported by the European Commission (FEDER) and Junta de Andalucía under projects APOLO (US-1264651) and EKIPMENT-PLUS (P18-FR-2895), by the Spanish Government (FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación) under project HORATIO (RTI2018-101204-B-C21), by MCIN/AEI/10.13039/501100011033/FEDER, UE under project BUBO (PID2021-126227NB-C22), by the Excellence Network SEBASENet 2.0 (RED2018-102472-T), by the FPU scholarship program, granted by the Spanish Ministry of Education and Vocational Training (FPU17/04077), and by a Fulbright Spain Predoctoral Research grant at the University of California, Berkeley. We would also like to thank Juan Domingo Martín Gómez for his technical support for the analysis of the data and for insightful discussions regarding the proposed fault clustering approach. [Alberto speaking:] Thanks, dad.

REFERENCES

[1] [n.d.]. Allure test reporting framework. http://allure.qatools.ru. Accessed: March 2022.
[2] [n.d.]. Amadeus Hotel Search API. https://developers.amadeus.com/self-service/category/hotel/api-doc/hotel-search/api-reference/v/2.1. Accessed: March 2022.
[3] [n.d.]. api.data.gov. https://api.data.gov. Accessed: March 2022.
[4] [n.d.]. APIs.guru. https://apis.guru. Accessed: March 2022.
[5] [n.d.]. Bug in Amadeus API – Response contains invalid phone number. http://restest.us.es/fse2022/tf/target/allure-reports/amadeus__r_art_custom/#behaviors/b1a8273437954620fa374b796ffaacdd/26e68cd72fea6346/. Accessed: March 2022.
[6] [n.d.]. Bug in Amadeus API – Response lacks required fax property. http://restest.us.es/fse2022/tf/target/allure-reports/amadeus__r_art_custom/#behaviors/b1a8273437954620fa374b796ffaacdd/8fe8e0421e364228/. Accessed: March 2022.
[7] [n.d.]. Bug in DHL API – 400 status code with message "Threat Detected". http://restest.us.es/fse2022/tf/target/allure-reports/dhl_locationFindByAddress__r_ft_/#behaviors/b1a8273437954620fa374b796ffaacdd/cfb50a0a6efc15e9/. Accessed: March 2022.
[8] [n.d.]. Bug in DHL API – 400 status code with supported country code 'KV' (Kosovo). http://restest.us.es/fse2022/tf/target/allure-reports/dhl_locationFindByAddress__r_rt_custom/#behaviors/b1a8273437954620fa374b796ffaacdd/2cf5fccaf1d0ac1a/. Accessed: March 2022.
[9] [n.d.]. Bug in DHL API – 500 status code when using a negative number for the limit parameter. http://restest.us.es/fse2022/tf/target/allure-reports/dhl_locationFindByAddress__r_rt_perturbation/#behaviors/b1a8273437954620fa374b796ffaacdd/a2c251ace36b16b0/. Accessed: March 2022.
[10] [n.d.]. Bug in FDIC API – 500 status code with valid request. http://restest.us.es/fse2022/tf/target/allure-reports/fdic__r_art_custom/#behaviors/b1a8273437954620fa374b796ffaacdd/f8ef9837ff94e078/. Accessed: March 2022.
[11] [n.d.]. Bug in Marvel API – 500 status code when using an out-of-range value for an enum parameter. http://restest.us.es/fse2022/tf/target/allure-reports/marvel__r_rt_perturbation/#behaviors/b1a8273437954620fa374b796ffaacdd/d18f37597689cbd9/. Accessed: March 2022.
[12] [n.d.]. Bug in Ohsome API – 500 status code with valid request. http://restest.us.es/fse2022/tf/target/allure-reports/ohsome__r_cbt_custom/#categories/882a17760bce9dccc99edbd9f7e59f35/ecb286a3c5503006/. Accessed: March 2022.
[13] [n.d.]. Bug in OMDb API – 500 status code showing stack trace of thrown exception. http://restest.us.es/fse2022/tf/target/allure-reports/omdb__r_ft_/#behaviors/b1a8273437954620fa374b796ffaacdd/68a2df1b2b5866af/. Accessed: March 2022.
[14] [n.d.]. Bug in OMDb API – JSON response contains non-escaped '"' character. http://restest.us.es/fse2022/tf/target/allure-reports/omdb__r_cbt_stateful/

