

Introducing Black-Box Fuzz Testing for REST APIs in Industry: Challenges and Solutions

Andrea Arcuri
Kristiania and OsloMet
Oslo, Norway
andrea.arcuri@kristiania.no

Alexander Poth
Volkswagen AG
Wolfsburg, Germany
alexander.poth@volkswagen.de

Olsi Rrjolli
Volkswagen AG
Wolfsburg, Germany
olsi.rrjolli@volkswagen.de

Abstract—REST APIs are widely used in industry, in all different kinds of domains. An example is Volkswagen AG, a German automobile manufacturer. Established testing approaches for REST APIs are time consuming, and require expertise from professional test engineers. Due to its cost and importance, in the scientific literature several approaches have been proposed to automatically test REST APIs. The open-source, search-based fuzzer EVOMASTER is one of such tools proposed in the academic literature. However, how academic prototypes can be integrated in industry and have real impact to software engineering practice requires more investigation. In this paper, we report on our experience in using EVOMASTER at Volkswagen. We share our learnt lessons, and identify real-world research challenges that need to be solved.

Index Terms—SBST, REST, API, black-box, industry, fuzzing

I. INTRODUCTION

REST APIs are used everywhere, to provide all different kinds of data and functionalities over a network (e.g., internet) [1], [2]. They are also common when developing backend applications, particularly when using microservice architectures [3], [4]. Nowadays, when interacting with a web page or a mobile app, often one or more REST APIs are involved. Therefore, the validation and verification of this type of web service is of paramount importance.

Volkswagen AG is a German automobile manufacturer.¹ As for many enterprises, its IT services rely on REST APIs. Due to the high cost of thorough testing from professional test engineers, significant effort has been spent to modernize their processes, and leverage what novel techniques and research outputs can provide in this context. In particular, the use of novel Artificial Intelligence (AI) techniques seems promising. To enhance the quality of their testing processes and reduce cost, different AI techniques available to the public, like based on LLM (e.g., StarCoder [5]) and Evolutionary Computation (e.g., EVOMASTER [6]), have already been evaluated at Volkswagen [7], with some initial success.

In the scientific literature, in the last few years there has been a lot of work on test automation for REST APIs [8]. “Fuzz testing” [9]–[11] (also known as “fuzzing”) is a term used to refer to the automated generation of test cases, typically with random or unexpected inputs, to find crashes and

security issues in the tested applications. Several techniques can be used to improve performance (e.g., to cover more parts of the code of the tested application), e.g., based on AI techniques. In the literature, several tools (i.e., fuzzers) have been proposed, like the aforementioned EVOMASTER. Most of these tools are open-source, like for example Restler [12] and RestTestGen [13]. Any enterprise in the world can download and try out those tools on their REST APIs.

Usually, though, in the scientific literature these tools have been evaluated only in the “lab”. Researchers might design and develop some novel techniques, implement them in a tool, and then carry out experiments on some APIs to evaluate the effectiveness (or lack thereof) of their novel techniques. Real-world APIs might be used for these experiments, but usually no engineers or QA specialist in industry would be involved in using and evaluating those tools. In other words, no “human aspects” of introducing fuzzing techniques in industry [14] has been studied so far in literature of testing REST APIs [8].

To fill this important gap in the scientific literature, the authors of EVOMASTER were eager to start working with the test engineers at Volkswagen AG. The first interaction happened in October 2023 when the test engineers at Volkswagen contacted the maintainers of EVOMASTER with questions about more advanced use cases.

This is what started the “open exchange” between the authors of EVOMASTER and the test engineers at Volkswagen AG, in particular with the Quality innovation Network (QiNET) of the Group IT, which focus on innovations about IT quality management and engineering [15], [16]. In the literature, there are many types of academia-industry collaborations [17]–[20]. Bridging the gap between academia and industry is an important research endeavour, that can provide benefits for both parties. As such, this is explicitly mentioned in the documentation of EVOMASTER regarding how people can contribute.²

In this paper, we report on the first year of this exchange between the developer team of EVOMASTER and QiNET at Volkswagen AG. The Volkswagen engineers evaluated EVOMASTER in an industrial setup which opened additional usage scenarios for the EVOMASTER team. We discuss all the technical details and features that have been implemented

¹<https://www.volkswagen.de>

²<https://github.com/WebFuzzing/EvoMaster/blob/master/docs/contribute.md>

to be able to better integrate an academic fuzzer such as EVOMASTER into the testing practices of a large enterprise such as Volkswagen AG [21]. We discuss all the challenges that we have overcome, and highlight several challenges that still need addressing. Many of these challenges are not specific to EVOMASTER, but would likely be relevant to all other kinds of fuzzers and test generators whose authors want to have an impact on industrial practice. Note: the work in [7] evaluated different openly available test generators, including EVOMASTER, “as they are”, before an exchange was started. In other words, this work can be considered a follow up of [7], discussing what has been identified and solved during one year of academia-industry exchange.

In particular, in this industry-report we aim to shed light on these important research questions:

RQ1: Why choosing EVOMASTER for an academia-industry evaluation and exchange instead of other tools?

RQ2: What features were needed to integrate a fuzzer like EVOMASTER in industrial testing processes such as for example at Volkswagen AG?

RQ3: What is missing from the generated tests of EVOMASTER compared to the existing manually written test suites?

RQ4: What are the current major challenges and most needed features?

II. RELATED WORK

In the last few years, due to its importance in industry, lot of research has been carried out on automated testing of REST APIs [8]. Several tools have been presented in the literature, such as for example (in alphabetic order): ARAT-RL [22], bBOXRT [23], CATS [24], DeepREST [25], Dredd [26], Fuzz-lightyear [27], Morest [28], ResTest [29], RestCT [30], Restler [12], RestTestGen [13] Schemathesis [31], and Tcases [32]. However, no industry report or empirical study with human subjects has been provided in the literature yet [8]. This paper is a first step to fill this important gap in the research literature.

In the literature of software testing, there have been several studies involving the evaluation of academic techniques in industry. These included topics such as unit testing [33], testing of embedded systems [34], user interface testing for ERP applications [35] and for mobile applications [36], [37], mutation testing [38], [39], fault debugging [40] and flaky tests [41]. This work extends such current body of knowledge by addressing the important topic of fuzzing REST APIs in industry.

EVOMASTER is a search-based fuzzer, open-source on GitHub since 2016 [42]. It is a mature tool [6], [43]–[45], originally designed for search-based white-box testing of REST APIs [46]. However, it has been extended to support black-box testing [47], as well as other types of web services such as GraphQL [48] and RPC [49]. For white-box testing, it uses several advanced search-based techniques, such as hyper-mutation [50], testability transformations [51], [52], and support for effectively handling the APIs’ environment

such as SQL databases [53] and external web services [54]. EVOMASTER has been among the best performing tools in empirical comparisons [55]–[57].

