

# Fuzzing Microservices: A Series of User Studies in Industry on Industrial Systems with EvoMaster

Man Zhang<sup>a,\*</sup>, Andrea Arcuri<sup>b</sup>, Yonggang Li<sup>c</sup>, Yang Liu<sup>c</sup>, Kaiming Xue<sup>c</sup>, Zhao Wang<sup>c</sup>,  
Jian Huo<sup>c</sup>, Weiwei Huang<sup>c</sup>

<sup>a</sup>*Beihang University, Beijing, China*

<sup>b</sup>*Kristiania University of Applied Sciences and Oslo Metropolitan University, Oslo, Norway*

<sup>c</sup>*Meituan, Beijing, China*

---

## Abstract

With several microservice architectures comprising thousands of web services in total, used to serve 630 million customers, companies like Meituan face several challenges in the verification and validation of their software. The use of automated techniques, especially advanced AI-based ones, could bring significant benefits here. EVOMASTER is an open-source test case generation tool for web services, that exploits the latest advances in the field of Search-Based Software Testing research. This paper reports on our experience of integrating the EVOMASTER tool in the testing processes at Meituan over almost 2 years (i.e., between October 2021 and July 2023). Two user studies were carried out in 2021 (with two industrial APIs) and in 2023 (with three industrial APIs) to evaluate two versions of EVOMASTER (i.e., v1.3.0 and v1.6.1), respectively, in tackling the test generation for industrial web services which are parts of a large e-commerce microservice system. The two user studies involve in total 321,131 lines of code from these five APIs and 27 industrial participants at Meituan. Questionnaires and interviews were carried out in both user studies with the engineers and managers at Meituan. The two user studies demonstrate clear advantages of EVOMASTER (in terms of code coverage and fault detection) and the urgent need to have such a fuzzer in industrial microservices testing. Given its clear advantages, EVOMASTER now has been integrated into the industrial testing pipelines at Meituan. To study how these results could generalize, a follow up user study was done in 2024 (with EVOMASTER v2.0.0) with five engineers in the five different companies. Our results show that, besides their clear usefulness, there are still many critical challenges that the research community needs to investigate to improve performance further.

**Keywords:** Empirical industrial study, Automated test generation, SBST, Fuzzing, Microservices

---

---

\*Corresponding author

*Email addresses:* manzhang@buaa.edu.cn (Man Zhang), andrea.arcuri@kristiania.no (Andrea Arcuri), liyonggang@meituan.com (Yonggang Li), liuyang352@meituan.com (Yang Liu), xuekaiming@meituan.com (Kaiming Xue), wangzhao25@meituan.com (Zhao Wang), huojian@meituan.com (Jian Huo), 529852594@qq.com (Weiwei Huang)

## 1. Introduction

In microservice architectures [1], large enterprise systems are split in hundreds/thousands of connected web services. This software architecture is widely applied in industry, to enable rapid and frequent delivery of the services in production. As the number of services grows and evolves over time in enterprise microservices, there are several challenges in the verification and validation of such systems, e.g., limited resources for testing hundreds of APIs coupled with the complexity of complex interactions, and maintaining system reliability posed by frequent updates. Engineers in industry have an immediate need for automated testing techniques to address these challenges, offering an opportunity for the research community to contribute. However, one challenge we have encountered as researchers working in microservice testing is that large enterprise microservices are rarely present in open-source repositories. The performance of newly proposed approaches is typically assessed on open-source APIs [2, 3, 4, 5, 6, 7] (such as APIs collected in the GitHub repository EMB [8, 9]). Unfortunately, most of these open-source APIs are web services that work in isolation, and are not part of a microservice architecture [8, 9].

EvoMaster [10, 11, 12] is an open-source academic prototype designed for addressing the challenges of microservice testing such as enabling search-based fuzzing of enterprise Web APIs that are part of a microservices architecture. With the term “fuzzing” [13, 14, 15] we simply mean the *ability of generating system level test cases that can detect faults* (e.g., program crashes). For generating test cases efficiently, we rely on the output of several decades of scientific research in the field of *Evolutionary Computation*, in particular in the context of *Search-Based Software Testing* (SBST) [16, 17, 18]. Furthermore, to be actually of use for practitioners, test cases should be *readable* (i.e., easy to understand), as one of their main goals is to help *debugging* when these tests detect faults. Note: in the scientific community there is no consensus on how these terms Fuzzing and SBST are and should be used [19]; it is an ongoing debate. According to the aforementioned definition, we state that EVOMASTER is a *search-based fuzzer*.

EVOMASTER has been evaluated as the most performing tool in recent studies on REST API fuzzers [20, 7]. As part of industry-driven research [21], it has been evaluated on industrial APIs as well (e.g., in [22]). Unfortunately, those were part of small systems. Since 2021, Meituan<sup>1</sup> collaborated with us to seek an automated API testing solution for optimizing their development/testing processes for microservices. Meituan is a large e-commerce company, and Meituan Select, a division of Meituan, is an e-commerce platform for community group bulk buying (e.g., fresh vegetables, meat). The platform is built with hundreds of web services (also known as microservices) that manage various business processes, such as ordering, allocation, packing, delivery, and payments. To better understand this industrial problem, it is important to identify the limitations of our current approach (i.e., EVOMASTER) and challenges that industry faces that the research community can address further. For a test generation tool, it is not only a matter of finding faults. How actual practitioners use those tools, and how they integrate them in their development processes, is of paramount importance [23]. This is particularly the case for large enterprise systems, where several non-technical challenges are present as well (e.g., communication issues when there are hundreds of engineers, and dependencies among different services developed by different teams). Therefore, we carried out two user

---

<sup>1</sup>[www.meituan.com/en-US/about-us](http://www.meituan.com/en-US/about-us)

studies with our industrial partner (i.e., Meituan), involving real industrial system under tests (SUTs) which are parts of a large-scale e-commerce system, and involving employees who can provide feedback from the point of view of practitioners through questionnaires and interviews. It is worth noting that EVOMASTER has now been integrated into the testing pipeline at Meituan, due to the clear advantages observed. To further generalize our findings, we also conducted a follow-up study outside of Meituan with engineers who volunteered to participate, including five participants from five different companies. Our main objectives in this study are:

- to evaluate the *usability*<sup>2</sup> of EVOMASTER in industrial contexts;
- to assess the *effectiveness*<sup>3</sup> of EVOMASTER at tackling test generation problems in a real, large industrial setting;
- to investigate essential features for enabling its further adoption into industrial practice;
- further, to study existing testing challenges in industry which could be addressed by the research community.

The first user study was conducted at the starting phase of the collaboration with Meituan in 2021. In the first user study, we employed EVOMASTER on two real industrial services, performed a detailed review of results achieved by EVOMASTER (e.g., its generated test cases), and conducted a questionnaire and interviews with eight industrial practitioners at Meituan. The questions in our interviews/questionnaire in the user study are a superset of an existing study in the unit testing domain [24], which enable us to do some forms of meta-analysis. Regarding feedback on the usability of the tool, for the engineers at Meituan, there did not seem to exist any major difficulty in applying the tool on their industrial applications. For effectiveness, results on the two chosen APIs show that EVOMASTER achieved up to 33.5% line coverage, and on average (i.e., arithmetic mean over two APIs) 26 faults could be detected by manually reviewing the generated tests. Such results demonstrated the potential benefits achieved by our approach to our industrial partner and can further strength the research-industry collaboration.

The second user study was a follow-up study conducted at Meituan in 2023 with our latest technique (i.e., addressed the most important challenges for Meituan by designing a novel approach for enabling a native fuzzing of APIs developed with Remote Procedure Call (RPC) [25]) for demonstrating how the identified challenges have been addressed and what current progress of integrating our fuzzer (academic prototype) into industrial practice is after 1.5 years (i.e., from October 2021 to May 2023). In the follow up study in 2023, we further included some new important questions that were not considered in the first study. Compared to the results obtained in the first study, readability of generated tests has been improved due to the support for RPC, such as native function invocation and thrown exceptions (used to identify potential faults) shown in the tests, and tests grouped by distinguishing whether they represent potential faults. Regarding

---

<sup>2</sup>the extent to which a new approach (such as EVOMASTER) is easy to apply and use in industrial practice

<sup>3</sup>the extent to which objectives can be achieved by the approach (such as EVOMASTER) in testing large-scale industrial applications

the effectiveness on the three RPC APIs, EVOMASTER achieved up to 33.4% line coverage and identified on average (i.e., arithmetic mean) 64.8 potential faults automatically. With the further feedback from the participants, they pointed out that readability should be further improved by having meaningful strategies to group and name tests, and the code coverage also needs to be improved by testing more combinations of various RPC functions and interfaces in order to better cover the business logic of their systems.

Besides assessing the usability and effectiveness of EVOMASTER, with these two user studies, we also investigated the potential adoption of our approach to industrial practice through the research-industry collaboration. We report on the major updates which our industrial partner and us have developed for enabling integration of EVOMASTER into their industrial development processes. EVOMASTER has now been integrated into the industrial pipelines of our industrial partner, and is used daily for testing tasks impacting hundreds of engineers at Meituan. Moreover, we discuss lessons learned in terms of testing setup, testing criteria, locating faults, and assertion generation while conducting this study. Furthermore, we summarize important common challenges we faced and future directions in the fuzzing of industrial microservices.

One limitation of these two user studies is that they are done in industry, with only one enterprise, i.e., Meituan. In contrast to novel algorithm advances (which require no evaluation by practitioners in industry to provide a scientific contribution), or user studies with students (which can provide generalizable results [26, 27]), user studies in industry with software engineering practitioners must be carried out in several different enterprises to have scientific value. To address this threat to validity, a third user study outside of Meituan was carried out in 2024. This involved five practitioners from five different Chinese enterprises, using online questionnaire, and it employed the same questions as the second user study from 2023. As we do not have formal collaboration agreements with these five companies, the five participants volunteered to take their free time to be part of this user study. Results showed that responses to *usability* were much more negative than those we received at Meituan. The task of writing drivers to enable white-box testing appeared to be significantly more difficult for participants outside of Meituan. In terms of *effectiveness*, all participants stated that they did not have enough time to inspect the generated tests in details. This indicates the challenge of conducting this type of user study in companies where there is no formal collaboration, and so the participants cannot spend too much of their free time on such a study (they could spend a few hours, but not more). In terms of *integration* and *existing challenges*, as they are independent of EVOMASTER, we found that most of the responses from outside of Meituan are consistent with what collected in the first two user studies conducted at Meituan.

The main contribution of this paper is an empirical study of the application of EVOMASTER, i.e., an academic fuzzer, in a real industrial setting, targeting the domain of large scale microservice architectures, including how software engineers would integrate such fuzzers in their development processes. The study reports on our experience of two user studies conducted with our industrial partner Meituan in assessing the usability and effectiveness of our academic fuzzer in an industrial setting, investigating the adoption of our fuzzer in industrial practices, and identifying important challenges existing in this industrial context. A third study was carried out outside of our industrial partner aimed at generalizing our finding to other enterprises.

We believe that the evaluation of scientific research in actual industrial settings, albeit within a single company as our industrial partner and with volunteer participants from

five different companies, is an essential step toward addressing the significant gap between research and practice. However, we do acknowledge that this is a very controversial statement in the Software Engineering Research community. On the one hand, research work only evaluated in the lab might rely on assumptions that might not hold true in practice. On the other hand, industry-relevant research with collaborations with industry partners take a massive amount of time [28], and it seems rare that this kind of work involve more than one industrial partner at a time [29]. This can lead to less generalizable results. Both kinds of studies are important, but, considering the current trends in Software Engineering research, “*Generalizability is overrated*” [30]. There is a large body of scientific work in designing novel fuzzing techniques, but their application by actual practitioners in industry is a software engineering aspect that is not widely investigated [23].

The paper is organized as follows. We introduce background information in Section 2, and discuss related work in Section 3. Section 4 presents our design of the empirical study followed by experimental results discussion in Section 5. Lessons learned and common challenges are summarized in Section 6. We clarify threats to validity in Section 7 and conclude the paper in Section 8.

## 2. Background

### 2.1. Web APIs

Application Programming Interface (API) is a common practice in building enterprise microservices. For example, REST and RPC are two methods used widely for facilitating communications (typically based on HTTP) among services in an enterprise [1].

#### 2.1.1. REST and OpenAPI

REpresentational State Transfer (REST) is a set of architectural guidelines rather than a protocol [31]. It is composed of a collection of design principles used to develop modern web services, using the HTTP/HTTPS protocol. For instance, to better manage resources in web services, REST suggests that resources should be identified by the Uniform Resource Identifiers (URIs), and implementations of operations on resources should always follow semantics of HTTP methods, e.g., GET should be used to retrieve resource(s). Nowadays, REST has been applied in many companies, such as Google<sup>4</sup>, Amazon<sup>5</sup>, and X (formerly Twitter)<sup>6</sup>.

OpenAPI<sup>7</sup> (previously known as Swagger) is a specification that standardizes descriptions of REST APIs. Such a specification allows both human and machine to understand how the API works and how to interact with its services. An example as below shows how an endpoint of a REST API can be described using OpenAPI. In this example, information of a pet can be retrieved by an HTTP GET request with a given pet ID of type `integer`, and the “responses” also lists possible HTTP status codes, e.g., “404” status code indicates that a pet with the given ID cannot be found.

---

<sup>4</sup><https://developers.google.com/drive/v2/reference/>

<sup>5</sup><http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html>

<sup>6</sup><https://dev.twitter.com/rest/public>

<sup>7</sup><https://www.openapis.org/>

```

1 /pet/{petId}: {
2   "get": {
3     "summary": "Find pet by ID",
4     "description": "Returns a single pet",
5     "operationId": "getPetById",
6     "produces": [
7       "application/json",
8       "application/xml"
9     ],
10    "parameters": [
11      {
12        "name": "petId",
13        "in": "path",
14        "description": "ID of pet to return",
15        "required": true,
16        "type": "integer",
17        "format": "int64"
18      }
19    ],
20    "responses": {
21      "200": {
22        "description": "successful operation",
23        "schema": {"$ref": "#/definitions/Pet"}
24      },
25      "400": {
26        "description": "Invalid ID supplied"
27      },
28      "404": {
29        "description": "Pet not found"
30      }
31    }
32  },
33  "delete": {...