#behaviors/b1a8273437954620fa374b796ffaacdd/cb2ee5c85b6bed2c/. Accessed: March 2022.
[15] [n.d.]. Bug in RESTCountries API – Inconsistency between status code and error message. http://restest.us.es/fse2022/tf/target/allure-reports/restcountries__r_rt_perturbation/#behaviors/b1a8273437954620fa374b796ffaacdd/5b8c51678fa58536/. Accessed: March 2022.
[16] [n.d.]. Bug in Spotify API – 502 status code when unfollowing playlists quickly. https://community.spotify.com/t5/Spotify-for-Developers/Web-API-issue-502-Server-Error-when-unfollowing-playlists/td-p/5346160. Accessed: March 2022.
[17] [n.d.]. Bug in Yelp API – Value of price property ('€') not found in enum possible values ('$', '$$', '$$$', '$$$$'). http://restest.us.es/fse2022/tf/target/allure-reports/yelp_Businesses__r_cbt_custom/#behaviors/b1a8273437954620fa374b796ffaacdd/d50ee751a281f9a3/. Accessed: March 2022.
[18] [n.d.]. Bug in YouTube comments API – 200 status code with invalid request. http://restest.us.es/fse2022/tf/target/allure-reports/youTube_CommentsAndThreads__rw_cbt_perturbation/#behaviors/b1a8273437954620fa374b796ffaacdd/bbb974de70976367/. Accessed: March 2022.
[19] [n.d.]. Bug in YouTube search API – 400 status code with valid request. http://restest.us.es/fse2022/tf/target/allure-reports/youTube_Search__r_cbt_custom/#behaviors/b1a8273437954620fa374b796ffaacdd/25f179afd3f2f309/. Accessed: March 2022.
[20] [n.d.]. Bug report in YouTube comments API – [GET /comments] Request with parameters id and maxResults returns successful response, although they are mutually exclusive. https://issuetracker.google.com/issues/220795144. Accessed: March 2022.
[21] [n.d.]. Bug report in YouTube search API – [GET /search] Valid request obtains 400 status code. Undocumented restriction: If location is used, then channelType cannot be used. https://issuetracker.google.com/issues/220859560. Accessed: March 2022.
[22] [n.d.]. Datadog. https://www.datadoghq.com. Accessed: March 2022.
[23] [n.d.]. DHL Location Finder API. https://developer.dhl.com/api-reference/location-finder#reference-docs-section. Accessed: March 2022.
[24] [n.d.]. FDIC API. https://banks.data.fdic.gov/docs/#api_endpoints. Accessed: March 2022.
[25] [n.d.]. Foursquare categories. https://developer.foursquare.com/docs/categories. Accessed: March 2022.
[26] [n.d.]. Foursquare Venus Search API. https://developer.foursquare.com/reference/v2-venues-search. Accessed: March 2022.
[27] [n.d.]. LanguageTool API. https://languagetool.org/http-api/#/default. Accessed: March 2022.
[28] [n.d.]. Marvel API. https://developer.marvel.com/docs. Accessed: March 2022.
[29] [n.d.]. Microsoft Graph REST API. https://docs.microsoft.com/en-us/graph/api/overview. Accessed: March 2022.
[30] [n.d.]. Netflix Conductor. https://netflix.github.io/conductor/apispec. Accessed: March 2022.
[31] [n.d.]. Ohsome API. https://api.ohsome.org/v1/swagger-ui.html. Accessed: March 2022.
[32] [n.d.]. OMDb API. http://www.omdbapi.com. Accessed: March 2022.
[33] [n.d.]. Online test reports generated. http://restest.us.es/fse2022/showcase. Accessed: March 2022.
[34] [n.d.]. OpenAPI Specification. https://www.openapis.org. Accessed: January 2022.
[35] [n.d.]. OpenAPI specification coverage. https://github.com/meetmatt/open-api-coverage. Accessed: March 2022.
[36] [n.d.]. Postman. https://www.postman.com. Accessed: March 2022.
[37] [n.d.]. ProgrammableWeb API Directory. http://www.programmableweb.com. Accessed: March 2022.
[38] [n.d.]. RapidAPI Hub. https://rapidapi.com/products/hub. Accessed: March 2022.
[39] [n.d.]. RapidAPI Testing. https://rapidapi.com/products/api-testing. Accessed: March 2022.
[40] [n.d.]. RapidAPI Testing - Pricing. https://rapidapi.com/products/api-testing/#pricing. Accessed: March 2022.
[41] [n.d.]. REST Assured. http://rest-assured.io. Accessed: March 2022.
[42] [n.d.]. RESTCountries API. https://restcountries.com. Accessed: March 2022.
[43] [n.d.]. Sauce Labs. https://saucelabs.com. Accessed: March 2022.
[44] [n.d.]. Spotify Web API. https://developer.spotify.com/documentation/web-api/reference. Accessed: March 2022.
[45] [n.d.]. Stripe Products API. https://stripe.com/docs/api/products. Accessed: March 2022.
[46] [n.d.]. Stripe tax codes. https://stripe.com/docs/tax/tax-codes. Accessed: March 2022.
[47] [n.d.]. Wikipedia, the free encyclopedia. https://wikipedia.org. Accessed: March 2022.
[48] [n.d.]. Yelp Businesses Search API. https://www.yelp.com/developers/documentation/v3/business_search. Accessed: March 2022.