III. RQ1: FIRST STEPS

In 2022, at the IT Test & Quality Assurance (TQA) a process was started to check and evaluate possible Artificial Intelligence (AI) techniques to enhance their testing activities in the Group IT. Discovering EVOMASTER was part of such activities. The tool came out when querying internet search engines (e.g., Google), using terms such as “AI” and “testing”. Few tools were found this way. The choice of trying out EVOMASTER was based on some specific properties:

- the tool or component should be fast to integrate and simple to scale in a cloud environment, like for example open-source solutions do. EVOMASTER is open-source since its first commit in 2016, with LGPL licence. However, enterprises might prefer more permissive licenses, such as MIT and Apache, especially if they want to make modifications and extensions to such tools for internal use. The choice of an open-source license for a research prototype is not trivial, as discussed for example in [44].
- the tool should be “AI-hype compatible”. In other words, it should explicitly states it uses AI techniques. White-box EVOMASTER is based on Evolutionary Computation, as explicitly stated in its documentation.
- the tool should be actively maintained, ideally with at least three active developers, and should had been around for some years. This provides more trust for its maintenance in the future. As of 2022, EVOMASTER was already available open-source for 6 years, with several active code contributors. As of 2024, there are at least 10 code contributors with 50 or more Git commits to EVOMASTER.³
- the outputs of the tool should be easily integrated in the current testing environments. In the case of Volkswagen AG, this means being able to run the generated test suites with Java and JMeter. As EVOMASTER can output tests in JUnit format, this was not a problem.
- the tool should have given good benchmark results. Engineers at Volkswagen AG checked some academic papers to see how different tools fare with each others.

EVOMASTER was the tool that matched all these criteria best. There are some other tools that “look interesting” for the engineers at Volkswagen, such as Restler [12], Schemathesis [31] and CATS [24]. However, they did not have the “AI” label. As such, EVOMASTER was preferred.

RQ1: *the maturity and health of a product or project are key elements to decide whether to invest time to try it out.*

To increase adoption in industry, it is recommended to explicitly provide such information in the documentation of the tools. Specifying which techniques a tool employs can be a major deciding factor, especially when “hype” and industry trends are involved (e.g., related to AI).

³<https://github.com/WebFuzzing/EvoMaster/graphs/contributors>

IV. RQ2: IMPLEMENTED FEATURES

In this section, we discuss all the major features needed by the test engineers at Volkswagen AG to enhance the usability and effectiveness of EVOMASTER on the testing of their APIs.

Domain Expert Inputs: Examples. Fuzzers have limitations, especially black-box ones. The most important needed feature was a way for the test engineers to provide help to the fuzzer to obtain better results. Ideally, a fuzzer should be fully automated. But, as their outputs are still not as good as manually developed test suites [7], any help that can be exploited could be useful. But, this is only as long as it does not take too much time to provide and prepare such input data and verify the outcomes. For example, this could be something as simple as providing valid ids of existing users in the database.

For each parameter of each endpoint, there was a need to provide a “dictionary” of meaningful data, based on the actual content of the database in the test environment at Volkswagen. From a technical perspective, this is nothing too complex. However, one issue here was to decide on which way and format such test data should be provided as input to EVOMASTER. Creating a custom format, e.g., a custom configuration file with the test data, was a possibility. However, it is not ideal for two main reasons: (1) out of the box, IDEs would not support such a new format (e.g., for auto-complete features and validation); (2) a custom format would not be supported by other fuzzers, and so any effort spent there would have to be repeated when using a different fuzzer. If there was no other option, a custom format would be a necessity. However, one alternative is to provide such test data directly in the OpenAPI schemas. The OpenAPI schemas provide fields called `example` and `examples` to specify possible examples of how parameters could look like. These entries can be used to provide the needed test data.

At that time, EVOMASTER did not have any heuristic to exploit `example` data specified in the OpenAPI schemas. This is implemented now since version 3.0.0 [58]. Given a certain probability, input is not generated at random, but taken from the provided dictionary (i.e., the values declared in the `example` entries), if any is specified. A further benefit here is that any other tool that can exploit `examples` entries would be able to directly use such test data. For example, another fuzzer that explicitly states that it supports `example` entries is RestTestGen [59].

Domain Expert Inputs: Links. A further issue in testing REST APIs is to identify dependencies among endpoint operations. For example, the needed input for an endpoint *Y* might require to come from the output of an endpoint *X*. In the literature, to improve performance several techniques have been designed to try to infer possible dependencies among operations, like the Operation Dependency Graph used in RestTestGen [59], or our own heuristics based on template test actions [60].

As for any heuristics, there is no guarantee that they would be helpful on all different types of schemas and styles in

```
paths:
  "/api/links/create":
    post:
      tags:
        - bb-links-application
      operationId: postCreate
      responses:
        '200':
          description: OK
          content:
            "*/*":
              schema:
                "$ref": "#/components/schemas/
                BBLinksDto"
      links:
        LinkToGetUser:
          operationId: getUser
          parameters:
            path.name: "$response.body#/data/id"
            query.name: BAR
            code: "$response.body#/data/code"
  "/api/links/users/{name}/{code}":
    get:
      tags:
        - bb-links-application
      operationId: getUser
      parameters:
        - name: name
          in: path
          required: true
          schema:
            type: string
        - name: name
          in: query
          required: false
          schema:
            type: string
        - name: code
          in: path
          required: true
          schema:
            type: integer
            format: int32
```

Fig. 1. Extract from an OpenAPI schema of an artificial API example, with a `links` definition.

which API are designed. However, as of version 3.0, OpenAPI schemas can now support the definition of “links”.⁴ Links allows to specify that any kind of output from an endpoint could be used as a meaningful input to another endpoint. Figure 1 shows a simple, artificial example of a link definition. This example is one of the end-to-end tests used in EVOMASTER to verify its functionalities [44] (in particular, the end-to-end test called BBLinksEMTest). In a link definition, several values could be defined, which can be constant or be any field or value from the request/response of the linked endpoints.

If a fuzzer can exploit such link definitions, then test engineers have incentive to spend time adding those links to the schemas of the APIs they are testing. Furthermore, regardless of testing concerns, adding links improve the quality of the schema and therefore as well the quality of the documentation of the API. However, supporting OpenAPI links has been far from trivial. Not only there are several ways to define

⁴<https://swagger.io/docs/specification/links/>

```

@Test(timeout = 60000)
public void test_2() throws Exception {

    ValidatableResponse res_0 = given().accept("*/*")
        .post(baseUrlOfSut + "/api/links/create")
        .then()
        .statusCode(200);
    String link_0__data_id = res_0.extract().body().
        path("data.id").toString();
    String link_0__data_code = res_0.extract().body().
        path("data.code").toString();

    given().accept("*/*")
        .get(baseUrlOfSut + "/api/links/users/" +
            link_0__data_id + "/" + link_0__data_code + "?
            name=BAR")
        .then()
        .statusCode(200);
}

```

Fig. 2. Example of generated test showing the dynamic use of a link, based on the OpenAPI schema defined in Figure 1.

links, but also more importantly the dynamic nature of link handling must be maintained in the generated tests. Otherwise, it would not be useful, as likely ending up producing flaky tests. Figure 2 shows an example of generated test on an API based on the schema defined in Figure 1. As we can see in that generated test, two values are dynamically extracted from the first HTTP request and used as input in the second request.

We do not know if any other fuzzers support OpenAPI links. Still, what is interesting here is that the use of fuzz testing can impact how schemas are extended and updated by practitioners.