```

Figure 1: An example of OpenAPI specification

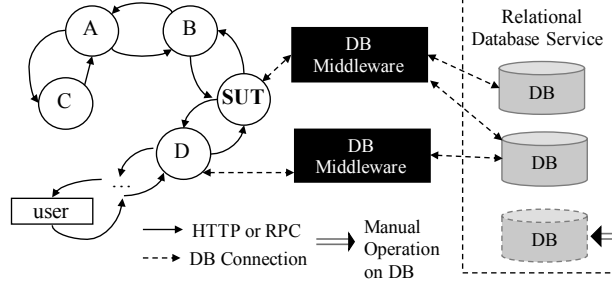


Figure 2: An example of the SUT with its connected services

### 2.1.2. RPC

Remote procedure call (RPC) is a communication protocol which is widely applied in enterprise services (such as Netflix, Alibaba), especially for distributed systems. An RPC API is defined by one or more RPC interfaces. Note that the implementation of the business logic is often organized by such interfaces. Each interface is further composed of a set of functions which are exposed to the other services. Unlike REST, there does not exist any widely accepted standards for describing RPC APIs, such as OpenAPI for REST. There exists diverse frameworks, such as Dubbo<sup>8</sup> by Alibaba, gRPC<sup>9</sup> by Google, Tars<sup>10</sup> by Tencent, Thrift<sup>11</sup> by Apache, and companies might also develop their own in-house RPC frameworks for satisfying their needs, e.g., Meituan developed MThrift, a modified version of Apache Thrift.

Figure 2 shows an example of a large microservice architecture with APIs, such as RPC and REST. With RPC communication, a stub is needed for supporting interactions between services. For instance, if the service *SUT* needs to access a function `foo` provided in *B* (Figure 2), the stub of *B* is instantiated in the *SUT*, that then invokes it as `result=stubB.foo()`. In this paper, the SUTs from Meituan are all RPC Java web services interacting with many other services and databases (as *SUT* shown in Figure 2). The SUTs were chosen by the team at Meituan we are in contact with, as those services implement their core business.

### 2.2. EvoMaster

EVOMASTER is an open-source testing tool designed for addressing system-level test case generation for enterprise web services (currently supports REST, GraphQL APIs and RPC APIs) [11, 32, 25]. An overview of EVOMASTER is shown in Figure 3. It is a search-based tool, using evolutionary algorithms to generate effective test cases. The tool enables both white-box [33, 34, 2, 35, 36] (for APIs running on the JVM and NodeJS) and black-box testing [37].

EVOMASTER is composed of two parts (see Figure 3), i.e., *Driver* and *Core*. *Driver* (only used when performing white-box testing) implements bytecode instrumentation,

<sup>8</sup><https://dubbo.apache.org>

<sup>9</sup><https://grpc.io>

<sup>10</sup><https://tarscloud.org/>

<sup>11</sup><https://thrift.apache.org/>

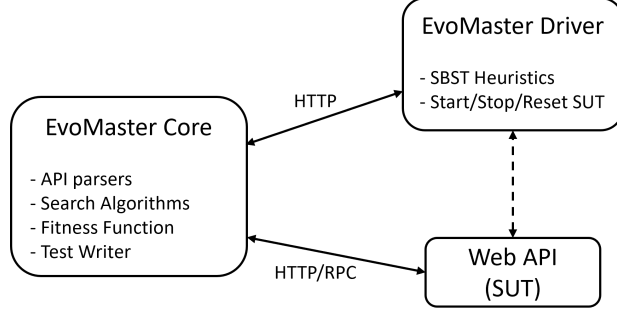


Figure 3: An overview of EvoMASTER

which enables the collection at runtime of code coverage and advanced white-box heuristics. To manipulate SUTs, *Driver* also defines interfaces which requires to be extended manually by users in order to configure how to start/stop/reset the SUTs. *Core* provides a set of search algorithms (such as MIO [34] and MOSA [38]) to generate tests, defines fitness functions to guide the search, and extracts information on how to access the SUT (e.g., OpenAPI [39] for REST APIs).

#### 2.2.1. The MIO Algorithm

The Many Independent Objective (MIO) [34] algorithm is specialized for system-level test case generation in the context of white-box testing. It is the default search algorithm for white-box EvoMASTER. Let us briefly summarize it here in this section. The algorithm is inspired by (1+1) EA [40], which only employs sampling and mutation operators. Pseudo-code of the algorithm is shown in Algorithm 1.

In the MIO algorithm, an *individual* is a test case, and the test case is composed of a sequence of actions (e.g., HTTP actions for REST API) to test the SUT. To guide the testing, we defined *fitness function*  $\delta$  with *testing targets*, which comprise all lines, branches, fault finding and domain specific targets (such as status code for REST). Each of them is a target to be optimized (i.e., covered) during the evolutionary search. For instance, for `x == 42`, we define two branch targets for the condition, i.e., one is for `true`, and other is for `false`. If the execution reaches the statement, we consider the two branch targets *Reached*. The `true` branch target can be considered as *Covered* only if the condition is evaluated as `true`, otherwise the other branch `false` is considered as *Covered*. Covering both branches would require the statement to be executed at least twice.

In the REST domain, 500 status code in the response could be used to represent a potential fault [7, 20], and the set of valid status codes for each endpoint is specified in the schema. Thus, each 500 status combined with distinct last executed line in the SUT is identified as a distinct potential fault. Any status code present in the response represents that a status code target is *Covered*, and an occurrence of invalid status code further considers that a mismatched schema fault is identified (i.e., *Covered*).

Each target has a population for it with a maximum size (denoted as  $n$ ). The MIO algorithm starts with an empty population (denoted as  $S$ ). At each iteration, the algorithm would either sample a new individual or mutate an existing individual in the



evolving populations, controlled by a probability  $P_r$ . The sampling operator is designed to generate new individuals by invoking various endpoints and combining them with different request sequences for exploring the diverse areas of the search space. If the individual reaches a new target, the MIO algorithm would create a new population for it (e.g., a new population  $T_k$  for the newly covered target  $k$ ). If a target is covered with the individual, the size of its corresponding population would be reduced to 1, and the target would not be considered further in the search. In the context of web service testing, the individual is a test which is a sequence of HTTP calls. After a certain percentage of search budget is used (denoted as  $F$ ), the MIO algorithm uses a *focused search*, where it increases the probability to mutate the individuals which have recent improvement, instead of exploring other areas of the search landscape. At the end of the search, the MIO algorithm outputs a test suite for the SUT, comprising the best evolved test cases for each testing target (denoted as  $A$ ).

Regarding *exploration* and *exploitation*, they are two fundamental phases in the search algorithm, i.e., *exploration*, which aims to discover diverse regions of the search space, and *exploitation*, which focuses on refining known promising regions. At the beginning of the search, *exploration* is typically applied with a higher probability in order to cover more regions. MIO controls the balance between the two phases using  $P_r$  and  $F$ .  $P_r$  decreases gradually to 0% once the  $F$  threshold is reached. By default in all three versions of EVOMASTER conducted in our user studies,  $P_r$  is set to 80%, while  $F$  is set to 50%. In terms of different search budget settings, e.g., 30 minutes, 1 hour and 10 hours, as controls for *exploration* and *exploitation*, the results achieved with 30 minutes and 1 hour are not comparable to the first 5% and 10% of the 10-hour search.

### 2.2.2. Integrated Novel Techniques

Throughout the years, since its inception in 2016, EVOMASTER has been extended with various novel techniques [41, 4, 6, 5, 32, 42] to serve a more comprehensive testing in the context of white-box testing for enterprise systems.

Enterprise web systems typically interact with databases and employ SQL to specify operations on it. To test such systems, states of interacted databases (such as what data are in the databases) would possibly have a direct impact on code coverage. EVOMASTER was enhanced with a SQL handling technique [41], which defines SQL query heuristics, and enables SQL execution monitoring and direct data insertion into the database with SQL commands from the tests. With such technique, EVOMASTER is capable of producing more effective tests with generated SQL commands based on the SQL heuristics to directly manipulate states of the SUT. In this context, a test (also referred as an individual) with additional SQL commands could result in a long and complex chromosome to evolve. To have an effective mutation for handling such individuals, an *adaptive weight-based hypermutation* [5] was developed that comprises a set of novel strategies to adaptively manipulate the number of genes to mutate, select genes to mutate, and guide how to mutate the values in these genes.

In the context of white-box testing, the *flag problem* is a common issue, i.e., lack of guidance to perform the search due to boolean expressions. To address this issue, *testability transformation* [4, 42] techniques have been developed in EVOMASTER that enable to transform the source code of the SUT (with *replacement* methods), for providing better gradient values, e.g., for *branch distance* computations [43]. In addition, test inputs are tracked in the transformed methods that further provides additional information

---

**Algorithm 1:** Pseudo-code of the MIO Algorithm [34]

---

**Input** : Stopping condition  $C$ , Fitness function  $\delta$ , Population size  $n$ , Probability for random sampling  $P_r$ , Start of focused search  $F$

**Output** : Archive of optimized individuals  $A$

```
1  $S \leftarrow SetOfEmptyPopulations()$ 
2  $A \leftarrow \{\}$ 
3 while  $\neg C$  do
4   if  $P_r > rand()$  then
5      $p \leftarrow RandomIndividual()$ 
6   else
7      $p \leftarrow SampleIndividual(S)$ 
8      $p \leftarrow Mutate(p)$ 
9   end
10  foreach element  $k \in ReachedTargets(p)$  do
11    if  $NewTarget(k)$  then
12       $S \leftarrow S \cup T_k$ 
13    end
14     $T_k \leftarrow T_k \cup \{p\}$ 
15    if  $IsTargetCovered(k)$  then
16       $UpdateArchive(A, p)$ 
17       $S \leftarrow S \setminus T_k$ 
18    else if  $|T_k| > n$  then
19       $RemoveWorst(T_k, \delta)$ 
20    end
21  end
22   $UpdateParameters(F, P_r, n)$ 
23 end
```

---

(referred as taint analysis) to better generate test data. Moreover, EVOMASTER uses specific transformations particularly for handling web service testing, e.g., to handle cases in which the OpenAPI schemas are underspecified.

REST is one of the most applied techniques for building web services in industry. To enable an automated testing for it, EVOMASTER has been integrated with a set of techniques based on REST domain. REST test generation problem was reformulated as a search problem [2] by defining various types of genes corresponding to REST schema, i.e., OpenAPI. In addition, the fitness function for MIO was enhanced with additional considerations on HTTP status codes and fault detection (i.e., 500 status code). Moreover, to better sample tests for REST, a smart sampling technique was developed with pre-defined structures. Furthermore, by further exploiting REST domain knowledge, a *resource-dependency based* MIO [6] was proposed that is composed of resource-based sampling, mutation and dependency heuristics for producing tests with effective resource handling. Besides REST APIs, EVOMASTER also enables the fuzzing of GraphQL APIs [32] (in 2022) and RPC APIs [25] (in 2023).

### 3. Related Work

Most of recent approaches for automated web service testing are developed for REST APIs in the context of black-box testing [44], e.g., [45, 46, 47, 48, 49, 50, 51]. For REST APIs, OpenAPI [39] provides a well-structured and machine-readable schema language to specify how to access these web services. Such schemas provide a base to enable automated testing techniques to access the system under test (SUT). For example, RESTler proposed by Atlidakis *et al.* [46] can automatically produce a sequence of requests to test REST APIs. The sequence is decided based on either a statistical analysis on the schema with request types, or a dynamic analyses on feedback received from previously executed requests. In addition, RESTler is now integrated with different testing techniques for regression testing [52], security aspect testing [53], and test data generation [47]. Viglianisi *et al.* [45] developed RESTTESTGEN that employs a dependency analysis on the schema to better generate test orders and inputs. The approach comprises two components, i.e., nominal tester and error tester, for finding defects by testing the SUT against nominal scenarios and error scenarios. Laranjeiro *et al.* [49] proposed bBOXRT for robustness testing in the context of black-box testing by testing the REST APIs with various invalid inputs. The invalid inputs are generated based on a set of mutation rules on data type with the schema. RESTest [50] developed by Martin-Lopez *et al.* is a black-box testing framework which has been integrated with different techniques such as fuzzing test inputs, adaptive random testing and constraint-based testing to generate tests for REST APIs. The constraint-based testing is performed based on inter-parameter dependencies analysis on the schema. RestCT [51] proposed by Wu *et al.* is a black-box combinatorial testing approach for REST APIs. The approach consists of two phases for generating orders and inputs of requests with OpenAPI specifications. Hatfield-Dodds and Dygalo developed Schemathesis [54] using property-based testing techniques. The tool derives structure and semantics of APIs for enabling REST API testing.

Considering that a black-box testing approach is independent from the programming language and source code, there are available more industrial SUTs to be used for evaluating it, compared with a white-box testing approach. For example, live services on the internet can be used for these kinds of experiments. Many industrial services (such as

Microsoft Azure, Office365 cloud services, Google Drive, Spotify), and hundreds of real REST APIs listed on public API aggregators, have been used to conduct experiments when evaluating the aforementioned black-box testing techniques. Several real faults were found, e.g., based on the returned HTTP status codes, in a process that is also known as *fuzzing*.

Testing REST APIs share some similar challenges with testing RPC APIs, e.g., related to the handling of interactions with databases and external services. At the time of our first experiment in 2021, there was no existing fuzzer for RPC APIs. Extending EVOMASTER to support RPC APIs [25] was a direct follow up from that first study, to fill such important gap in the research literature. Since that time, a work on supporting the testing of RPC APIs has been published in a workshop in 2023 [55], based on a master thesis published in 2022 [56]. Such work presents a black-box fuzzer based on OpenRPC specifications [57], which is not applicable to the systems of Meituan (as not using OpenRPC).

With recent studies [20, 7] conducted for studying existing fuzzers (including the fuzzers described above and EVOMASTER), white-box EVOMASTER achieved the best performance on open-source APIs. In addition, to the best of our knowledge, there does not exist in the literature any industrial evaluation for white-box system testing of microservices. This paper fills this important gap in the research literature.