[49] [n.d.]. YouTube Comments API. https://developers.google.com/youtube/v3/docs/comments. Accessed: March 2022.
[50] [n.d.]. YouTube Data API. https://developers.google.com/youtube/v3/docs. Accessed: March 2022.
[51] [n.d.]. YouTube Search API. https://developers.google.com/youtube/v3/docs/search/list. Accessed: March 2022.
[52] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Ralf Laemmel, Erik Meijer, et al. 2020. WES: Agent-Based User Interaction Simulation on Real Infrastructure. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 276–284.
[53] Khaled Alnawasreh, Patrizio Pelliccione, Zhenxiao Hao, Mårten Rånge, and Antonia Bertolino. 2017. Online Robustness Testing of Distributed Embedded Systems: An Industrial Approach. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 133–142.
[54] Juan C. Alonso, Alberto Martin-Lopez, Sergio Segura, Jose Maria Garcia, and Antonio Ruiz-Cortes. 2022. ARTE: Automated Generation of Realistic Test Inputs for Web APIs. *IEEE Transactions on Software Engineering* (2022).
[55] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM TOSEM* 28, 1 (2019), 1–37.
[56] Andrea Arcuri. 2021. Automated Blackbox and Whitebox Testing of RESTful APIs With EvoMaster. *IEEE Software* (2021).
[57] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *International Conference on Software Engineering*. 748–758.
[58] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking Security Properties of Cloud Services REST APIs. In *International Conference on Software Testing, Verification and Validation*.
[59] Xiaoying Bai, Muyang Li, Xiaofei Huang, Wei-Tek Tsai, and Jerry Gao. 2013. Vee@Cloud: The Virtual Test Lab on the Cloud. In *2013 8th International Workshop on Automation of Software Test (AST)*. 15–18.
[60] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering (FOSE'07)*. 85–103.
[61] Antonia Bertolino, Guglielmo De Angelis, Lars Frantzen, and Andrea Polini. 2007. The Plastic Framework and Tools for Testing Service-Oriented Applications. In *Software Engineering: International Summer Schools, ISSSE 2006-2008*. 106–139.
[62] Antonia Bertolino, Guglielmo De Angelis, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. DevOpRET: Continuous Reliability Testing in DevOps. *Journal of Software: Evolution and Process* (2020), In press.
[63] Antonia Bertolino, Pietro Braione, Guglielmo De Angelis, Luca Gazzola, Fitsum Kifetew, Leonardo Mariani, Matteo Orrù, Mauro Pezze, Roberto Pietrantuono, Stefano Russo, et al. 2021. A Survey of Field-based Testing Techniques. *ACM Computing Surveys (CSUR)* 54, 5 (2021), 1–39.
[64] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia - A Crystallization Point for the Web of Data. *Journal of Web Semantics* 7, 3 (2009), 154–165.
[65] Tien-Dung Cao, Patrick Félix, Richard Castanet, and Ismail Berrada. 2010. Online Testing Framework for Web Services. In *2010 Third International Conference on Software Testing, Verification and Validation*. 363–372.
[66] Mariano Ceccato, Davide Corradini, Luca Gazzola, Fitsum Meshesha Kifetew, Leonardo Mariani, Matteo Orru, and Paolo Tonella. 2020. A Framework for In-Vivo Testing of Mobile Applications. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 286–296.
[67] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. 2007. A Metamorphic Testing Approach for Online Testing of Service-Oriented Software Applications. *International Journal of Web Services Research (IJWSR)* 4, 2 (2007), 61–81.
[68] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Empirical Comparison of Black-Box Test Case Generation Tools for RESTful APIs. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 226–236.
[69] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Restats: A Test Coverage Tool for RESTful APIs. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 594–598.
[70] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2022. Automated Black-Box Testing of Nominal and Error Scenarios in RESTful APIs. *Software Testing, Verification and Reliability* (2022).
[71] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. 2019. A Snowballing Literature Study on Test Amplification. *Journal of Systems and Software* 157 (2019), 110398.
[72] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *International Enterprise Distributed Object Computing Conference*. 181–190.
[73] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. 2002. Test Case Prioritization: A Family of Empirical Studies. *IEEE transactions on software*