Schema Validation. This feature was strongly related to the handling of `examples` and `links` entries in OpenAPI schemas. When we implemented the support for `examples` and `links` in EVOMASTER, the first feedback from an industrial perspective showed that it was not enough, as those entries were not used during the search. First suspect was a fault in these new functionalities, which would not be unexpected. However, it turned out that the used schemas were invalid.

To better explain this issue, consider Figure 3 that shows a faulty variant of the OpenAPI schema defined in Figure 1. Can the reader spot the problem? That schema is a syntactically valid YAML file, but not an extract of a valid OpenAPI schema. Here, the entry `links` at Line 14 is invalid, because defined under the object `responses` and not under `'200'`. Without any check or warning, such `links` definition would be ignored.

If a schema is written manually with just the support of a text editor or an IDE, then this type of issues can be easily missed. Editors that have custom support (e.g., via plugins) for OpenAPI are necessary to avoid this kind of problems. Still, as we found out, even a valid OpenAPI schema can be problematic for a fuzzer. For example, the type of the entry `example` in a Schema Object definition is `Any`.⁵ This means

⁵<https://swagger.io/specification/#schema-object>

```

paths:
  "/api/links/create":
    post:
      tags:
        - bb-links-application
      operationId: postCreate
      responses:
        '200':
          description: OK
          content:
            "*/*":
              schema:
                "$ref": "#/components/schemas/
                BBLinksDto"
          links:
            LinkToGetUser:
              operationId: getUser
              parameters:
                path.name: "$response.body#/data/id"
                query.name: BAR
                code: "$response.body#/data/code"

```

Fig. 3. Faulty definition of endpoint, based on schema from Figure 1.

that inserting wrong types for an example (e.g., not matching the type of field the example is for, like an array of strings for an integer value) could be silently ignored by an editor/IDE, even if it has support for OpenAPI validation.

To handle this issue, EVOMASTER was extended to validate the input OpenAPI schemas. If there is any issue, we make sure to print meaningful warning messages to the users, inviting them to fix those issues to enable a more performant fuzzing experience. Even if a schema has issue, EVOMASTER would not crash, but rather it will try its “best-effort” to fuzz the target API, exploiting all the information it can have access to.

Authentication. In most cases, real-world APIs require some sort of authentication to identify the user that is making the requests. To be able to effectively fuzz an API, authentication information for some existing users must be provided. There can be several different types of authentication mechanisms, typically based on some sort of userid and password provided by the user.

Among the different types of authentication mechanisms, the two most commons that we have experienced in practice are:

Static. A secret is sent at each HTTP request, either in a HTTP header, query or path parameter. The secret could be the combination of userid/password, or a random string uniquely associated to the user.

Dynamic. A secret, typically userid and password, is sent to a “login” endpoint. If the secret is valid, then such endpoint will return a body payload with an authentication “token”. Such “token” can then be used as a secret like in the “Static” case, i.e., can be sent in a header, query or path parameter. The difference here is that the “token” might be short-lived, i.e., only valid for some minutes or hours.

Supporting authentication in fuzzing is not a trivial task. That can be a reason why few fuzzers seem to have only

```
[[auth]]
name="logintoken"
[auth.loginEndpointAuth]
endpoint="/api/logintoken/login"
payloadRaw= ""
{"userId": "foo", "password":"123"}
""
verb="POST"
contentType="application/json"
[auth.loginEndpointAuth.token]
headerPrefix="Bearer "
extractFromField = "/token/authToken"
httpHeaderName="Authorization"
```

Fig. 4. Example of TOML configuration file for authentication using tokens extracted from a login endpoint, and that can then be sent as Bearer in the HTTP Authorization header.

limited or no support at all for it [56].

Black-box EVOMASTER already supported the “Static” case, with for example options such as `--header` to specify static HTTP headers sent on each request. However, handling the “Dynamic” case was not supported for black-box testing. At the beginning, engineers at Volkswagen had to make “manual” calls to the login endpoints, and then use the result with `--header` option when starting EVOMASTER. This could be very frustrating, especially in cases in which API tokens only had a five minute lifespan.

Supporting the “Dynamic” case is more complicated. There is the need to specify several options, like which endpoint to call, how to extract the auth information, and how then to use it in all the following requests. And this has to be done dynamically, e.g., the generated JUnit test suite files will still then need to be able to do the same. Once all this information is provided, then it is up to the tool (i.e., EVOMASTER in this case), to use such information to collect the needed authentication tokens, and use them during the fuzzing.

This latter part was already implemented in EVOMASTER, for the support of white-box testing. There, authentication information can be specified in the so called EVOMASTER *driver* classes [43]. Those are classes written in Java or Kotlin, where the user can specify several options, especially how to start, stop and reset the API. Among those options, there is also the possibility to specify authentication information. Several utilities are provided to simplify such task.

Doing the same for black-box testing is not straightforward, as there is no Java/Kotlin driver class. Providing all needed info on the commandline (e.g., with parameters such as `--header`) did not look like a good idea, as there would be many parameters to set. In the end, from a usability perspective, the choice was made to define *configuration files* to specify authentication information. These files can be written in either YAML or TOML format. We explicitly decided to do not support the JSON format, as that format is in our opinion not sufficient for configuration files, as it does not support writing comments in it.

Once written, these auth configuration files can then be given as input to EVOMASTER when it starts the fuzzing.

```
@Test(timeout = 60000)
public void test_0() throws Exception {

    final String token_logintoken = "Bearer " +
        given()
            .contentType("application/json")
            .body(" { " +
                " \"userId\": \"foo\", " +
                " \"password\": \"123\" " +
                " } ")
            .post(baseUrlOfSut + "/api/logintoken/login")
            .then().extract().response()
                .path("token.authToken");

    given().accept("*/*")
        .header("Authorization", token_logintoken)
        .get(baseUrlOfSut + "/api/logintoken/check")
        .then()
        .statusCode(200)
        .assertThat()
        .contentType("text/plain")
        .body(containsString("OK"));
}
```

Fig. 5. Example of generated test showing how authentication tokens can be dynamically retrieved and used in following HTTP calls.

Figure 4 shows an example of such a configuration file on an artificial API, used as end-to-end test in EVOMASTER [44] (in particular, the test `BBAuthTokenEMTest`). Figure 5 shows a generated test in which such info is used to authenticate a user.

Computational Resources. In a large organization, there can be many, many APIs that need testing. And, even for the same API, there can be several different versions that might need testing. Then, this testing process has to be repeated for each new release, for each API. All this requires a non-trivial amount of computational resources. When dealing with testing at scale, how to best to use these computational resources becomes critical.

One of the first questions when using a fuzzer is for how long to run it. Should it be for just 30 seconds? 1 hour? or 48 hours? We have no answer for this. A user would want results as soon as possible. But, a fuzzer would still need some time to be able to produce good results. Also, the time required to properly fuzz an API would likely depend on its size and complexity. In industry, there is the need for a sustainability design perspective [61] to make all components, including a fuzzer such as EVOMASTER, in the scaled setup “ready”.

For white-box testing EVOMASTER, we usually recommend to run it at least 1 hour. This can be set with the parameter `--maxTime`. However, the value “1-hour” is just an educated guess. For black-box testing, it might be more trickier to give a time recommendation. Of course, the longer the better, but it is likely that the fuzzing process could stagnate soon if there is no white-box code heuristic that can be exploited to generate better tests. In other words, one might not see much improvements after 5 or 10 minutes (this of course all depends on the tested API).