Regarding industrial evaluations, there exist some recent work to conduct empirical studies on automated unit testing in industrial settings related to financial applications [24] and embedded systems [58]. Regarding system-level testing, the tool evaluations have been mainly investigated for user interface (UI) testing on ERP applications [59] and mobile applications [60, 61]. Other testing techniques, such as mutation testing [62, 63], fault debugging [64] and flaky tests [65] have also been evaluated in industry. However, none of such work provides the same type of contribution we give in this paper: an empirical study in which engineers and testers in industry apply a test generation tool on their industrial systems by themselves (without any intervention from researchers, using the publicly available documentation of the tool), and then evaluate the quality (e.g., readability) and benefits (e.g., code coverage and fault detection) of the obtained results (i.e., generated test cases) on their industrial systems, involving tens/hundreds of thousands of lines of code for the business logic.

There has been work on evaluating the readability of automatically generated tests. This has been the case for unit testing, with tools such as EvoSuite [66]. However, these studies usually involve only students (e.g., 30 students in [66]). As the subjects of these studies are single classes, which have a limited number of lines of code, this type of experiments on *unit testing* are feasible. Similarly, since 2023 the Java Test Case Generation Tool Competition<sup>12</sup> has started to use readability to score the competing tools. This was done by hiring external people (possibly engineers), which are not the authors of the tested classes. However, to the best of our knowledge, there is no work in the scientific literature that has studied the readability of automatically generated *system level tests*, in particular not when applied on large industrial systems.

In this article, we report on 3 user studies over 3 years, involving a total of 32 practitioners. In the software engineering research literature, there have been several

---

<sup>12</sup><https://sbft23.github.io/tools/java>

studies involving human subjects. An early survey [67] considering the years from 1993 to 2002, based on a selection of 5,453 articles from flagship software engineering research venues, shows that only 103 (i.e., less than 2%) studies included human subjects. Afterwards, Ko *et al.* [29] analyzed 1,701 articles published from 2001 to 2011 in flagship software engineering research venues. They showed that 345 of these articles (i.e., 20%) contain tool evaluations with human participants. When considering the time period from 2011 to 2018, based on a survey [68] of 1,584 articles published in flagship venues, the number of articles involving studies with human subjects was 397 (i.e., 25%). There is a clear increase of this type of studies throughout the years.

Some examples of this type of study include the work of Briand *et al.* [69], where they presented a user study on traceability involving 20 students. Scanniello *et al.* [70] reported on 4 controlled empirical studies (from 2011 to 2013), involving 88 participants. Of those, 25 were professionals in industry, whereas the others were students. The study involved “*comprehend a nontrivial chunk of an open-source software system*” [70]. Fraser *et al.* [71] investigated the utility of unit test generators with 2 studies involving 92 students and 5 industrial developers on 4 Java classes from open-source projects. Paulweber *et al.* [72] carried out a study on object-oriented abstractions with 98 students. Santos *et al.* [73] conducted a family of studies (8 with students and 4 in industry) on TDD for a total of 411 participants. Five “toy tasks” [73] were used for the experiments.

In the review from 1993 to 2002 [67], 72% of the analyzed articles exclusively involved students. Such percentage decreased to 23% in the 2001 to 2011 review [29] (where 56% included at least 1 professional, and the remaining did not provide details on the participants). Whether students or professionals in industry should be used for this type of study has been long debated. For example, to shed light on this issue, Salman *et al.* [26] ran an empirical study on TDD with 17 students and 24 professional on “toy tasks” [26]. In their experiments, “*neither of the subject groups performs better than the other when they apply a new technology during experimentation*” [26]. Falessi *et al.* [27] stated “*Using students as participants remains a valid simplification of reality needed in laboratory contexts*”. They carried out a survey among “experts” to study how the role of students and professionals in studies with human subjects in software engineering research is seen. Using students instead of professionals can simplify the complexity of running this kind of study. As such, they are more common among researchers. As stated by Vegas *et al.*, “*experiments in industry are thin on the ground. Of the few existing cases, most are 1-1 (running one experiment at one company), just a few are n-1 (running n experiments at one company) and still fewer are 1- n (running one and the same experiment at n companies)*” [74]. They reported on the difficulty of running experiments related to TDD in 3 different companies, including “*Professionals were troublesome, undermotivated, and performed worse than students*” [74]. However, although experiments were carried out in industry, the software subjects were green-field exercises (and not the industrial systems developed at those 3 companies).

Carrying out studies with human subjects is complex and time consuming. This can put off researchers from conducting this kind of research. For example, as stated by Ko *et al.*: “*Recent research has also shown that many software engineering researchers view this form of tool evaluation as too risky and too difficult to conduct, as they might ultimately lead to inconclusive or negative results*” [29]. To address this issue, Davis *et al.* [28] carried out interviews with 26 researchers to define a taxonomy of 18 barriers researchers encounter in this kind of study. They then proposed 23 solution strategies to

address 8 of those barriers.

What presented in this paper is significantly different from what reported in this section so far. None of the aforementioned work deal with *user studies with professionals in industry on tasks involving their industrial systems*. Some exceptions exist, like the study on Copilot made by the researchers at GitHub [75]. However, without a systematic review of the literature (e.g., the  $103 + 345 + 397 = 845$  articles identified in [67, 29, 68]) we cannot say for sure how rare such kind of study is.

## 4. Experiment Design

To investigate performance of our testing tool, EVOMASTER, we design our experiment by following evidence-based software engineering guidelines for conducting empirical studies on testing tools [76, 77, 78, 79, 80, 81].

### 4.1. Research Questions

To investigate EVOMASTER in industrial settings and understand industrial challenges in the testing of web services, we conducted an empirical study to answer the following research questions:

**RQ1:** How difficult is to set up a tool like EVOMASTER in real industrial settings?

**RQ1.1:** How difficult do industrial practitioners think is to use EVOMASTER?

**RQ1.2:** What challenges do exist in setting up EVOMASTER in industrial settings?

**RQ2:** How does EVOMASTER perform in real industrial settings?

**RQ2.1:** How does EVOMASTER perform on industrial SUTs in terms of line coverage and fault detection?

**RQ2.2:** How effective do industrial practitioners consider the automatically generated tests?

**RQ3:** How is EVOMASTER integrated into industrial development process?

**RQ3.1:** What are major barriers of adopting EVOMASTER from the viewpoint of industrial practitioners?

**RQ3.2:** What is the collaborative process of integrating EVOMASTER into industrial pipeline at our industrial partner?

**RQ3.3:** How would industrial practitioners like to use EVOMASTER?

**RQ4:** What are the most important testing challenges that practitioners meet in their testing context?

#### 4.2. Experimental Tasks

Our RQs are designed to investigate EVOMASTER in industrial contexts in terms of *usability* (RQ1), *effectiveness* (RQ2), *integration* (RQ3) and *existing challenges* that industry currently faces (RQ4). Table 1 outlines the tasks for answering each RQ. We carried out three user studies with industrial practitioners. Two of these studies (i.e., User Study 2021 and User Study 2023) were conducted in collaboration with our industrial partner, Meituan. The third user study (i.e., User Study 2024) involved practitioners from other five companies with whom we do not have existing collaborations, aiming to generalize findings obtained at Meituan.

Regarding tasks, as show in Table 1, assessments of *usability* (RQ1), *effectiveness* (RQ2), *integration* (RQ3) of EVOMASTER in industrial contexts were designed to consider perspectives of both researchers (RQ1.2, RQ2.1 and RQ3.2) and practitioners (RQ1.1, RQ2.2, RQ3.1 and RQ3.3). Benefiting from research-industry collaboration, as authors of academic prototype (i.e., researchers), we gain access to valuable information from industrial SUTs and have the opportunity for direct communication. More specifically, as the authors of EVOMASTER, we can check enterprise APIs chosen by our industrial partner and the quality of drivers to ensure that EVOMASTER was properly used for performing white-box testing on the enterprise APIs. With this opportunity, we shared our experience for answering RQ1.2. In addition, with information about the SUTs and direct feedback from industrial partner, it allows us to better analyze the line coverage and faults detected by EVOMASTER on industrial SUTs. This enables us to better understand EVOMASTER’s performance in real-world settings, as outlined in RQ2.1. Without such a collaboration, it would be difficult to perform such analyses. Therefore, this kind of analysis was excluded in User Study 2024.

Note that, in the first attempt to use EVOMASTER at Meituan, i.e., in User Study 2021, one employee involved in the task for RQ2.1 (see Table 1) together with the authors of EVOMASTER did manually review the tests generated by EVOMASTER and their coverage reports. This was done in order to identify the potential benefits EVOMASTER could offer. Such a review was also necessary to assess EVOMASTER’s fault detection capabilities to avoid issues of EVOMASTER on identifying faults of RPC APIs using domain knowledge of REST in User Study 2021. To minimize the efforts required from the employee, we selected one test suite for each SUT, opting for the test suite that achieved the best result for review. Moreover, this collaboration also enables us to discuss our perspectives in the collaborative process of integrating EVOMASTER into the industrial development process, addressing RQ3.2. Regarding assessments conducted from practitioners (RQ1.1, RQ2.2, RQ3.1 and RQ3.3), the analyses were performed based on responses of practitioners in questionnaires/interviews.

RQ4 (*existing challenges*) aims to study the challenges that practitioners face. It was conducted only from the practitioners’ perspective with questionnaires and interviews. Identified challenges can be further addressed in EVOMASTER and may also be useful for researchers in this field to better understand industrial problems.

#### 4.3. User Studies at Meituan

The two user studies at Meituan were performed at the starting point of the collaboration (i.e., User Study 2021) and at a time point when we addressed an important industrial problem specific to Meituan (i.e., User Study 2023), respectively. In the user

Table 1: Experimental tasks for each RQ

RQs	Tasks	User Study 2021 User Study 2023		User Study 2024
		Researchers	Practitioners (Meituan)	Practitioners (Others)
<i>RQ1.1</i>	Ask participants to follow the EvoMASTER documentation to fuzz services they are familiar with; then, collect their feedback based on their experience using EvoMASTER	×(2021) ×(2023)	✓(2021) ✓(2023)	✓
<i>RQ1.2</i>	Conduct a preliminary study of EvoMASTER on enterprise APIs and report researchers' experience of making the tool feasible for use in industrial setting of <b>industrial partner</b>	✓(2021) ✓(2023)	×(2021) ×(2023)	×
<i>RQ2.1</i>	Analyze line coverage and fault detection achieved by EvoMASTER on industrial SUTs <b>from industrial partner</b>	✓(2021) ✓(2023)	✓(2021) ×(2023)	×
<i>RQ2.2</i>	Collect participants' opinions on tests generated by EvoMASTER on industrial SUTs	×(2021) ×(2023)	✓(2021) ✓(2023)	✓
<i>RQ3.1</i>	Collect feedback for adopting EvoMASTER from participants' viewpoint	×(2021) ×(2023)	✓(2021) ×(2023)	✓
<i>RQ3.2</i>	Report researchers' experience of integrating EvoMASTER into <b>industrial partner's</b> development process	✓(2021) ✓(2023)	×(2021) ×(2023)	×
<i>RQ3.3</i>	Collect participants' preference for applying EvoMASTER in industrial development process	×(2021) ×(2023)	✓(2021) ✓(2023)	✓
<i>RQ4</i>	Collect participants' feedback on existing challenges they face	×(2021) ×(2023)	✓(2021) ✓(2023)	✓



studies at Meituan, our industrial partner managed the studies and decided which SUTs and participants to involve.

#### 4.3.1. SUTs

Meituan Select is a large-scale e-commerce platform for community group bulk buying (e.g., fresh vegetables, meat) that is a part of Meituan, and its products have covered 2 600 cities and counties in China. Products can be bought directly from farmers or distributors, and then be delivered to the community. With this e-commerce platform, its daily order volume is more than 30 millions. More than 630 million users are registered and use this e-commerce platform. The platform is constructed with hundreds of web services (referred as microservices) for, e.g., ordering, allocation, packing, delivery, and payments.

Table 2 presents detailed descriptive information of all five SUTs employed in the two user studies for studying effectiveness of EVOMASTER on enterprise APIs at Meituan, e.g., the number of Java class files, lines of codes and the number of exposed endpoints (i.e., methods that can be invoked via RPC). In 2021, we firstly conducted the user study with two of those services at Meituan, denoted as *CS1 2021* and *CS2 2021*. In 2023, the second study was conducted with three cases studies, denoted as *CS1 2023*, *CS2 2023* and *CS3 2023*. Note that *CS1 2021* and *CS1 2023* refer to the same system (i.e., *CS1*) but different versions as it evolved over the years, and the same applies to *CS2*, i.e., *CS2 2021* and *CS2 2023*. In 2023, we could not conduct the experiment with the SUTs using the same version as 2021, as these services (not only the two services under test but also all the interacted services and databases) have been significantly evolved since then to satisfy Meituan’s business requirements (see Table 2). #Services is the number of other services that the SUT directly interacts with (as *B* and *D* in Figure 2). Among them, #U is the number of its direct upstream services which the SUT depends on (e.g., *B*), and #D is the number of its direct downstream services, which call and use the SUT (e.g., *D*). Note that here we only report the number of services with direct communications. For instance, the upstream services of *SUT* have further upstream services, e.g., *A* for *B* in Figure 2.

Regarding the databases, in industrial settings where there is the need to scale to *hundreds of millions* of customers, the applied database is often distributed. At Meituan, this is achieved by sets of databases with an ad-hoc Relational Database Service (RDS), developed internally by Meituan to meet their scale needs. Connections between the SUT and databases are managed by sets of database middlewares. For instance, as shown in Figure 2, a connection between *SUT* and databases might be further distributed to multiple data sources (e.g., read, write) to different databases. In addition, the RDS audits and monitors all SQL commands executed on the databases. For example, in the first experiment, we found that **TRUNCATE** and **DROP** commands are forbidden from the web services, as the databases and tables are allowed to be created only manually through an audit process at Meituan. In Table 2, we also report the number of tables and the number of rows of data in the databases for all chosen web services.

All these web services chosen as SUTs in this paper are services whose previous versions are actively used in production at Meituan. Before running the experiments for this paper, no known bugs and faults were present (as all discovered bugs are promptly fixed).