engineering 28, 2 (2002), 159–182.

[74] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Ph. D. Dissertation.

[75] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. 2019. Automating SLA-Driven API Development with SLA4OAI. In *International Conference on Service-Oriented Computing*. 20–35.

[76] Antonio Gamez-Diaz, Pablo Fernandez, Antonio Ruiz-Cortés, Pedro J Molina, Nikhil Kolekar, Prithpal Bhogill, Madhurranjan Mohaan, and Francisco Méndez. 2019. The Role of Limitations and SLAs in the API Industry. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1006–1014.

[77] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *Commun. ACM* 63, 2 (2020), 70–76.

[78] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API Data Fuzzing. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 725–736.

[79] Arnaud Gotlieb. 2015. Constraint-Based Testing: An Emerging Trend in Software Testing. In *Advances in Computers*. Vol. 99. Elsevier, 67–101.

[80] Michaela Greiler, Hans-Gerhard Gross, and Arie van Deursen. 2010. Evaluation of Online Testing for Services: A Case Study. In *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*. 36–42.

[81] Joachim Hänsel and Holger Giese. 2017. Towards Collective Online and Offline Testing for Dynamic Software Product Line Systems. In *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*. 9–12.

[82] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. 2019. A Survey on Adaptive Random Testing. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2052–2083.

[83] Mohammad S Islam, William Pourmajidi, Lei Zhang, John Steinbacher, Tony Erwin, and Andriy Miranskyy. 2021. Anomaly Detection in a Large-scale Cloud Platform. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 150–159.

[84] Daniel Jacobson, Greg Brail, and Dan Woods. 2011. *APIs: A Strategy Guide*. O'Reilly Media, Inc.

[85] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs. In *International Conference on Software Testing, Verification and Validation*.

[86] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A Black Box Tool for Robustness Testing of REST Services. *IEEE Access* 9 (2021), 24738–24754.

[87] 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 *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 231–241.

[88] Alberto Martin-Lopez, Sergio Segura, Carlos Müller, and Antonio Ruiz-Cortés. 2021. Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. *IEEE Transactions on Services Computing* (2021). Article in press.

[89] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. Test Coverage Criteria for RESTful Web APIs. In *A-TEST*. 15–21.

[90] 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*. 459–475.

[91] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: Automated Black-Box Testing of RESTful Web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21)*.

[92] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2022. [Supplementary material] Online Testing of RESTful APIs: Promises and Challenges. https://doi.org/10.5281/zenodo.6676117.

[93] 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 *International Workshop on Testing for Deep Learning and Deep Learning for Testing*.

[94] Marc Oriol, Xavier Franch, and Jordi Marco. 2015. Monitoring the Service-based System Lifecycle with SALMon. *Expert Systems with Applications* 42, 19 (2015), 6507–6521.

[95] Leonard Richardson, Mike Amundsen, and Sam Ruby. 2013. *RESTful Web APIs*. O'Reilly Media, Inc.

[96] Omur Sahin and Bahriye Akay. 2021. A Discrete Dynamic Artificial Bee Colony with Hyper-Scout for RESTful web service API test suite generation. *Applied Soft Computing* 104 (2021), 107246.

[97] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic Testing of RESTful Web APIs. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1083–1099.

[98] Burr Settles. 2012. Active Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 18 (2012), 1 – 111.

[99] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2021. Improving Test Case Generation for REST APIs Through Hierarchical Clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 117–128.

[100] Margus Veanes, Pritam Roy, and Colin Campbell. 2006. Online Testing with Reinforcement Learning. In *Formal Approaches to Software Testing and Runtime Verification*. 240–253.

[101] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *ACM/IEEE 44th International Conference on Software Engineering*.

[102] Li Yujian and Liu Bo. 2007. A Normalized Levenshtein Distance Metric. *IEEE transactions on pattern analysis and machine intelligence* 29, 6 (2007), 1091–1095.

[103] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.