To take this feedback and insight into account, we in-

roduced a new parameter `--prematureStop`, where a timeout can be specified. If there is no improvement (i.e., no new testing target is covered) within the timeout, then the fuzzing process is prematurely stopped. For example, using something like `--maxTime 1h` and `--prematureStop 10m` would mean that the fuzzing will run for one hour, but, if no improvement is reached at any point in time within the last 10 minutes, then the fuzzing process is stopped. Doing something like this enables testers to allow EVOMASTER to run for its needed time, but, if for any reason it looks like it gets stuck and does not provide any further improvement, then EVOMASTER can be automatically stopped. Stopping EVOMASTER prematurely enables releasing computational resources that could be used to fuzz other APIs.

Another needed feature related to the handling of computational resources was the ability to fuzz only subsets of an API. In an OpenAPI schema there can be many declared endpoints, sometime in the order of hundreds. There might be the need to test only some of those endpoints (e.g., related to some newly introduced functionalities), or having EVOMASTER run in parallel on several different processes/servers, each one testing in parallel a subset of the API. Supporting such industrial needs was relatively easy, as it was not particularly difficult to provide filtering options to specify which endpoints to test. This can for example be based on the prefix of the endpoint paths (e.g., only use endpoints that start with a `/v3/items*`, by using the option `--endpointPrefix`), or by “tags” specified in the OpenAPI schema (e.g., using the option `--endpointTagFilter` to filter all endpoints that have the specified tags declared in their schema).

Coverage Criteria. Since its inception in 2016, EVOMASTER has been first and foremost a *white-box* search-based fuzzer for APIs running on the JVM. The support for black-box testing was introduced years later, in 2020 [47]. Still, the black-box mode was introduced only for demonstration purposes, as it is much easier to use (e.g., no need to first write any driver class). However, currently Volkswagen is more interested in black-box testing, so it was the right time to improve EVOMASTER’s black-box capabilities in the context of system testing.

The first “shocking” revelation was how few tests were generated by EVOMASTER. During a fuzzing session of one hour, depending on the API, hundreds of thousands of HTTP calls can be made by a fuzzer. For example, a 5ms per call would result in 720 000 calls in one hour. However, no sane fuzzer would output test suites with hundreds of thousands of HTTP calls. What is outputted at the end of the fuzzing process is supposed to be a minimized set, including all the most relevant tests. However, “relevance” here all depends on what is defined as coverage criteria.

Besides fault finding, the coverage criteria for white-box EVOMASTER are based on code metrics. For example, on line and branch coverage. If any new test evolved during the search triggers the execution of a new line in the source code of the tested API, then we make sure such test will be part of the final output test suite. Unfortunately, this is not possible for

black-box testing, as no code metric is collected (if it was, then it would not be black-box). Black-box coverage criteria were based only on combinations of endpoints and returned HTTP status code. In other words, we save each test that can return a status code not seen before on each specific endpoint in the API. Unfortunately, this turned out to be quite limited.

To address such a major limitation, EVOMASTER was extended with the set of black-box coverage criteria defined in [62]. This includes for example considering all boolean combinations of optional query parameters (i.e., on and off), as well as covering all values of input enums. However, we made two major extensions to those coverage criteria defined in [62]: (1) in the coverage targets, we included all `example(s)` and followed `links` declared in the schema, as those are the input data the testers are interested into and want to see in the generated tests; (2) for each coverage target per endpoint, we check when it is covered regardless of the returned status code, and as well as when the returned code is in the 2xx success family.

This latter point requires some explanation. Assume two query parameters for an endpoint `Z`: the string `X` and the enumeration `Y` with 10 possible values. If `X` has some constraints that are not satisfied (e.g., a regex), then the API returns 400 when calling `Z` with such invalid input, regardless of the value of `Y`. Having in the final test suite 10 test cases for each different value of `Y` when the endpoint `Z` anyway returns a 400 due to invalid `X` would be of little interest, as those values of `Y` have no impact on the test. On the other hand, if the endpoint returns a 2xx, then such values of `Y` might cover different execution paths inside the API. Still, it might be possible that an endpoint `Z` is never covered with a 2xx during the search. In those unlucky cases, it might still be worthy to have different values of `Y` in the generated tests.

Databases. Databases play a critical role in system testing. The data contained in them influence what can be tested from the API endpoints. These test environments are often production comparable systems with a dedicated test-data set in their databases, to be able to run and stimulate the tests of the system. To the best of our knowledge, it is a common practice in industry to test APIs in a test environment with such anonymized or synthetic data. This is the case for example for the testing in the context of TQA, and in all the other companies the EVOMASTER team has worked in and collaborated with. Using databases that are copies of production data snapshots is usually not possible, due to privacy and legal issues (e.g., GDPR in Europe).

The data in these databases are a potential source of useful inputs that can be exploited. For example, consider testing a GET operation on the path `/products/{id}`. If there is no operation to create a new product (e.g., this is done by a separated admin API), without knowing any valid `id` entry, it would be difficult to test any scenario besides the trivial 404 (i.e., resource not found). Having testers adding such values as `examples` entries is possible, but ideally such manual interventions should be minimized, if possible.

For white-box testing, this is not a problem, as a scenario

like this can be trivially solved with SQL handling techniques as introduced in [53]. White-box EVOMASTER can analyze all interactions with SQL databases [53], and then use all different kinds of taint analyses techniques [51], [52] to generate input data that make sure each SQL SELECT command done in the API returns data.

To try to exploit existing database information in black-box testing as well, we implemented the same kind of *Response dictionary* technique introduced in [59]. In simple terms, at a high level GET operations on collections (e.g., `GET:/products`) can provide info on what present in the database. From those returned payloads, the ids of the resources can be inferred (with different types of string-matching algorithms and heuristics [59]). Those collected ids at runtime can then be stored in a dictionary, and used afterwards for the endpoints that require ids as input (e.g., `PUT:/products/{id}`).

This approach might work well when testing remote APIs on internet, where login is done with a newly created user account made only for academic experiments. However, when dealing with test environments shared by possibly several testers, and having existing manual test cases relying on some specific data, having a fuzzer starting to delete entry by doing `DELETE:/products/{id}` with admin credentials is unwise. A test environment can still be reset and rolled-back. But, doing that at each fuzzing execution might not be a viable strategy. To address this issue, in our implementation the inferred “Response dictionary” is exclusively used only for read operations (i.e., GET). This is an important problem, which we will go back to in Section VI.

Rate Limiter. When fuzzing an API, the more test cases can be evaluated, the better results can be expected at the end. However, sending as many requests as possible toward an API could put major strain/stress on its operations.

When academics perform empirical experiments on open-source APIs locally on their machines (e.g., from the EMB corpus [63]), this is not a problem. However, when testing an API on the internet (e.g., as done in [59]), this can become problematic. For example, to prevent and mitigate Denial-of-Service (DoS) attacks, the API could stop replying to requests from those fuzzers. A solution here is to have a “rate limiter”, i.e., a way to specify how many requests to send at most given a specified amount of time. For example, in EVOMASTER there is the option `--ratePerMinute`. A value like $N = 30$ would mean that, within a minute, no more than 30 HTTP requests are sent, i.e., at most 1 request every 2 seconds. When a request takes X milliseconds, EVOMASTER will wait $(60000/N) - X$ milliseconds before making a new HTTP request.