Table 2: Descriptive statistics of industrial SUTs at Meituan

SUT	Original Type	Type of Fuzzing	#Scripts	File LOCs	#Endpoints	#Services (U, D)	#Tables	#RowsOfData
<i>CS1 2021</i>	RPC	REST	245	32,393	33	18 (14, 4)	142	256,024
<i>CS2 2021</i>	RPC	REST	98	12,152	13	11 ( 7, 4)	17	1,840
<i>CS1 2023</i>	RPC	RPC	421	57,209	62	27 (21, 6)	210	401,291
<i>CS2 2023</i>	RPC	RPC	185	26,353	20	33 (23, 10)	19	65,915
<i>CS3 2023</i>	RPC	RPC	308	193,024	63	24 (22, 2)	8	1,017,504
<i>Total</i>	-	-	1,257	321,131	190	113 (87, 26)	268	1,742,574

-Services(U,D) represents a number of services that the SUT directly interacts with, #U is a number of upstream services and #D is a number of downstream services; #RowsOfData is a number of rows of existing data in related tables.

#### 4.3.2. User Study 2021

The microservices at Meituan are built with Remote Procedure Call (RPC), a widely applied technique in industrial applications. To the best of our knowledge, there did not exist any tool which was available to enable automated testing for RPC APIs in 2021, when we first introduced EVOMASTER at Meituan (this can be regarded as one challenge identified). As REST API fuzzing and RPC API fuzzing share parts of domain knowledge in terms of Web APIs testing (such as database interactions) and white-box system level testing (such as code coverage and fault detection) and EVOMASTER has been evaluated as the most performing tool on open-source REST APIs [20, 7], our industrial partner considered that it was worth to apply EVOMASTER for assessing its performance in testing their services. A temporary solution to use EVOMASTER was to build an additional layer to adapt RPC APIs to REST APIs (i.e., implement a one-to-one mapping from an RPC function to a REST endpoint). However, implementing such a layer was straightforward, as discussed with our industrial partner. In industry, it is not uncommon to build a “proof-of-concept” to evaluate the feasibility of an approach, before investing a considerable amount of resources. For example, for a first validation it was easier and quicker to build a simple REST API layer on top of 2 existing APIs, instead of building a fuzzer for RPC APIs from scratch, as none existed in 2021. The former takes just a short amount of time (in the order of minutes, at most few hours), but has to be done for each single API, whereas the latter took several months, but needs to be done only once. Requiring to manually write REST layers is not a scalable solution, nor it is user-friendly compared to directly fuzz the RPC APIs. It was simply done just for experimentation sake. Therefore, we conducted the first empirical study of EVOMASTER in industry by fuzzing RPC APIs with REST’s strategies (see *Type of Fuzzing CS1 2021* and *CS2 2021* in Table 2) at Meituan in 2021.

#### 4.3.3. User Study 2023

As the potential benefits of EVOMASTER was demonstrated in the first user study, we continued our research with our industrial partner and addressed one of the most important challenges, i.e., fuzzing RPC APIs [25], in EVOMASTER v1.6.1 [82]. In such work [25], we provided the scientific contribution of designing the first white-box approach for fuzzing RPC APIs in the literature. Experiments were carried out on a series of APIs at Meituan, where performance and effectiveness were measured based on criteria such as code coverage and fault detection. However, such empirical study had no human subject. Its focus was on algorithmic analysis, and not on the human aspects of introducing and

applying these kinds of techniques in industry among practitioners. A main difference here, compared to [25], is that we employed questionnaire for assessing EVOMASTER with additional point of view from the industrial partner. Furthermore, the authors of EVOMASTER were not involved in running the tool (the participants had to do it by themselves, based only on the publicly available documentation of the tool).

Once the significant problem (i.e., fuzzing RPC) was addressed in EVOMASTER for Meituan, to perform a followup user study of EVOMASTER, we conducted the second user study at Meituan that applied the same questions as the study in 2021 for assessing usability and *effectiveness* of EVOMASTER v1.6.1, with three industrial APIs in 2023 (see *CS1 2023*, *CS2 2023*, and *CS3 2023* in Table 2). However, this second study is not a replication of the first study. The main difference is that in this second user study we used the novel introduced native support for RPC APIs in EVOMASTER. The generated tests call the RPC APIs directly. There was no need to create a REST layer anymore, and potential faults are identified based on the domain knowledge of RPC API rather than REST.

#### 4.3.4. EvoMaster Settings

To investigate effectiveness of EVOMASTER in the two user studies at Meituan, we conducted two experiments for fuzzing enterprise APIs using EVOMASTER v1.3.0 (discussed in Section 4.3.2) and EVOMASTER v1.6.1 (discussed in Section 4.3.3) respectively. In the User Study 2021, as the first attempt to apply EVOMASTER, the experiment was ran on a local machine of an employee at Meituan, thus, we can only conduct it once due to such resource constraints. In the User Study 2023, as EVOMASTER is integrated into the testing pipelines at Meituan, we were able to repeat the second experiment multiple times, i.e., repeated 10 times.

Regarding time budget settings, as discussed with the industrial partner, testing performed on their CI pipeline typically takes a couple of minutes. From a point of view of developers, since they might wait for implementing other tasks, the time cost for the fuzzer is preferably less than 1 hour. However, as a search-based method, the specified search budget in EVOMASTER would have a major impact on its performance. With a consideration on time constraints in industry, therefore, we set three stopping criteria (i.e., 30 minutes, 1 hour and 10 hours) in these experiments. Note: this is rather different from the typical amount of 24 hours used in fuzzing literature. Hence, the two experiments of running EVOMASTER on the five SUTs took 368 hours, i.e.,  $(0.5h + 1h + 10h) \times 2APIs \times 1 + (0.5h + 1h + 10h) \times 3APIs \times 10$ , which took more than 15 days.

#### 4.3.5. Industrial Participants

In the two user studies, we involved eight employees at Meituan in 2021 and 19 employees at Meituan in 2023. As a result of the collaboration, employees at Meituan were allowed to participate in these user studies during their working hours.

Table 3 represents information about all these industrial participants, i.e., positions (see *Position*), years of working in testing in industry (see *#Years*), and the size of group the participant leads (see *#Group*). Note that terms in *Position* are based on the job title terminology used internally at Meituan. The roles of all participants vary over seven different positions, which are responsible for various tasks, such as testing, development and product. This could provide diverse viewpoints to this industrial evaluation. More specifically, the *Director of Software Quality (DSQ)* participant is in charge of all testing

Table 3: Description of Industrial Participants at Meituan

Position	#Par User Study 2021	#Par User Study 2023	#Years	#Group	Abb.
<i>Director of Software Quality</i>	1	0	15+	200+	DSQ
<i>Principal Software Engineer</i>	1	0	11-12	30-50	PSE
<i>Quality Assurance (QA) Manager</i>	3	3	8-10	10-25	QAM
<i>Quality Assurance (QA) Engineer</i>	3	10	0-10	-	QAE
<i>Product Manager</i>	0	1	11	10	PDM
<i>Development Manager</i>	0	2	10-15	10-35	DPM
<i>Developer</i>	0	3	2-6	-	DPE
<i>Total</i>	8	19			

#Par represents the number of participants in the experiments in either 2021 or 2023, #Years represents years of working in testing in industry, and #Group is a number of members in the group the participant leads. “-” represents not applicable.

departments at Meituan Select. He is involved in the interview study mainly to provide viewpoints to challenges relating to their business scope. The *Principal Software Engineer* (PSE) participant is the department manger of internal testing tool development at Meituan Select, who can share opinions from the standpoint of applicability and further integration of the fuzzer and challenges they meet in order to enable testing of industrial services. The *Quality Assurance Manager* (QAM) participants manage testing tasks and are responsible for various specific services, such as services for inventory management at warehouse and payment. They could give us feedback on preference of applying EVOMASTER to their testing tasks, and share their difficulties/experience in managing testing tasks for various kinds of industrial services. The *Quality Assurance Engineer* (QAE) participants were involved based on their familiarity with the selected SUTs. They mainly handle testing tasks for specific services, such as write test cases and perform manual testing. Thus, they could give feedback on the generated tests and share their experience in daily tasks which fuzzers like EVOMASTER aim to automate. Besides feedback from practitioners who work in software testing, we also involved *Product Manager* (PDM), *Development Manager* (DPM) and *Developer* (DPE) at Meituan for assessing EVOMASTER from different viewpoints.

#### 4.4. User Study Outside of Meituan

To generalize the results obtained at Meituan to other enterprises, we conducted the third user study in 2024 outside of Meituan (User Study 2024) with EVOMASTER v2.0.0. Note that there is no difference in terms of capability of fuzzing Web APIs between v2.0.0 and v1.6.1. Since we do not have other collaborations with additional companies, one of the authors contacted few acquaintances in their professional network in China. Five responded positively, willing to take some of their time to participate in this user study. Then, the third study was carried out with five practitioners from five different Chinese enterprises by online questionnaire. In the questionnaire, we included additional questions to request permissions to collect the name of their company, information about the APIs they employed to run EVOMASTER and results. Only one out of the five participants granted permission to disclose the company name, called *Hebei Happy Consumer Finance Co., Ltd.* that mainly engages in financial-related business, such as offering loans to consumers. Unfortunately, none of the participants granted permission to share the information about their APIs and the results outputted by EVOMASTER. Therefore, we cannot report detailed statistics in this paper, and exclude the SUTs in analyzing

results of EVOMASTER from researchers’ perspectives (i.e., RQ1.2, RQ2.1 and RQ3.2). In addition, since we do not have collaboration agreements with these companies, it is not feasible to ask participants to conduct the experiment as what we did at Meituan, e.g., fuzz APIs with 10 hours’ time budget. Thus, in User Study 2024, the participants themselves can decide which SUT to use and what time budgets to set.

Regarding the participants, based on information they provided, the five practitioners have different job roles, i.e., 1 Development Manager (DPM), 1 Developer (DPE), 1 Software Architect (SWA), and 2 Testers/Quality Assurance Engineers (Tester/QAE).

#### 4.5. Question Design for Questionnaire/Interview

One of the main tasks in an industrial evaluation is to collect feedback from the industry practitioners (all tasks relating to practitioners’ perspectives in Table 1). In software engineering research, typical methods to collect such feedback are based on questionnaires and interviews conducted with employees at these companies [83, 84]. In this study, to empirically assess the fuzzer EVOMASTER and evaluate its further potential integration in industrial settings, we designed a set of questions for the questionnaire and interviews that collect responses from industrial participants in terms of *usability* (for RQ1.1), *effectiveness* (for RQ2.2), *integration* (for RQ3) and *existing challenges* (for RQ4) in the three user studies. All questions we designed for the three user studies are represented in Table 4, and the number of participants for each question is shown in Table 5. As seen <sup>†</sup> from Table 5, the interviews were performed only with *RQ3* and *RQ4* that involved 1 DSQ, 1 PSE and 2 QAM. 5-Point likert scale [85] is one of the typically methods to measure perceptions and opinions of humans in questionnaires and surveys, and it has been applied in various studies in software engineering [24, 86, 87]. In this study, we employ a 5-Point likert scale [85] to collect participants’ feedback on assessing difficulty of using EVOMASTER (*QA1-5*), readability (*QB1*) and quality (*QB2*) of tests generated by EVOMASTER, and participants’ familiarity on the SUT (*PR1*). Such questions were not used in the interviews.

##### 4.5.1. Usability

To evaluate the usability of EVOMASTER, we defined *QA* questions according to steps of using EVOMASTER as shown in Table 4. On EVOMASTER website<sup>13</sup>, documentation and examples including some training videos are provided about how to get started with EVOMASTER. This includes the following five steps.

- *Set up*: there are three options to set up EVOMASTER, i.e., use a released runnable jar file, install it with installers for Windows/OSX/Linux since version 1.2.0, or build it from source code;
- *Resolve dependencies*: it might need to resolve conflicts of dependencies (such as different versions of used libraries) between EVOMASTER and SUT;
- *Write driver*: to use EVOMASTER in white-box mode, it requires to write a driver for extending pre-defined methods which enables starting/stopping/resetting of the SUT programmatically (see Figure 3);

---

<sup>13</sup>[www.evomaster.org](http://www.evomaster.org)

Table 4: Questions in questionnaire and interview.

	#	Questions
Usability	<i>QA1-5</i>	How difficult was it for you to set up EVO MASTER (QA1), resolve dependencies (QA2), write driver (QA3), run EVO MASTER (QA4), and execute tests(QA5)? [ ] <i>Very Difficult</i> [ ] <i>Difficult</i> [ ] <i>Moderate</i> [ ] <i>Easy</i> [ ] <i>Very Easy</i>
	<i>QA6-10</i>	How much time did it take you to set up EVO MASTER (QA6), resolve dependencies (QA7), write driver (QA8), run EVO MASTER (QA9), and execute tests(QA10)?
Effective -ness	<i>PR1</i>	How familiar are you with the web service used in the study? [ ] <i>1-Very Unfamiliar</i> [ ] <i>2-Unfamiliar</i> [ ] <i>3-Moderate</i> [ ] <i>4-Familiar</i> [ ] <i>5-Very Familiar</i>
	<i>PR2</i>	What is the most important feature that you would like to have in the generated test cases? [ ] <i>Achieved code coverage</i> [ ] <i>Detected faults</i> [ ] <i>Readability</i> [ ] <i>Other</i> __
	<i>PR3</i>	What properties are important for you to rate readability? (such as size of test cases, name of tests, grouped tests into different files, etc.)
	<i>PR4</i>	What properties are important for you to rate quality? (such as code coverage, automatically test the web services regarding input validation, scenarios (e.g., invalid inputs, business logic) to be tested automatically, etc.)
	<i>PR5</i>	Before answering the following questions, how much time did it take you to read and understand the tests generated by EVO MASTER?
	<i>PR6</i>	Did you have as much time as you needed to inspect all the test cases in order to answer these questions? [ ] <i>Yes</i> [ ] <i>No</i>
	<i>QB1</i>	How would you like to rate the readability of the generated tests? [ ] <i>Very Low</i> [ ] <i>Low</i> [ ] <i>Moderate</i> [ ] <i>High</i> [ ] <i>Very High</i>
	<i>QB2</i>	How would you like to rate the quality of the generated tests? [ ] <i>Very Low</i> [ ] <i>Low</i> [ ] <i>Moderate</i> [ ] <i>High</i> [ ] <i>Very High</i>
	<i>QB3</i>	How can the generated tests be improved?
	<i>QB4</i>	Describe what you like better about manually written tests than generated tests?
	<i>QB5</i>	Would you keep the generated system-level tests? [ ] <i>Yes</i> [ ] <i>No</i>
	<i>QB6</i>	If yes, how would you like to keep and use them?
	<i>QB7</i>	If you are using EVO MASTER, for how long (eg, minutes, hours) do you typically run it for generating test cases? (Default <code>--maxTime</code> option is 1 minute)
	<i>QB8</i>	Based on results achieved by EVO MASTER, which option of time-cost do you prefer based on the trade-off with the achieved code coverage? [ ] <i>30 minutes</i> [ ] <i>1 hour</i> [ ] <i>10 hours</i>
Integration	<i>QC1</i>	What are the major barriers from your point of view in adopting the EVO MASTER tool?
	<i>QC2</i>	Given your current infrastructure setup, how would you like to have automated system-level test generation framework integrated?
Existing Challenges	<i>QD1</i>	What is one of your current major issues/time consuming activity with manual testing that you would like to have automation for?
	<i>QD2</i>	What kinds of faults are harder to detect in the system?
	<i>QD3</i>	What are the most important challenges that you meet in testing?

Table 5: Regarding each question in questionnaire and interview, number of participants for answering each question in each user study. Note that “-” indicates that the question is not used in the user study. The symbol <sup>†</sup> indicates that participants answered the questions through interviews.

	#Question	#Total of Par.	#Participants per User Study		
			2021	2023	2024
Usability	<i>QA1-5</i>	16	<b>1</b> (1QAM)	<b>10</b> (5QAE, 2QAM, 2DPE, 1DPM)	<b>5</b> (1DPE, 1DPM, 1SWA, 2Tester/QAE)
	<i>QA6-10</i>	15	-		
Effective-ness	<i>PR1-6</i>	20	-	<b>15</b> (8QAE, 3QAM, 2DPE, 1DPM, 1PDM)	
	<i>QB1-6</i>	25	<b>5</b> (3QAE, 2QAM)		
	<i>QB7-8</i>	20	-		
Integration	<i>QC1</i>	13	<b>8</b>	-	
	<i>QC2</i>	32	(3QAE, 1QAM, 1 <sup>†</sup> DSQ, 1 <sup>†</sup> PSE, 2 <sup>†</sup> QAM)	<b>19</b> (10QAE, 3QAM, 3DPE, 2DPM, 1PDM)	
Existing Challenges	<i>QD1-3</i>	32			
# Answers		$(QAs: 5 \times 16 + 5 \times 15) + (PRs: 6 \times 20) + (QBs: 6 \times 25 + 2 \times 20) + (QCs: 13 + 32) + (QDs: 3 \times 32) = 606$			

- *Run* EVOMASTER: start the fuzzer from command line for automatically generating tests for the SUT;
- *Execute tests*: execute generated tests on the SUT.

Given such documentation, we asked the industrial participants to use the fuzzer with their services, then collect their feedback.

In 2021, at the starting phase of collaboration with Meituan, due to limited resources, EVOMASTER was assessed by one participant who provided his/her ratings of difficulty for the five steps of using EVOMASTER (see *QA1-5* in Table 4). In 2023, with more resources from Meituan allocated to the collaboration, the usability assessment in the User Study 2023 was conducted with 10 participants. In addition, we defined five new questions (i.e., *QA6-10* in Table 4) to assess the time cost of using EVOMASTER according to the five steps. In User Study 2024, all *QAs* were employed.

#### 4.5.2. Effectiveness

To evaluate effectiveness of EVOMASTER from the point of view of industrial practitioners, we carried out experiments of fuzzing enterprise APIs with EVOMASTER, and then conducted a questionnaire about generated tests, given to industrial participants.

The questions were designed to collect feedback on the tests regarding their readability, their quality, how they can be improved and whether they keep them as *QBs* shown in Table 4. Given the results on the effectiveness of EVOMASTER, it is also interesting to investigate practitioners’ preferences regarding time cost settings. Then, in User Study 2023 with EVOMASTER v1.6.1, we designed two further questions (i.e., *QB7* and *QB8*) to collect responses related to time budget that practitioners would like to use.

In the second user study, we further designed a set of pre-check questions, as *PRs* shown in Table 4. The goal was to obtain information regarding their familiarity on the SUT they analyze (*PR1*), preference to assess the generated tests (*PR2-4*), time spent on performing this experiment and whether they had enough time (*PR5-6*). With the three SUTs applied in 2023, for each one, we involved five employees at Meituan (in total

15 participants for all three SUTs). To better assess the generated tests, ideally there would need to involve more people who are familiar with the SUT to test. However, in industry, services are typically developed by different groups, and, in quality assurance departments, they typically divide tasks among different employees bases on the services. Then, there might not be many employees who are familiar with the same SUT. The five employees per SUT is the maximum that Meituan could reach, without having to involve employees that are not familiar with those APIs.

In User Study 2024, the same questions as User Study 2023 were asked to the five participants.

#### 4.5.3. Integration

We defined two questions (i.e., *QC1* and *QC2*) in Table 4 in our questionnaire and interview for inquiring potential integration and usage of EVOMASTER in industrial settings.

*QC1* relates to barriers in adopting EVOMASTER that was involved in User Study 2021 and User Study 2024. We executed it in User Study 2023, because EVOMASTER has been integrated into the testing pipelines at Meituan. In User Study 2024, we included *QC1* to collect the feedback from practitioners outside of Meituan. Regarding *QC2*, we asked it in all of the three user studies (see Table 5).

#### 4.5.4. Existing Challenges

The research task for RQ4 aims at understanding existing testing difficulties in industry, and discuss potential solutions that tools like EVOMASTER could help to tackle. To achieve this goal, we defined three questions (i.e., *QD1-3*) as shown in Table 4 and asked them to all industrial participants in the three user studies (see Table 5).

## 5. Experiment Results

### 5.1. Results for RQ1: Usability

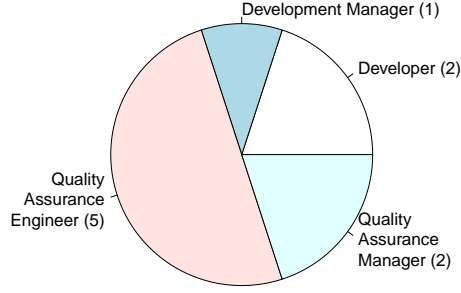
The goal of this RQ1 is to evaluate the usability of EVOMASTER in industrial settings in the view of industrial participants (i.e., RQ1.1) and discuss challenges we faced (i.e., RQ1.2).

#### 5.1.1. Results for RQ1.1

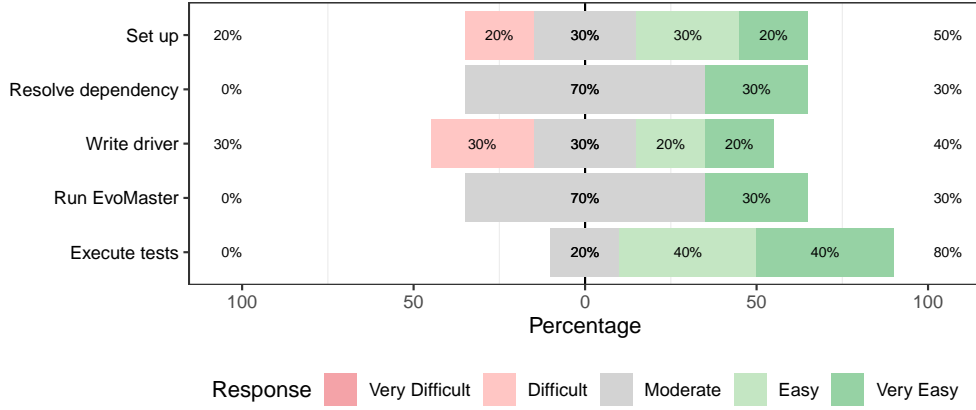
**Results at Meituan.** In the User Study 2021, one employee at Meituan (i.e., 1 QAM) applied EVOMASTER to test enterprise RPC APIs accessed with exposed REST endpoints. His/her responses for *QA1-5* on difficulty rates of using EVOMASTER are *Easy* for setting up, writing driver, running the tool and executing tests, and *Moderate* for resolving dependencies.

In 2023, as the version v1.6.1 supported RPC API fuzzing, we conducted the User Study 2023 for assessing its usability with a native support for fuzzing enterprise RPC APIs. The User Study 2023 involved 10 participants at Meituan, and positions of the participants are shown in Figure 4a. Results of answers as the same set of questions in 2021 (i.e., *QA1-5*) are shown in Figure 4b, and answers of new questions in terms of time cost the participants spent to use EVOMASTER (i.e., *QA6-10*) are shown in Table 6.





(a) Participants in 2023



(b) Diverging stacked barplot for responses of difficulty rates using 5-points likert-scale collected in 2023. Note that positive responses (i.e., “easy”) extending to the right and negative responses (i.e., “difficult”) to the left from the diverging point.

Figure 4: Answers provided by industrial partners about the difficulties on applying EvoMASTER (QAs)

Table 6: Time (minutes) spent by participants for applying EvoMASTER in 2023

Position	Setup	Resolve dependency	Write driver	Run EvoMASTER	Execute tests	Total
DPE	180	30	60	30	<u>180</u>	480
DPE	20	<b>1</b>	<b>3</b>	10	5	<b>39</b>
DPM	20	2	5	10	<b>2</b>	<b>39</b>
QAE	240	60	60	30	120	510
QAE	240	60	60	30	120	510
QAE	<u>300</u>	90	<u>120</u>	<u>90</u>	120	<u>720</u>
QAE	120	30	30	30	30	240
QAE	120	20	60	60	30	290
QAM	120	5	60	30	5	220
QAM	<b>15</b>	<u>180</u>	20	<b>3</b>	10	228
Median	120	30	60	30	30	265

Based on responses of difficulty rates, in general, participants did not meet much difficulty in applying the tool and analyze its outputs. Within these two experiments, none of the participants rated *Very Difficult* in any of the five steps. In the User Study 2021, the participant met some dependency conflicts when using EVOMASTER client library. This required a manual fix, and therefore the participant rated this task as *Moderate*. In the User Study 2023, with 10 participants on five tasks (in total 50 rate answers), there are two *Difficult* rates on setup and three *Difficult* rates on writing driver. For the rest of the answers, they are either neutral (i.e., *Moderate*) or positive (i.e., *Easy* or *Very Easy*). Regarding setting up EVOMASTER, two participants met the same difficulty when building it from source code as they needed to solve versions of tools and dependencies, such as Kotlin, Java 8/17 and Maven. Regarding writing driver, its rate might depend on familiarity of participants on the implementation of the SUT. For instance, three participants (all *QA Engineers*) considered that it is *Difficult* to implement methods for starting/stopping/resetting the SUT programmatically, and we received two *Very Easy* answers from a *Development Manager* and a *Developer*. If the participants (such as QA Engineer) do not need to write the code for programmatically handling the SUT (e.g., write unit tests or manual tests), they might meet some difficulties to figure out how to do that. Regarding rates of resolving dependencies and running EVOMASTER, 70% of participants considered it *Moderate* and 30% of participants considered it *Very Easy*. Executing tests seems to be evaluated as the easiest task as 80% participants rated it either *Easy* or *Very Easy*, where only 20% participants rated it *Moderate*.

Table 6 presents the time cost spent at each step per participant. The time cost varies from participant to participant, and there are various factors which can impact results, e.g., their daily task and proficiency relating to techniques that EVOMASTER employs or needs. Those who spent less time (e.g., DPE and DPM took 39 minutes) might be familiar with tasks such as resolving dependencies with Maven (Resolve dependency), starting/stopping/resetting APIs programmatically (Write driver), starting the tool with the command line (Run EVOMASTER), and executing JUnit tests independently of their testing framework (Execute tests). The median cost is 265 minutes. For instance, the minimum time cost of applying EVOMASTER and executing generated tests is 39 minutes, while the maximum time cost is 720 minutes (i.e., 12 hours  $\approx$  2 working days). Based on median time cost for the tasks, *resolving dependencies*, *running the tool* and *executing tests* are tasks which take less time (i.e., 30 minutes), while setting up EVOMASTER needs more time to figure it out (i.e., 2 hours).

Note that the time values reported in Table 6 are *self-reported*. Getting exact time measures would have significantly complicated the study, as there would have been the need to monitor each of the participant’s laptop/PC. And those machines are locked-down to protect the IP of Meituan (e.g., source-code) from cyber threats (i.e., we would not have been able to install any monitoring software on such machines, as that would have been a security vulnerability). As the exact time values are not critical, and approximate self-reported values could be enough to get insight into this problem, we considered such an approach as a reasonable compromise.

Regarding tool setup, in the two user studies, we found out that surprisingly all participants did set up EVOMASTER by building it from source code. The choice of setup might relate to documentation, that might lack step-by-step guideline to get started with EVOMASTER. In addition, the documentation of EVOMASTER is currently written only in English. As all participants are not native English speakers, then they might not take

full advantage of all installation information when they get started to use the tool (e.g., video tutorials).

**Results outside of Meituan.** Out of the five practitioners who participated in the User Study 2024, only one (i.e., Software Architect (SWA)) successfully completed all five steps and was able to use EVOMASTER for test generation, while the other four did not manage to write a driver within their free time. Answers provided by the one who completed all steps are:

- difficulty rates: *Moderate* for setting up, resolving dependencies, running the tool and executing tests, and *Difficult* for writing driver;
- time cost: 10 minutes for setting up, 5 minutes for resolving dependencies, 120 minutes for writing driver, 10 minutes for running the tool, and 1 minute for executing tests.

Results indicate that among the tasks required for utilizing EVOMASTER, writing drivers to enable white-box testing appears to be the most challenging. Compared to results obtained at Meituan, writing drivers also received the most *Difficult* rates (see Figure 4b), and it stands out as the second most time-consuming task for participants (see Table 6).