Although this parameter was originally introduced in EVOMASTER only for empirical studies, it was a necessity as well for a proof-of-concept study at Volkswagen AG. APIs are tested in a testing environment, mimicking production settings, including DoS prevention/mitigation mechanisms.

TABLE I
STATISTICS OF THE TWO USED INDUSTRIAL APIs IN THE EXPERIMENTS, INCLUDING THEIR NAME, LINES OF CODE (#LOC), NUMBER OF HTTP ENDPOINTS (#ENDPOINTS), AND THE NUMBER OF THE EXISTING ALREADY WRITTEN TESTS (#TESTS).

Name	#LOC	#Endpoints	#Tests
auth-service	5750	3	16
user-service	8591	21	56

RQ2: *the most needed features are related on how users can provide extra guidance to the fuzzing, how to do authentication and how computational resources can be optimized.*

V. RQ3: COMPARISON WITH EXISTING TESTS

Several of enhancements introduced in EVOMASTER were related to its usability at industrial level. Still, ultimately, what is important is how the generated tests can help the testers and QA engineers in their testing tasks. As such, how these generated tests compare with the existing manually written tests is of major importance. The closer in quality they are, the better the tool would be when addressing the testing of a new API for which no test exists yet.

Initial experiments done with older versions of EVOMASTER (e.g., 1.6.1) and with LLM techniques showed that manual tests are still significantly better [7]. In this paper, we repeat such experiments with the latest version of EVOMASTER at the time of writing (i.e., 3.2.1-SNAPSHOT). Furthermore, API schemas were enhanced with extra information, in particular regarding `examples` and `links`, as now these features are supported by EVOMASTER.

The experiments involve two APIs developed at Volkswagen AG, chosen by its test engineers as initial case study for evaluating the potential benefits of automated test generation, as part of the innovation activities of QiNET, which is part of TQA. Table I shows some statistics of these two APIs.

On these APIs, we ran EVOMASTER three times, for a time budget of 10 minutes. The outcome variation was between 55 and 60 test cases for the `user-service`. The outcome was always 8 test cases for the `auth-service`. However, a longer search time does not seem to improve the outcome significantly, as still in the last minutes of the 10 minute search no additional testing target was covered. For each API, we took the “worst case” of the generated test suites, with the minimum number of test-cases out of the three runs, for a conservative evaluation result. Table II shows the results of this manual analysis after verification by the engineers against the handcrafted reference test-suites referenced in Table I.

This comparison was manually done by test-engineers, with more than 10 years of experience, within an industrial setting. In particular, we are interested in checking how many generated tests could be kept (i.e., they are evaluated as “good”), how many need to be modified (e.g., for LLM-based generated tests in previous work [7], most of them needed

TABLE II

EVALUATION OF THE TESTS GENERATED BY EVOMASTER COMPARED TO THE EXISTING MANUALLY WRITTEN TESTS. OUT OF THE *total* NUMBER OF GENERATED TESTS, WE REPORT HOW MANY TESTS ARE *accepted* (I.E., COVERING SAME SCENARIOS AS IN MANUAL TESTS, WITHOUT NEED TO BE MODIFIED), HOW MANY NEEDED TO BE MANUALLY *modified* TO BE USABLE, HOW MANY WERE *removed*, AND HOW MANY TESTED *new* SCENARIOS THAT ARE NOT COVERED IN THE MANUAL TESTS. WE ALSO REPORT HOW MANY TEST SCENARIOS FROM THE HANDCRAFTED TESTS ARE *missing* IN THESE GENERATED TESTS.

Name	total	accepted	modified	removed	new	missing
auth-service	8	8	0	0	0	8
user-service	55	46	0	2	7	8

to be modified to remove hallucinations related to generated assertions on the returned responses), how many need to be removed (i.e., they are redundant or simply considered not useful), and how many are missing to reach the same quality level of the reference implementation (i.e., the already existing manually written tests). Also, we are interested in what was newly discovered and added to the test-suite by EVOMASTER.

As we can see in Table II, many generated tests are useful. Overall, the generated tests save time and effort for the engineers. However, the generated tests are not at a level that they are directly usable as test-suites. They still need validation and enhancement by engineers to reach a sufficient quality level for a usable test-suite. But, the automatically generated test-suites also add tests which would usually not be written by engineers, as too much work for less “value”. Here, the AI-facilitation offers a chance to improve the test-suite quality beyond the existing handcrafted test cases. Notable test-suite enhancements are the typical test set of the lower and upper boundaries and a random data set. The engineers often do not define all potential variants. This results in that the EVOMASTER and a test-engineer “pair” can deliver in less time a better test-suite than only a handcrafted one would be under real-world economical constraints. Still, a major impediment for future improvement is the verification effort of the generated tests. Here, activities are needed on the EVOMASTER test-case generator to make the verification less time and effort consuming, e.g., by improving the readability of the generated tests and provide some kind of traceability to the used OpenAPI schema components.

RQ3: *The tests generated by EVOMASTER are of practical usefulness in industrial contexts. Compared to existing manually written tests, these generated tests can cover many similar scenarios, as well as new ones. However, few important test scenarios are missing in the generated tests, which makes them not a full substitute for the manual tests.*

VI. RQ4: OPEN CHALLENGES

EVOMASTER is currently used by some test engineers at Volkswagen AG. Several challenges still need to be addressed before it can be used more widely inside Volkswagen AG. Improving fuzzing algorithms to increase coverage and fault detection is always beneficial. However, we will not discuss those latter here. We will rather focus on the challenges and features directly pointed out and requested by the test

engineers involved in this work. Note that these challenges and features have nothing specific to Volkswagen.

Domain Knowledge. There is a need to enhance and guide fuzzers with domain knowledge. This can be complex, representing business requirements and specific chains of API calls with specific database states. In a black-box testing context, it would likely be infeasible to automatically discover such information within reasonable time. Using `examples` and `links` in the OpenAPI schemas is a useful first step, but it is not sufficient. How to best provide information on such domain knowledge, and how to make sure that such information can be properly “interpreted” by the fuzzers, are open research questions. Furthermore, the *traceability* from the generated tests to this provided domain knowledge is crucial to verify sufficient correctness and completeness of the generated test suites.

Seeding Tests. Fuzzers are not meant to replace human software testers. For sure, at least not as long as the results of fuzzers are worse than test suites manually crafted by experienced test engineers [7]. Especially when dealing with existing APIs, in professional environments there might be existing end-to-end tests for those APIs. This could be written in programming languages such as Python and Java, using specialized libraries to make HTTP calls. Or they could be written in a domain-specific-languages (DSL) for specific API testing tools, such as for example Postman,⁶ Insomnia,⁷ Hoppscotch⁸ and Bruno.⁹ Using these existing tests (if any) as *seeds* for the fuzzers could boost their effectiveness, providing interesting starting test cases to fuzz. Seeding existing tests is a known technique to reuse existing domain knowledge, like for example done in search-based unit test generation with tools such as EvoSuite [64]. As such, seeding end-to-end tests for REST APIs is an important feature.