**Discussion.** These results provide some interesting insight and reflections. First, who are the target users for these fuzzers in industry? Developers and testers are likely candidates, but their background profiles can be very different. A software developer would have the coding skills to setup the drivers to enable white-box testing, whereas a tester might rarely require such skills in their daily work, or require significantly more time. For these latter, a black-box approach that requires no coding setup might be more effective, as easier to use. Even if white-box testing can achieve significantly better results (i.e., higher code coverage and fault detection), there is still an important role left for black-box testing. Notice that, at the time of this writing, for RPC APIs we only support white-box testing in EVOMASTER.

Another interesting aspect is regarding possible language barriers. The de-facto language of science is English, and most software developers in the world have to deal with the English languages on a daily base (e.g., most programming languages are text based, and use keywords from the English language). This can create a bubble, where academics could wrongly assume that the output of their work written in English would be understandable to practitioners in industry at large, and not possibly just to a minority that can read and understand English. It is not in the scope of this paper to do a full analysis of language barriers in the software industry around the world, but it is clearly an issue that we experienced in this work, especially when dealing with “non-coders”.

Research prototypes are already seldom documented. Providing documentation and user manuals in different languages (e.g., Mandarin and Spanish) would not be viable in most cases. However, that could be yet another barrier for technology transfer from academic research to industrial practice.

Based on results of usability obtained in the three user studies, writing drivers appears to be the most challenging task. The difficulty was more noticeable among participants outside of Meituan, i.e., four out of five participants failed to manage to write the driver. Writing drivers requires participants to have programming skills and be familiar with tasks such as using third-party libraries, configuring schemas to access endpoints, and

programmatically starting, stopping, and resetting APIs. These capabilities, however, are often not part of their day-to-day responsibilities, as such tasks are typically handled by testing toolkits that are closely integrated with their own development frameworks. Although we provided examples, documentation, and videos to demonstrate how to configure drivers for APIs built with Spring Boot<sup>14</sup>, different companies may use their own frameworks. In addition, it takes time to fully understand the examples, documentation, and videos on their own. This may be especially challenging for participants outside of Meituan, as they completed the questionnaire in their free time. To improve the usability of EVOMASTER, the task should be simplified, and better documentation (e.g., better tutorials) should be provided. It is also worth mentioning that, our industrial partner now have automated the five steps of applying it on their services and integrated EVOMASTER into their testing pipelines. Industrial APIs are often built with the same pattern and use dependencies from their internal repository, thus, code for writing the driver (such as starting, stopping and resetting) and configuration of resolving dependencies can be templated. In their testing pipeline, EVOMASTER is pre-set with a specific version, then it could be further used for fuzzing industrial APIs automatically. Settings (such as choice of search budget and using specific algorithm options) for generating tests with EVOMASTER can be configured easily with a script, and our industrial partner uses the same setting for all its APIs.

In the User Study 2024, we received fewer valid responses (i.e.,  $1/5 = 20\%$ ) compared to valid responses obtained at Meituan (i.e.,  $11/11 = 100\%$ ). This discrepancy might depend on the context, i.e., if the research collaboration exists. There, the user studies at Meituan were approved by higher management, and participants worked on it during their working hours. In monetary value, that was a non-negligible cost (e.g., the salary of the employees). This was not the case in these other five enterprises, where there is currently no research collaboration with the authors of EVOMASTER. Collaborations between academia and industry are valuable, but hard to setup and maintain [88, 21].

Learning to use a new tool can take time. Once got familiar with it, each following use would be easier and less time consuming. However, for practitioners there is the need to see some direct benefits in it before being willing to invest time to learn how to use such a new tool, or build a customized infrastructure to ease its use on their systems (as done at Meituan).

**RQ1.1:** *To apply EVOMASTER at Meituan, based on difficulty rates from 11 participants on five steps, it is likely not problematic to use EVOMASTER as we received 0 very difficult rate, 5 difficult rates (9%), 23 moderate rates (42%) and 27 easy/very easy rates (49%), and the median time cost to apply the tool with the five steps for the first time is 265 minutes. In the user study outside of Meituan involving five participants from different companies, we observed that writing drivers appears to be the most challenging task, with four out of five participants struggling to use EVOMASTER and failing to complete the task. Similarly, this task was also rated as the most Difficult in user studies conducted at Meituan. To enhance usability, it requires simplification and improved documentation. As the industrial APIs are often built with same pattern, applying EVOMASTER has been automated, and now EVOMASTER has been integrated into the development pipelines at Meituan.*

<sup>14</sup>[https://github.com/WebFuzzing/EvoMaster/blob/master/docs/write\\_driver.md](https://github.com/WebFuzzing/EvoMaster/blob/master/docs/write_driver.md)

### 5.1.2. Results for RQ1.2

EVOMASTER can be applied for automating test case generation for industrial APIs. In order to better support such automation, there still exist some challenges which require both researchers and industrial partners to address. With the first experiment using EVOMASTER v1.3.0 in 2021, we found that, since EVOMASTER mainly addresses REST APIs and did not directly support any RPC mechanism (e.g., such as Apache Thrift) yet, in order to employ EVOMASTER on their services, our industrial partner needed to implement an additional REST layer. This was implemented based on one-to-one mapping from each RPC function to a REST endpoint. Building such a layer was rather straightforward. However, the manual work with the additional layer might bring new problems, and without a native RPC support, the effectiveness of EVOMASTER might be limited (e.g., strategies specific to REST domain might not work for the services developed with RPC). Therefore, we developed a native white-box fuzzing for RPC APIs [25] (supported in EVOMASTER v1.6.1 [82]) that is currently deployed in the testing pipelines at Meituan.

Another challenge relates to reset the state of SUT in industrial setting. In order to generate independent tests, EVOMASTER requires a proper reset of the SUT, e.g., clean modified data in databases, reset/manipulate states of related services. The industrial APIs often are parts of a large microservice architecture, and they require to interact with databases and other external services (see Figure 2). At Meituan, they had deployed a specific testing environment where all services are up and running (which is a typical practice in industry). Within this environment, a service under test is able to interact with all its required services (e.g., external services and databases shown in Figure 2). With such a real industrial setting, it is not trivial to reset the state of the SUT. First, the interacted external services are not controllable from the test cases, which means that we cannot manipulate them to specific states before executing a new test. Besides, the databases in real practice are more restricted and complex. For instance, in the first experiment, we found that some types of SQL commands are blocked (e.g., `TRUNCATE`), then it is not applicable to reset the databases using SQL commands, such as a `DbCleaner` utility provided in EVOMASTER to clean data in directly connected SQL databases. Once the SQL command restriction issue was identified, we discussed with our industrial partner, and they implemented a solution to provide us access to execute SQL commands. But a challenge we faced later is how to handle a large amount of data in the databases. In industry, to better test their services, they often prepare data in the database, and the amount of data can be very large from the point of view of testing (e.g., 1,017,504 rows of data in *CS3 2023* shown in Table 2), which could be created manually or collected during production. Such data is maintained and shared to all development and testing groups. To manipulate various states of the SUT, EVOMASTER integrates SQL handling to insert data into the databases with defined SQL heuristics for test generations [41]. Considering the large amount of data in the databases, it is not viable to reset them (i.e., clean and then re-add) before executing every test, as it would become a drastic performance bottleneck. However, if we do not clean the data, the newly inserted data might affect their testing environment. In a scientific research context, our industrial partner could help us to isolate a testing environment with an empty database (the empty database can significantly reduce the time cost of the reset), as we did in [25]. It is not a feasible solution for our industrial partner when they apply EVOMASTER

in their real development/testing process. For instance, isolating a complete testing environment is time-consuming, and it is impractical to perform the operation every time when applying EVOMASTER. In addition, with enabled SQL handling, the tests generated by EVOMASTER would have INSERT SQL commands and perform the reset that likely introduces problems into their testing environment. As discussed with our industrial partner, without yet a viable solution to handle such large amount of data, we decided to disable the reset and SQL handling in the real application of EVOMASTER at Meituan.

In the second experiment we conducted in 2023, EVOMASTER was configured with the settings used by our industrial partner, i.e., without the reset and SQL handling. To avoid conflicts with other teams which might test the same services, our industrial partner isolated a testing environment where all connected services are up and running, and databases have the same data as the main testing environment for running EVOMASTER. In this isolated environment, the databases and services (the SUT and its connected services) cannot be accessed by other teams. Thus, the tests generated by EVOMASTER could be further used in the main testing environment. Note that there might be side-effects when disabling the database reset, e.g., a sequence of executing tests might affect the results of other tests. However, such possible side-effect was deemed acceptable by our industrial partner.

Furthermore, we found that, besides authentication restrictions, there also exist some restrictions based on intra-service business logic. For instance, an endpoint in a SUT  $X$  might only accept requests which contain specific information, and the specific information could be put by an endpoint in another service  $Y$ . Thus, in order to access the endpoint in the SUT  $X$ , the endpoint in  $Y$  must be invoked first, or the request must be sent by the endpoint in  $Y$ . Such information among services are handled in an internal *Tracer* service developed by Meituan. Thus, in order to access the endpoints in testing, we extended EVOMASTER for providing a method for users to define pre-actions to resolve such constraints, and the invocation of the pre-actions would be handled with a probability as authentication in EVOMASTER, i.e., a probability to control whether the request contains such required information. The authentication is currently configured automatically in the driver generated by Meituan, using a default setting.

**RQ1.2:** *The challenge of having a native support for fuzzing RPC APIs has been addressed with the latest version of EVOMASTER. However, due to complex interactions with databases and external services, there exist challenges in having an effective solution to properly reset the state of the SUT and enable SQL handling.*

## 5.2. Results for RQ2 : Effectiveness

Based on results obtained by EVOMASTER on industrial APIs, the effectiveness of EVOMASTER was firstly analyzed by the authors of EVOMASTER (Section 5.2.1). The analyses were performed to investigate the performance of EVOMASTER on the industrial APIs with various time budget settings, and to assess the line coverage and fault detection achieved by EVOMASTER on the given industrial APIs. However, for the User Study of 2024, due to the lack of collaborative agreements with the enterprise, we do not have access to the code and results, and therefore, cannot perform the analyses.

The effectiveness of EVOMASTER was also assessed by practitioners with tests generated by EVOMASTER in terms of readability and quality (Section 5.2.2). In the user studies at Meituan, we provided the tests which achieved the best results for each SUT

and results of the analyses (such as line coverage and number of potential faults obtained by EVOMASTER with each time budget) to participants for conducting the assessment. For the user study outside of Meituan, participants can select the tests by themselves for the assessment.

### 5.2.1. Results for RQ2.1

**Performance of EVOMASTER on industrial APIs with various time budget settings.** EVOMASTER identifies various metrics related to testing criteria as testing targets<sup>15</sup>, such as code coverage (e.g., class, method, branch, statement), identified faults and domain specific targets (e.g., different execution results for RPC), and produces tests in order to maximize the number of targets being covered. To investigate performance of EVOMASTER with various time budget setting, Figure 5 plots the number of testing targets covered at every 5% budget used by the search for the five applied SUTs, with search budgets of 30 minutes, 1 hour and 10 hours. Note that, as controls for *exploration* and *exploitation* in EVOMASTER (see Section 2.2.1), the results achieved with 30 minutes and 1 hour are not comparable to the first 5% and 10% of the 10-hour search.

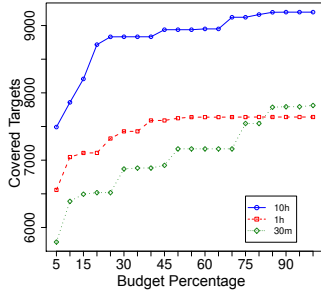
Based on the figures, for the SUTs, results achieved by 10 hours clearly outperform results with 30 minutes and 1 hour. Comparing results between 30 minutes and 1 hour, their performances are close except *CS1 2021* (see Figure 5c). In *CS1 2021*, the budget with 30 minutes achieves even better results than the 1-hour budget. Considering the random nature of search algorithms, it could happen by chance as the experiment on *CS1 2021* was only performed once. But it also reveals that probably there is a lack of exploration of the search landscape with the small budget.

Applying search for many objectives such as test generation, it is essential to explore the search landscape at early stages for providing diversification of individuals. Then, such individuals could enable a further possibility to cover new targets, e.g., code coverage and fault finding in our context, in the later exploitation stage. Therefore, with automated testing approach such as EVOMASTER, a longer budget would be required, i.e., more than 1 hour, to achieve better results. However, a too long search budget (e.g., 24 hours) might be not viable, if engineers are not willing to wait so long to get results. But this also strongly depends on whether tools like EVOMASTER would be typically run on a developer machine, or on a remote dedicated CI server. Also, it depends on whether the fuzzing is done during a regular development day, or before a major software release (this latter would like entail spending a longer time for testing).

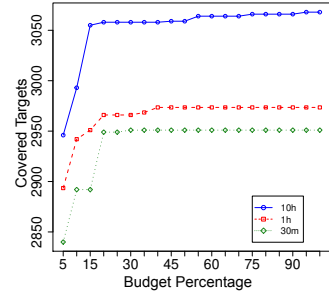
**Line coverage achieved by EVOMASTER on industrial APIs.** Table 7 represents line coverage achieved by two versions of EVOMASTER (i.e., v1.3.0 and v1.6.1) in the two experiments. Results show that EVOMASTER v1.3.0 achieved on average 28.9% line coverage in *CS1 2021* and *CS2 2021*, and EVOMASTER v1.6.1 achieved on average 18.1% line coverage in *CS1 2023*, *CS2 2023* and *CS3 2023*.

---

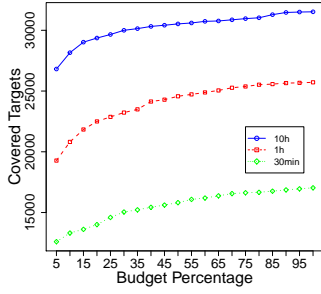
<sup>15</sup>For instance, a line of code such as `if(x == 42)` will be treated as three distinct testing targets for the line itself (denoted as LINE), the branch where the condition is evaluated as true (denoted as BRANCH\_TRUE), and the branch where the condition is evaluated as false (denoted as BRANCH\_FALSE) respectively. Once this line is executed, we consider the LINE target is covered, and the two branches targets are *reached*. The BRANCH\_TRUE target can be considered as *covered* only if `x` is 42, while the BRANCH\_FALSE target can be considered as *covered* only if `x` is not 42. In order to cover these three targets, tests need to be generated to execute the line and to cover true and false outcomes of the branch. More detail about the testing targets can be found in [34, 25].



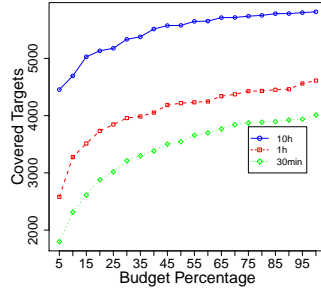
(a) *CS1 2021*



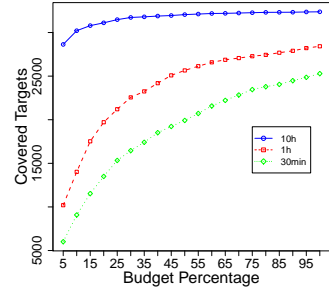
(b) *CS2 2021*



(c) *CS1 2023*



(d) *CS2 2023*



(e) *CS3 2023*

Figure 5: Average covered targets (y-axis) with 30m (green line), 1h (red line) and 10h (blue line) throughout the search for industrial applications, reported at 5% intervals of the used budget allocated for the search (x-axis)



In User Study 2021, to promote further integration of our approach into industrial setting, we performed a manual review on the tests generated with 10 hours and coverage reports with the industrial partner, together. Such review could help us to understand about industrial APIs and provide industrial partner information about potential benefits (if any) achieved by our approach. By reviewing coverage report, classes for implementation industrial services at Meituan can be categorized into three parts, i.e., *Client* for designing how the RPC-based API can be accessed, *DAO* for data persistence and communication with database, and *Server* for implementing business logic. Line coverage achieved by EVOMASTER on each part is (1) *CS1 2021*: 62.8% on *Client*, 10.9% on *DAO*, and 46.3% on *Server*; (2) *CS2 2021*: 79.8% on *Client*, 15.3% on *DAO*, and 57.1% on *Server*. First, as seen from the line coverage on each part, we found that achieved coverage on *DAO* is low. One possible reason for the low coverage in *DAO* is that we disabled *SQL heuristics* in these experiments, as discussed in Section 5.1.2. Thus, without any heuristic on SQL, it might result in such underperformance in *DAO*. In addition, we also found that there exist some unreached classes in *DAO* which define SQL operations (e.g., `SELECT ALL`, `DELETE BY WHERE`), and those classes seem never been used (i.e., dead-code). As checked with the developers at Meituan, those classes are actually automatically generated by their development framework. In addition, they also pointed out that, in their services, there possibly exist some other unused code which is not deleted due to their frequent updated services. We also compared to manually written automated end-to-end (E2E) tests, if the SUT has any, in terms of code coverage. At the time of performing the first experiment in 2021, there were 11 manually written JUnit tests for *CS2 2021* which achieve 16.8% line coverage, and no manually written JUnit tests for *CS1 2021*. Our generated tests (i.e., 24.8% with 30m, 26.9% with 1 hour, and 32.6% with 10 hours in Table 7a) show clear better results than the existing manually written JUnit tests.

In EVOMASTER v1.6.1, we supported a native fuzzing of RPC APIs [25]. As the issues relating database handling are not tackled yet, effectiveness of EVOMASTER in terms of line coverage is still limited as results obtained in User Study 2023. By looking at code coverage in the two user studies (see Table 7), EVOMASTER might achieve better results on APIs which have relatively less complexity. For instance, for *CS1 2021* (32,292 LOC, and 33 endpoints), *CS2 2021* (12,152 LOC, and 13 endpoints) and *CS2 2023* (26,353 LOC, and 29 endpoints), EVOMASTER achieved around 30% line coverage using 10-hours time budget. For *CS3 2023* (193,024 LOC and 63 endpoints), EVOMASTER achieved 22.8% on average of 10 runs using 10-hours time budget. On *CS1 2023* (57,209 LOC and 62 endpoints), the performance of EVOMASTER is limited, i.e., 10.4% on average. Compared to *CS3 2023*, *CS1 2023* has less code and similar amount of exposed endpoints, but, EVOMASTER achieved the worst code coverage. Then, by further investigating *CS2 2023* and discussing it with the industrial partner, we found that this service involves much more complex business logic than *CS3 2023*, which might explain the underperformance of EVOMASTER. Note that as parts of user studies, analyses in terms of effectiveness (such as line coverage) aimed to provide evidence of potential benefits EVOMASTER can bring to industry then promote further integration of our approach into industrial setting. Analyzing the limitations of EVOMASTER in fuzzing industrial APIs is beyond the scope of these user studies that requires the effort of a dedicated paper, e.g., [7].

**Faults detected by EVOMASTER.** Results of fault detection in the two user studies at Meituan are represented in Table 7. As a white-box fuzzer, EVOMASTER

Table 7: Results of line coverage and detected faults achieved by EvoMASTER with three time budget settings, i.e., 30 minuses, 1 hour and 10 hours.

(a) Results of one run collected in 2021 using an additional REST API layer. Note that # Detected Faults (REST) was reported by EvoMASTER based on 500 status code or mismatched schema for REST API, and # Detected Faults (Manual) was based on a manual review performed by the first author and one employee at Meituan.

SUT	TB	#Tests Reviewed Manually	Line Coverage %	# Detected Faults (REST)	# Detected Faults (Manual)
<i>CS1 2021</i>	10h	231	33.5	12	34
	1h	-	27.1	6	-
	30m	-	27.8	3	-
<i>CS2 2021</i>	10h	64	32.6	13	18
	1h	-	26.9	8	-
	30m	-	24.8	8	-
Arithmetic Mean		-	28.9	8.3	26

(b) Results of 10 runs collected in 2023. The results are reported with mean, median, minimum and maximum of 10-times repetition for each setting on each SUT as Mean (Median) [Minimum, Maximum].

SUT	TB	Line Coverage %	# Detected Faults
<i>CS1-2023</i>	10h	10.4 (10.3) [9.9, 11.6]	188.0 (188.5) [170, 210]
	1h	8.7 (8.6) [8.0, 9.6]	73.2 (73) [63, 82]
	30m	5.8 (7.8) [0.2, 8.8]	46.8 (64) [1, 76]
<i>CS2-2023</i>	10h	30.6 (30.8) [27.9, 33.4]	23.4 (23.5) [22, 25]
	1h	24.5 (24.7) [22.5, 26.8]	21.3 (22) [20, 23]
	30m	21.6 (21.4) [17.7, 26.2]	18.9 (20) [12, 22]
<i>CS3-2023</i>	10h	22.8 (22.8) [22.4, 23.4]	72.7 (72.5) [71, 78]
	1h	20.2 (20.1) [19.7, 21.1]	71.6 (71) [69, 80]
	30m	18.0 (18.1) [17.0, 19.0]	67.7 (68) [66, 69]
Arithmetic Mean		18.1	64.8

```

1 public Response interfaceName(Reuquest resuest){
2     Preconditions.checkArgument(resquest.getUserId()>0,"invalid userId");
3     ...
4     return response
5 }

```

```

exceptionName:com.xx.aa.bb.cc.thrift.common.XXXXTException
exceptionMessage:invalid userId

```

Figure 6: An example of a low-priority fault to fix

detects faults with the corresponding line numbers where they occur, i.e., representing the last lines executed in the business logic of the SUT. As all reported faults at the end of the search are associated with distinct lines, they can be considered unique. For *CS1 2021* and *CS2 2021*, the results reported by EVOMASTER v1.3.0 (# Detected Faults (REST)) are derived based on REST characteristics, i.e., 500 status code and mismatch issues in OpenAPI schemas. To avoid potential issues of identifying faults in RPC APIs using domain knowledge of REST, we performed a further manual fault identification with an employer who is familiar with two APIs at Meituan using the generated tests (i.e., execute the tests and review the tests and their execution results).

By reviewing 231 tests for *CS1 2021* and 64 tests for *CS2 2021*, 34 unique faults for *CS1 2021* and 18 unique faults for *CS2 2021* with the manual identification are reported in the # Detected Faults (Manual) in Table 7a. Note that each test generated by EVOMASTER is composed of not only requests to the SUT but also the corresponding responses. Thus, the employee can identify faults by checking whether the responses match the expected outcomes based on the requests or whether exceptions are properly handled. With the first experiment, we found that, without a native support on RPC-based API fuzzing, it is not effective to recognize faults in tests generated by EVOMASTER. Such problem has been solved in the second experiment with EVOMASTER v1.6.1 as we have enabled such a native support for automated fuzzing of RPC API that comprises a schema parser for RPC API and new heuristics specific to RPC domain (such as covering various responses of RPC calls and maximizing found faults in RPC APIs) [25]. For *CS1 2023*, *CS2 2023*, and *CS3 2023*, with EVOMASTER v1.6.1, on average 64.8 faults (see # Detected Faults in Table 7b) are derived based on RPC characteristics, i.e., exceptions thrown from distinct places in the code or service error derived using customized methods [25]. Ideally, all such problems should be fixed, e.g., exceptions should be properly handled to avoid further propagation in the microservices architecture. Hence, there is no need to conduct the manual fault identification for results reported by EVOMASTER v1.6.1.

**Feedback from industrial partner.** In the process of reviewing the tests, the industrial partner found that, the tests outputted by EVOMASTER are useful for them from three perspectives which are rarely covered with manual testing and manually written automated tests<sup>16</sup>: (1) *Input validation*: the generated tests are capable of

<sup>16</sup>At Meituan, there exist two kinds of system-level tests. One is manual testing, i.e., testers act as their real users from the user side involving many services for processing the user request (see Figure 2) as real business scenarios. The other kind is manually written automated tests with JUnit, which mainly addresses their business scenarios and logic for a service.

```

1 public Response interfaceName(Request request){
2     List<ResponseDo> list = service.queryData(request.getIds());
3     List<Integer> ids = list.stream().map(ResponseDo::getId).collect(
4         Collectors.toList())
5     ResultSet rs = rep.findDetailsByIds(ids);
6     ...
7     return response;
8 }

```

```

exceptionName:java.sql.SQLException
exceptionMessage:Error querying database.
Cause: java.sql.SQLException: java.sql.SQLException: java.lang.
ArrayIndexOutOfBoundsException\n
### The error may exist in class path resource [mybatis/shard/XXXXMapper.
xml]\n
### The error may involve defaultParameterMap\n
### The error occurred while setting parameters\n
### SQL: select xxxxx from XXXX where xxx in ( ? , ? , ? , ? )\n
### Cause: java.sql.SQLException: java.sql.SQLException: java.lang.
ArrayIndexOutOfBoundsException\n
uncategorized SQLException; SQL state [null]; error code [0]; java.sql.
SQLException: java.lang.ArrayIndexOutOfBoundsException; nested
exception is java.sql.SQLException: java.sql.SQLException: java.lang.
ArrayIndexOutOfBoundsException

```

(a) A high-priority fault before fixing

```

1 public Response interfaceName(Request request){
2     Response response = null;
3     List<ResponseDo> list = service.queryData(request.getIds());
4     List<Integer> ids = list.stream().map(ResponseDo::getId).collect(
5         Collectors.toList())
6     // fix the fault by adding an empty check for ids
7     if(ids.isEmpty()){
8         return response;
9     }
10    ResultSet rs = rep.findDetailsByIds(ids);
11    ...
12    return response;
13 }

```

(b) A high-priority fault after fixing

Figure 7: An example of a high-priority fault to fix

covering various input validations of the services, and their combinations; (2) *Assertion generation for complex responses*: EVOMASTER is able to generate successful requests to the SUT, and have complete assertions with respects to the responses that could be not possibly achieved (way too time consuming) by manually written tests when the elements in the response are many (e.g., large and complex returned responses); (3) *Exceptional branch coverage*: the tests also enable covering different critical exceptional scenarios such as invalid operations on resources at a given state, authentication, or time related constraints on operations (e.g., exceptions regarding too frequent operations on the endpoints). It is noteworthy that the identified benefits gave our industrial partner confidence to take effort to use our approach and promote further its integration into their development/testing process.

Regarding fault fixing, in the first experiment, out of the total detected 52 faults (i.e.,  $34 + 18$ ), the industrial partner treated 21 faults (10 for *CS1 2021* and 11 for *CS2 2021*) as relatively critical. These 21 faults have been fixed. In the second experiment, with a setting of 10-hours time budget, a total of 2,841 faults were detected over 10 repetitions (i.e.,  $188.0 \times 10$  for *CS1 2023*,  $23.4 \times 10$  for *CS2 2023*, and  $72.7 \times 10$  for *CS3 2023*). It was impractical to check all of them. However, since EVOMASTER now has been integrated into the pipeline at Meituan, the fault fixing rate is around 10% reported by our industrial partner. The fixing rate might relate to the priority of the faults to be fixed and the effort required to check them. For instance, Figure 6 represents a fault detected by EVOMASTER v1.6.1 which our industrial partner considered low-priority to fix, while Figure 7 represents a high-priority fault to fix. In Figure 6, a `com.xx.aa.bb.cc.thrift.common.XXXXXException` exception might be thrown in line 2 during request validation if the specified `userId` does not exist in the database. As the API is typically invoked by other APIs from the large microservices rather than being directly accessed by users, our industrial partner considers that it is uncommon to have invalid `userId`, then the fault is the low-priority to fix. In Figure 7a, a `java.sql.SQLException` exception might be thrown in line 9 if the provided `ids` is empty, and how the fault is fixed is shown in Figure 7b. Our industrial partner considers that exceptions related to the database are of high priority to fix, as the database may be accessed by many APIs, and `ids` could potentially be empty in the scenario. Moreover, the low fixing rate can also be attributed to the non-trivial effort required to check faults after each run of applying EVOMASTER. Since the faults are unclassified in EVOMASTER, QA engineers or testers must assess whether each fault requires attention before reporting them to developers. A further strategy is needed to better distinguish high-priority faults from all other faults.