In the past, in EVOMASTER there was a first attempt, i.e., a proof-of-concept, to seed tests written for the Postman tool [65]. However, this so called test “carving” process was a fool’s errand. Unfortunately, there are just too many ways in industry to write end-to-end tests for REST APIs. Trying to support all of them is infeasible. Even if one concentrate on a single type, e.g., JUnit tests, there are simply too many

⁶<https://www.postman.com>

⁷<https://insomnia.rest>

⁸<https://hoppscotch.io>

⁹<https://www.usebruno.com/>

ways to write test cases and so many different libraries used to make HTTP calls. Implementing a test carver that can handle all these different cases is simply not a viable option.

Fortunately, though, in the case of REST APIs, a feasible alternative could be to use a HTTP proxy. Regardless of how the existing tests are written (e.g., in Python or in the DSL used by Bruno), they make HTTP calls. All these HTTP calls can be intercepted and analyzed with a proxy, regardless of how the tests were written. At a high level, this would be similar to how black-box coverage tools such as Restats work [66]. Then, from these HTTP logs, the proxy can output a format representing the executed tests that can be directly used by the chosen fuzzer (e.g., EVOMASTER in our case).

One challenge here, though, is how to deal with dynamic data. In other words, when some output of a call is used as input in a following call. For example, recall the cases of Figure 5. All these cases would need to be detected and handled by the proxy. If not, if the input data is used as it is (e.g., with hardcoded authentication tokens), then the seeded tests could become flaky.

One advantage of test seeding is that it can enable *test enhancement* as well. Given an output test suite generated by EVOMASTER, testers could modify it to improve it. These improvements will not be lost if then these modified test suites can be fed back to the fuzzer as new seeds.

Test Case Reuse. When academics run experiments with different fuzzers, or different configurations of the same fuzzer, those are run for a certain amount of time. This could be for example 10 minutes, 1 hour or 24 hours. But this is not really close to the development cycles in industry. Usually, a REST API is not implemented in one day, tested once, and then be done with it. It might takes weeks, months or years to develop and maintain/update an API. Each day, any new code change in the API could require and warrant a new fuzzing session. Therefore, throughout the lifespan of an API, it might require to be fuzzed hundreds of times.

With the passing of time, each new introduced code change might invalidate any existing regression test. Still, fuzzing from scratch each day would be inefficient. It would likely be beneficial to re-use the generated tests from the latest run like “seeds”. This would create a chain of seeding, starting from the first run. However, as the semantics of the API and its OpenAPI schema might change through time, there is the need to make sure to do not crash the fuzzer if any previous test case is no longer valid (e.g., referring to an endpoint or parameter that no longer exist).

Conceptually, this is very similar to the previously discussed seeding from existing tests. Two differences though: (1) as already mentioned, some tests might be obsolete, or referring to no longer existing parameters. A fuzzer should not crash in those cases; (2) besides the output type chosen by the user (e.g., JUnit 5 in Kotlin), a copy of the tests can be saved as well in any internal representation used by the fuzzer, as only the fuzzer needs to read those copied tests. This latter point should make this feature easier to implement than, for example, a proxy used to record and analyze the execution of

existing tests on-the-fly.

This concept of seeding from previous executions has already been investigated in the literature, for example in the context of unit test generation in Continuous Integration [67]. From a research standpoint, there would not be much novelty here. Still, this would be a critical feature to have for a successful technology transfer from academic research to industrial practice.

Test Names. Currently, EVOMASTER generate tests that are named with a counter, like `test_0` and `test_2` (recall Figure 2 and Figure 5). This is not good for practitioners, as it increases the cognitive load needed to understand what the generated tests do. In this regards, test generators based on LLM techniques scored better than EVOMASTER [7].

In the literature, in particular for unit test generation, different techniques have been developed to provide better test names [68], [69]. There is a clear avenue for research on novel techniques tailored for REST API testing. However, readability is not really something that can be objectively quantified (e.g., like achieved code coverage or detected faults). It requires “human studies”, i.e., to check if indeed a novel naming strategy is better than current practice you need test engineers to manually look at and evaluate those generated tests. Unfortunately, user studies with human subjects in industry based on tool outputs are very risky, expensive and tricky to carry out and then publish in the software engineering research community [70], as we have unfortunately directly experienced for example in [71]. As such, this type of studies is rare in the literature. In our professional experience, it is not something we can really recommend to do, especially for untenured young researchers.

On the one hand, from a practitioner point of view, better test naming is a critical feature. On the other hand, dealing with user studies in industry is a high risk endeavour for researchers. Solving this conundrum is an open challenge. However, from a practical view point, existing best practices about naming conventions and commenting for tests could be used as a first step, while research will be needed to find a more fit and appropriate approach.

Database Handling. As we have previously discussed, using existing data in the databases (if any is used by the tested API) would be beneficial for the fuzzing. However, once a fuzzer starts to interact with existing data, there is no guarantee that the state of the database will be the same after the fuzzing has ended. If other manual test cases rely on specific existing data, this can be a problem. Rollbacking the database to a previous snapshot after each fuzzing session is technically possible, but not ideal.

There can be two major different kinds of modifications: (1) delete/change existing data and (2) create new data.

The latter case (2) is perhaps less problematic, as long as the fuzzing does not create gigabytes or terabytes of data per hour. Explicit DELETE operations could be added after each test case execution. However, in a black-box approach, determine which data is created, and which endpoint would result in deleting it, is not always simple. For example, not all APIs follow REST architectural guidelines on how to design

hierarchical endpoints to manipulate resources in a predictable way.

The case (1) of deleting/changing existing data can be indeed problematic. One approach could be to do not directly interact with such data, but rather read it (e.g., with GET) and then make copies of it (e.g., create new resources with same data using operations such as POST). Then, in each test that creates new data, DELETE operations can be automatically added.

Handling these issues might require making new HTTP calls (e.g., DELETE), using different strategies. This, however, could reduce the performance of the fuzzer, as those extra calls take time. How to best handle this issue is an important research problem.

AI Hype. Currently, as of 2024, there is a lot of hype about what Generative AI can do in industry. Several fields have experienced disruptive innovations thanks to Generative AI. As for everything, there are always strengths and weaknesses in any new approach. Therefore, how these novel techniques can be best used in software testing is an ongoing research investigation. The use of AI-based techniques was a decisive factor to choose EVOMASTER (recall Section III). However, further AI techniques could be exploited to achieve better results.

In the case of testing REST APIs, there has been some work in leveraging Large Language Models (LLM), like for example to analyze possible parameter constraints expressed in natural language documentation [72], or to sample possible valid inputs based on their name [25]. Integrating this type of techniques inside EVOMASTER, or combining other LLM approaches with it, is an important feature, based on the internal long-term vision for strategic focus in large enterprises such as Volkswagen AG.

RQ4: *how to best exploiting existing tests, as well as tests generated from previous runs of the fuzzer, is of paramount importance. How to best name the generated test cases, and how to deal with databases, are also important challenges to investigate. Furthermore, innovations from novel AI techniques are highly sought in industry.*

VII. THREATS TO VALIDITY

This industrial report shares our lessons learned in the enhancement of a specific tool (i.e., EVOMASTER) applied at a real-world enterprise (i.e., Volkswagen AG). Threats to external validity include how these lessons learnt could be applicable to other tools and enterprises. Most of the discussed enhancements are rather general, and likely any fuzzer that wants to be applicable to industrial APIs would need to implement the same or similar features. Furthermore, there is nothing specific in the analyzed APIs that is limited to Volkswagen AG. From a technical perspective, those APIs share many high-level characteristics with existing open-source APIs, such as the ones collected in EMB [63] used for evaluating fuzzing techniques. As such, it seems feasible that our results could generalize in other industrial contexts

and tools, although this is not possible to know for certain without further studies.

The identified challenges and research priorities are based on the feedback from some test engineers in the Group IT of Volkswagen AG. Even within the same organization, other test engineers might have different opinions. Therefore, we cannot say for sure that what identified are indeed the most important challenges and priorities for fuzzing REST APIs in industry. Regardless, those are important problems that need to be addressed.

Regarding the empirical study, threats to internal validity might come from faults in our new feature implementations. For example, faults in the implementation of handling examples and links features might have negatively impacted our results. As EVOMASTER is open-source on GitHub, with new releases automatically uploaded to Zenodo, anyone can review its code.

Another potential threat is that, as the generated test suites were manually evaluated with the existing test suites, such manual process might have been affected by human mistakes. Unfortunately, due to confidentiality and intellectual property constraints, we cannot provide a replication package for this industrial study.

VIII. CONCLUSION

Automated testing of REST APIs is a topic that has attracted a lot of interest from the research community in the last few years [8]. A reason for this is that the verification of REST APIs is of paramount importance in industry, especially in large enterprises such as Volkswagen AG. However, no experience report of introducing these novel techniques in industry has been presented in the academic literature so far [8], till now.

In this paper, we have reported our experience in the technology transfer from academic results (e.g., our EVOMASTER search-based fuzzer) to industrial practice (e.g., at Volkswagen AG). We have discussed which technological and scientific challenges we have faced in making EVOMASTER more usable for practitioners in industry. Practical solutions have been presented and evaluated. Still, there are several challenges that need to be addressed, which we summarized in this paper.

The challenges reported in this paper do not seem to be specific to EVOMASTER or Volkswagen. Likely, any other fuzzer or enterprise will face similar issues. As such, this industry report can provide useful information for other researchers on fuzzing techniques that want to see them applied in industry practice.

Our fuzzer EVOMASTER is open-source, and freely available on GitHub: <https://github.com/WebFuzzing/EvoMaster>

ACKNOWLEDGMENT

Andrea Arcuri is supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972).

REFERENCES

- [1] “APIs.guru,” <https://apis.guru/>, online, Accessed August 6, 2024.
- [2] “RapidAPI,” <https://rapidapi.com/>, online, Accessed August 6, 2024.
- [3] S. Newman, *Building microservices*. ” O’Reilly Media, Inc.”, 2021.
- [4] R. Rajesh, *Spring Microservices*. Packt Publishing Ltd, 2016.
- [5] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcode: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.
- [6] A. Arcuri, J. P. Galeotti, B. Marculescu, and M. Zhang, “Evomaster: A search-based system test generation tool,” *Journal of Open Source Software*, vol. 6, no. 57, p. 2153, 2021.
- [7] A. Poth, O. Rjollli, and A. Arcuri, “Technology adoption performance evaluation applied to testing industrial rest apis,” *Automated Software Engineering*, vol. 32, no. 1, p. 5, 2025.
- [8] A. Golmohammadi, M. Zhang, and A. Arcuri, “Testing restful apis: A survey,” *ACM Transactions on Software Engineering and Methodology*, aug 2023. [Online]. Available: <https://doi.org/10.1145/3617175>
- [9] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, “The fuzzing book,” <https://www.fuzzingbook.org>, 2019.
- [10] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, “Fuzzing: A survey for roadmap,” *ACM Computing Surveys*, vol. 54, no. 11s, sep 2022. [Online]. Available: <https://doi.org/10.1145/3512345>
- [11] P. Godefroid, “Fuzzing: Hack, art, and science,” *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.
- [12] V. Atlidakis, P. Godefroid, and M. Polishchuk, “Restler: Stateful REST API fuzzing,” in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2019, p. 748–758.
- [13] E. Vigliani, M. Dallago, and M. Ceccato, “Resttestgen: Automated black-box testing of restful apis,” in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2020.
- [14] O. Nourry, G. BAVOTA, M. LANZA, and Y. KAMEI, “The human side of fuzzing: Challenges faced by developers during fuzzing activities,” *ACM Transactions on Software Engineering and Methodology*, 2023.
- [15] A. Poth and C. Heimann, “How to innovate software quality assurance and testing in large enterprises?” in *Systems, Software and Services Process Improvement: 25th European Conference, EuroSPI 2018, Bilbao, Spain, September 5-7, 2018, Proceedings 25*. Springer, 2018, pp. 437–442.
- [16] —, “Frugal innovation approaches combined with an agile organization to establish an innovation value stream,” in *European Conference on Software Process Improvement*. Springer, 2023, pp. 260–274.
- [17] V. Garousi, D. Pfahl, J. M. Fernandes, M. Felderer, M. V. Mäntylä, D. Shepherd, A. Arcuri, A. Coşkunçay, and B. Tekinerdogan, “Characterizing industry-academia collaborations in software engineering: evidence from 101 projects,” *Empirical Software Engineering*, vol. 24, no. 4, pp. 2540–2602, 2019.
- [18] V. Garousi, K. Petersen, and B. Ozkan, “Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review,” *Information and Software Technology (IST)*, vol. 79, pp. 106–127, 2016.
- [19] V. Garousi, M. Felderer, M. Kuhrmann, and K. Herkiloğlu, “What industry wants from academia in software testing?: Hearing practitioners’ opinions,” in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2017, pp. 65–69.
- [20] V. Garousi and M. Felderer, “Worlds apart: a comparison of industry and academic focus areas in software testing,” *IEEE Software*, vol. 34, no. 5, pp. 38–45, 2017.
- [21] A. Poth, O. Rjollli, and A. Riel, “Integration-and system-testing aligned with cloud-native approaches for devops,” in *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*. IEEE, 2022, pp. 201–208.
- [22] M. Kim, S. Sinha, and A. Orso, “Adaptive rest api testing with reinforcement learning,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 446–458.
- [23] N. Laranjeiro, J. Agnelo, and J. Bernardino, “A black box tool for robustness testing of rest services,” *IEEE Access*, vol. 9, pp. 24 738–24 754, 2021.
- [24] “Cats: REST API Fuzzer and negative testing tool for OpenAPI endpoints,” <https://github.com/Endava/cats>.
- [25] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato, “Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning,” *arXiv preprint arXiv:2408.08594*, 2024.
- [26] “Language-agnostic HTTP API Testing Tool,” <https://github.com/apiaryio/dredd>.
- [27] “Fuzz-lightyear: Stateful fuzzing framework,” <https://github.com/Yelp/fuzz-lightyear>.
- [28] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao, “Morest: Model-based restful api testing with execution feedback,” in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2022.
- [29] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “RESTest: Automated Black-Box Testing of RESTful Web APIs,” in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2021, pp. 682–685.
- [30] H. Wu, L. Xu, X. Niu, and C. Nie, “Combinatorial testing of restful apis,” in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2022.
- [31] Z. Hatfield-Dodds and D. Dygalo, “Deriving semantics-aware fuzzers from web api schemas,” in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2022, pp. 345–346.
- [32] “Tcases for OpenAPI: From REST-ful to Test-ful,” <https://github.com/Cornutum/tcases/tree/master/tcases-openapi>.
- [33] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, “An industrial evaluation of unit test generation: Finding real faults in a financial application,” in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2017, pp. 263–272.
- [34] C. Zhang, Y. Yan, H. Zhou, Y. Yao, K. Wu, T. Su, W. Miao, and G. Pu, “Smartunit: Empirical evaluations for automated unit testing of embedded software in industry,” in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2018, pp. 296–305.
- [35] M. Brunetto, G. Denaro, L. Mariani, and M. Pezzè, “On introducing automatic test case generation in practice: A success story and lessons learned,” *Journal of Systems and Software*, vol. 176, p. 110933, 2021.
- [36] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, “Deploying search based software engineering with Sapienz at Facebook,” in *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2018, pp. 3–45.
- [37] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, “An empirical study of android test generation tools in industrial cases,” in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 738–748.
- [38] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer, “What it would take to use mutation testing in industry—a study at facebook,” in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 268–277.
- [39] G. Petrović and M. Ivanković, “State of mutation testing at google,” in *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2018, pp. 163–171.
- [40] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, “Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study,” *IEEE Transactions on Software Engineering (TSE)*, 2018.
- [41] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, “Root causing flaky tests in a large-scale industrial setting,” in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, ser. ISSTA 2019, 2019, p. 101–111.
- [42] “EvoMaster,” <https://github.com/WebFuzzing/EvoMaster>.
- [43] A. Arcuri, “EvoMaster: Evolutionary Multi-context Automated System Test Generation,” in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018.
- [44] A. Arcuri, M. Zhang, A. Belhadi, B. Marculescu, A. Golmohammadi, J. P. Galeotti, and S. Seran, “Building an open-source system test generation tool: lessons learned and empirical analyses with evomaster,” *Software Quality Journal*, pp. 1–44, 2023.
- [45] A. Arcuri, M. Zhang, S. Seran, J. P. Galeotti, A. Golmohammadi, O. Duman, A. Aldasoro, and H. Ghianni, “Tool report: Evomaster—black and white box search-based fuzzing for rest, graphql and rpc apis,” *Automated Software Engineering*, vol. 32, no. 1, pp. 1–11, 2025.
- [46] A. Arcuri, “Restful api automated test case generation with evomaster,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, p. 3, 2019.
- [47] —, “Automated black-and white-box testing of restful apis with evomaster,” *IEEE Software*, vol. 38, no. 3, pp. 72–78, 2020.

- [48] A. Belhadi, M. Zhang, and A. Arcuri, “Random testing and evolutionary testing for fuzzing graphql apis,” *ACM Transactions on the Web*, 2023.
- [49] M. Zhang, A. Arcuri, Y. Li, Y. Liu, and K. Xue, “White-box fuzzing rpc-based apis with evomaster: An industrial case study,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–38, 2023.
- [50] M. Zhang and A. Arcuri, “Adaptive hypermutation for search-based system test generation: A study on rest apis with evomaster,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, 2021.
- [51] A. Arcuri and J. P. Galeotti, “Enhancing Search-based Testing with Testability Transformations for Existing APIs,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–34, 2021.
- [52] A. Arcuri, M. Zhang, and J. P. Galeotti, “Advanced white-box heuristics for search-based fuzzing of rest apis,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2024. [Online]. Available: <https://doi.org/10.1145/3652157>
- [53] A. Arcuri and J. P. Galeotti, “Handling sql databases in automated test generation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–31, 2020.
- [54] S. Seran, M. Zhang, and A. Arcuri, “Search-based mock generation of external web service interactions,” in *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2023.
- [55] M. Kim, Q. Xin, S. Sinha, and A. Orso, “Automated test generation for rest apis: No time to rest yet,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 289–301. [Online]. Available: <https://doi.org/10.1145/3533767.3534401>
- [56] M. Zhang and A. Arcuri, “Open problems in fuzzing restful apis: A comparison of tools,” 2023. [Online]. Available: <https://doi.org/10.1145/3597205>
- [57] H. Sartaj, S. Ali, and J. M. Gjøby, “Rest api testing in devops: A study on an evolving healthcare iot application,” 2024. [Online]. Available: <https://arxiv.org/abs/2410.12547>
- [58] A. Arcuri, M. Zhang, A. Belhadi, S. Seran, J. P. Galeotti, Bogdan, A. Golmohammadi, O. Duman, A. Aldasoro, A. M. López, H. Ghianni, A. Panichella, K. Niemeyer, and M. Maugeri, “Emresearch/evomaster: v3.0.0,” Apr. 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.10932122>
- [59] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, “Automated black-box testing of nominal and error scenarios in RESTful APIs,” *Software Testing, Verification and Reliability*, p. e1808, 2022.
- [60] M. Zhang, B. Marculescu, and A. Arcuri, “Resource and dependency based test case generation for restful web services,” *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–61, 2021.
- [61] A. Poth and P. Momen, “Sustainable software engineering—a contribution puzzle of different teams in large it organizations,” *Journal of Software: Evolution and Process*, p. e2677, 2024.
- [62] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “Test coverage criteria for restful web apis,” in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2019, pp. 15–21.
- [63] A. Arcuri, M. Zhang, A. Golmohammadi, A. Belhadi, J. P. Galeotti, B. Marculescu, and S. Seran, “Emb: A curated corpus of web/enterprise applications and library support for software testing research,” in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 433–442.
- [64] J. M. Rojas, G. Fraser, and A. Arcuri, “Seeding strategies in search-based unit test generation,” *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 366–401, 2016.
- [65] A. Martin-Lopez, A. Arcuri, S. Segura, and A. Ruiz-Cortés, “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)*. IEEE, 2021, pp. 231–241.
- [66] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, “Restats: A test coverage tool for restful apis,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 594–598.
- [67] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, “Continuous test generation: enhancing continuous integration with automated test generation,” in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 55–66.
- [68] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, “Modeling readability to improve unit tests,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 107–118.
- [69] E. Daka, J. M. Rojas, and G. Fraser, “Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 57–67.
- [70] M. C. Davis, E. Aghayi, T. D. LaToza, X. Wang, B. A. Myers, and J. Sunshine, “What’s (not) working in programmer user studies?” *ACM Transactions on Software Engineering and Methodology*, 2023.
- [71] M. Zhang, A. Arcuri, Y. Li, Y. Liu, K. Xue, Z. Wang, J. Huo, and W. Huang, “Fuzzing microservices: A series of user studies in industry on industrial systems with evomaster,” 2024. [Online]. Available: <https://arxiv.org/abs/2208.03988>
- [72] M. Kim, T. Stennett, D. Shah, S. Sinha, and A. Orso, “Leveraging large language models to improve rest api testing,” in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 37–41.