**RQ2.1:** *As a search-based approach, i.e., EVOMASTER, a relatively higher budget (e.g., 10-hours) is required to tackle industrial test generation. With the recent EVOMASTER (i.e., v1.6.1) on three industrial APIs at Meituan, it is capable of achieving up to 33.4% (on average 18.1%) code coverage and identifying up to 210 (on average 64.8) potential faults. The fault fixing rate is low (i.e., around 10% reported by Meituan) that might relate to the the ratio of high-priority faults detected and non-trivial effort required to check all faults.*

Table 8: Answers of pre-check questions of assessing effectiveness collected in User Study 2023 and User Study 2024

CS	Position	PR1: Familiarity	PR2: Achieved Code Coverage	PR2: Detected Faults	PR5: Time Cost (minutes)	PR6: Enough Time?
<b>User Study 2023 at Meituan</b>						
<i>CS1 2023</i>	QAM	2	×	✓	30	<b>Y</b>
	QAE	5	×	✓	60	<b>Y</b>
	QAM	4	✓	×	20	<b>Y</b>
	DPE	5	×	✓	5	<b>Y</b>
	QAE	5	✓	✓	60	<b>Y</b>
<i>CS2 2023</i>	DPM	5	✓	✓	120	<b>Y</b>
	PDM	3	×	✓	10	<b>Y</b>
	QAE	5	✓	×	20	<b>Y</b>
	QAE	5	✓	✓	10	<b>Y</b>
	QAM	5	✓	×	10	<b>Y</b>
<i>CS3 2023</i>	DPE	3	✓	×	120	<b>Y</b>
	QAE	5	✓	×	120	<b>Y</b>
	QAE	3	✓	×	60	<b>Y</b>
	QAE	5	✓	×	120	N
	QAE	1	✓	×	120	N
Median		5	-	-	60	-
Sum		-	11	7	-	13/15
<b>User Study 2024 outside of Meituan</b>						
Untitled	DPM	1	✓	×	-	N
Untitled	DPE	2	✓	×	-	N
Untitled	SWA	4	✓	×	30	N
Untitled	Tester/QAE	1	×	✓	-	N
Untitled	Tester/QAE	3	×	✓	-	N
Median		2	-	-	-	-
Sum		-	3	2	-	0/5

### 5.2.2. Results for RQ 2.2

To evaluate effectiveness of EVOMASTER from the point of view of industrial practitioners, we conducted a questionnaire that comprises a set of questions (see Section 4.5.2) for collecting feedback of industrial participants about generated tests. This questionnaire experiment was performed in all of the three user studies.

Table 8 represents responses for *RPs* (see Table 4). *RPs* was designed after the first user study (i.e., User Study 2021) then employed in User Study 2023 and User Study 2024 in order to better understand answers of industrial participants, such as their familiarity with the API they employed in this user study (*PR1*), the feature they preferred most in tests (*PR2*), the property they consider most important in assessing tests in terms of readability (*PR3*) and quality (*PR4*), the time cost they spent to inspect the generated tests (*PR5*), and whether they had enough time to conduct this user study (*PR6*).

*PR1: How familiar are you with the web service used in the study?* In User Study 2023 at Meituan, except one participant in *CS1 2023* and one participant in *CS3 2023*, all other participants have at least *3-Moderate* familiarity. Among the 15 participants, median familiarity is *5-Very familiar*. In User Study 2024 outside of Meituan, among 5 participants from five different companies, two rated *Moderate Familiar* and *Familiar* while the other three rated either *Very Unfamiliar* or *Unfamiliar*.

*PR2: What is the most important feature that you would like to have in the generated test cases?* In User Study 2023, among the 15 participants, 7 out of them voted *detected faults*, 11 out of them voted *achieved code coverage*, none of them voted *readability* and none of them specified other features they would most like to have. Note that this question was marked to select only one choice. However, there are three participants who selected two most important features (i.e., *achieved code coverage* and *detected faults*). Based on the results, code coverage and fault detection seem the most two important features that participants would like to have in the generated tests. Based on the answers of these 15 participants, we received more votes for code coverage (i.e., 78.6%) compared to fault detection (i.e., 46.7%). Responses obtained in User Study 2024 also have more votes for code coverage than fault detection, i.e., 3 vs. 2.

*PR3: What properties are important for you to rate readability? (such as size of test cases, name of tests, grouped tests into different files, etc.)* In User Study 2023, we received 7 answers relating to group tests, 5 answers relating to name of tests and 2 answers relating to provide comments. All answers suggest to group the tests, name the tests or add comments that could refer to which RPC interfaces (see Section 4.3.1) involve. If the test involves multiple interfaces, it would be better to represent what scenario the test examines. In addition, two answers mention that tests should have clear requests, responses and test assertions. Moreover, one suggests that the assertions would better to represent what it tests for, thus once it fails, it could help them to fix bugs. Furthermore, there is one recommendation to better separate test data and test cases into different files, which might help them to analyze and maintain the tests. Regarding the size of the tests, we received only one answer which concerns about this issue. In User Study 2024, we received only one response, and the participant (i.e., DPM) considers grouping tests to be important for rating readability.

*PR4: What properties are important for you to rate quality? (such as code coverage, automatically test the web services regarding input validation, scenarios (e.g., invalid inputs, business logic) to be tested automatically, etc.)* In User Study 2023, 13 participants out of 15 describe one of the most important properties to rate the quality of tests is related to real business scenario, e.g., whether the tests are able to test real business logic, whether the tests are able to test the core business logic, and what scenarios the tests are able to test (e.g., invalid input and happy path). 7 participants consider that high code coverage is important for them to rate the quality, and one participant considers the quality of assertions. There is also one participant (who is a developer) considering the quality relating to the capability of tests to help developers to detect and fix bugs. In User Study 2024, one (i.e., DPM) out of five participants answered this question and indicated that business logic coverage is important for rating the quality of tests.

*PR5: Before answering the following questions, how much time did it take you to read and understand the tests generated by EVOMASTER?* Results obtained by employees at Meituan are reported in Table 8, i.e., (0, 10m): 1, [10m, 30m): 6, 1h: 3, and 2h: 5. Median time spent to read and understand the tests is 1 hour. However, as only SWA generated tests successfully with EVOMASTER in User Study 2024 (discussed in Section 5.1.1), we only received one response, and the participant took 30 minutes to read and understand the tests generated by EVOMASTER.

*PR6: Did you have as much time as you needed to inspect all the test cases in order to answer these questions?* In User Study 2023, among the 15 participants, two of them claimed that they did not have enough time to read and understand the tests. This

Table 9: Answers of QB1,2,5,7 and 8 in detail for each participant

CS	Position	QB1: Readability	QB2: Quality	QB5: Keep Tests?	QB7: Time Preference	QB8: Time Selection
<i>CS1 2021</i>	QAM	Low	Moderate	✓	-	-
<i>CS1 2021</i>	QAE	Low	Moderate	✓	-	-
<i>CS1 2021</i>	QAE	Low	Moderate	✓	-	-
<i>CS2 2021</i>	QAM	High	Moderate	✓	-	-
<i>CS2 2021</i>	QAE	High	Very High	✓	-	-
<i>CS1 2023</i>	QAM	High	Low	✓	2h	10h
<i>CS1 2023</i>	QAE	High	Low	✓	1h	10h
<i>CS1 2023</i>	QAM	Very High	Moderate	✓	2h	10h
<i>CS1 2023</i>	DPE	Very High	Very Low	×	default	10h
<i>CS1 2023</i>	QAE	Moderate	Moderate	✓	24h	10h
<i>CS2 2023</i>	DPM	High	Low	✓	1h	10h
<i>CS2 2023</i>	PDM	Low	Moderate	×	default	30m
<i>CS2 2023</i>	QAE	High	Moderate	✓	1h	10h
<i>CS2 2023</i>	QAE	High	Moderate	✓	1h	10h
<i>CS2 2023</i>	QAM	Very High	Very High	✓	5h	10h
<i>CS3 2023</i>	DPE	Moderate	Low	✓	5h	10h
<i>CS3 2023</i>	QAE	Moderate	Moderate	✓	5h	10h
<i>CS3 2023</i>	QAE	Moderate	Moderate	✓	5h	1h

could happen in the company as they have tight scheduled tasks for their daily jobs, and possibly they did not get enough time for this research task from their direct manager, or suddenly had other pressing work with higher priority. Based on such responses, we excluded answers of these two participants on *CS3 2023*. Thus, for the experiment in 2023, the analysis of effectiveness was performed based on answers from 13 employees. Regarding the responses in User Study 2024, all five participants claimed that they did not have enough time to inspect the generated tests. Given these results, it is not feasible for us to analyze assessments of the five participants outside of Meituan on tests generated by EVOMASTER (i.e., QBs). Conducting such as a time-intensive user study on effectiveness of a tool (such as, run the tool for fuzzing enterprise APIs with various settings of time budget, and take time to inspect results) is challenging, especially in companies with which the researchers do not collaborate.

Regarding responses of QBs, answers of QBs1,2,5,7 and 8 are reported in Table 9 in detail for each participant. Diverging stacked barplots for analyzing rates on readability and quality are shown in Figure 8.

*QB1: How would you like to rate the readability of the generated tests?* Regarding the readability shown in Figure 8a, we received 3 rates (60%) on *Low* and 2 rates (40%) on *High* in 2021. All *Low* rates are about *CS1 2021*, and all *High* rates are about *CS2 2021* (see Table 9). Compared to the results collected in 2021, we achieve better rates on readability reported in Figure 4b. We received 1 rate (8%) on *Low*, 4 rates (31%) on *Moderate*, 5 rates (38%) on *High*, and 3 rates (23%) on *Very High*. Such improvements might be due to the native support of fuzzing RPC APIs. In addition,



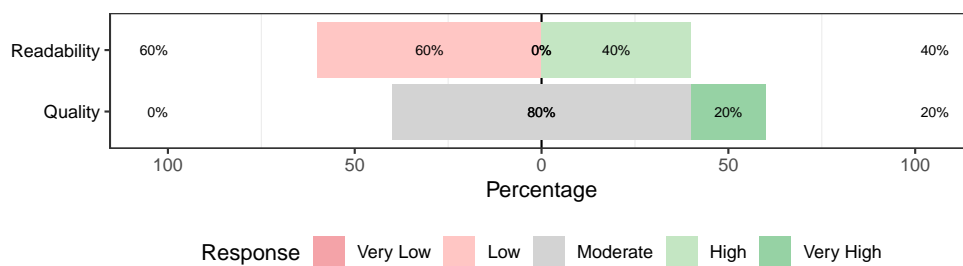
suggested by our industrial partner, EVOMASTER v1.6.1 has been integrated with a simple test grouping strategy to put tests which represent faults (i.e., throwing exceptions) into a different file. The strategy might help to have better rates on the readability. Regarding rates on different SUTs (see Table 9), the *Low* rate is about *CS2 2023* specified by a product manager. Note that the product manager provided a comment as “it is unclear what functions the tests examine” for answering *QB3: How can the generated tests be improved?*, and it might be the reason for the *Low* rate on readability. For *CS2 2023*, the other rates are 3 *High* (by a development manager and two QA engineers) and 1 *Very High* (by a QA engineer), then the rate results might also depend on their daily tasks, e.g., the product manager typically does not need to handle services at the source-code level, thus, they might have high requirements on the readability of the tests. In addition, among the three APIs, we have the best rates of readability in *CS1 2023*, i.e., 1 *Moderate*, 2 *High* and 2 *Very High*. For *CS3 2023*, all of the three rates are *Moderate*.

*QB2: How would you like to rate the quality of the generated tests?* For *CS1 2021* and *CS2 2021*, in terms of overall quality, we received 4 rates (80%) on *Moderate* and 1 rates (20%) on *VeryHigh*. In 2023, we received more negative rates (38%) on the quality (i.e., 1 *Very Low* and 4 *Low*), and less positive rates (8% with 1 *Very High*). This might be related to less code coverage we achieved on *CS1 2023*, *CS2 2023* and *CS3 2023*, for instance, 3 negative rates specified on *CS1 2023* (see Table 9). In addition, we found that all three participants who are in the development department (i.e., one development manager and two developers) gave negative rates. Currently, EVOMASTER derives assertions based on responses. The responses in the industrial APIs could be huge (hundreds or thousands of lines), then having too many assertions on each data entry might not help developers to easily identify the sources of the faults. This might be a reason why we do receive negative rates from the developers. How to improve EVOMASTER for better supporting developers will be a challenge we need to address.

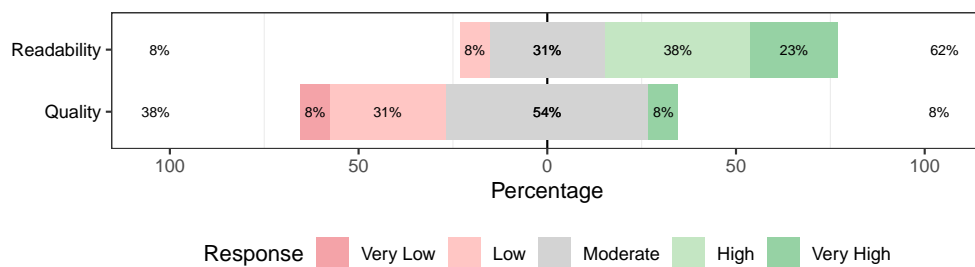
We discuss *QB3: How can the generated tests be improved?* and *QB4: Describe what you like better about manually written tests than generated tests?* together as the two are correlated. With the two questionnaires in 2021 and 2023, we received feedback on needs of further improvements on readability. For instance, it would be better to have some endpoint or interface information on the names, which could help them to read the tests. Grouping the tests according to interfaces and scenarios is highly requested, and it would be better to have further detailed clustering, such as under each interface, further clustering tests for input validation, business operation and exception thrown. In addition, there exist many lines to instantiate input parameters of requests and assert responses. To improve readability, it might be better to put such heavy information into a separated file, e.g., JSON. Moreover, regarding assertions, one participant suggests that the assertions should be created specific to a scenario to test, and there is no need to assert every property of the responses. As an automated approach, it is not trivial to decide which property should be shown, and this will require a further investigation in the future.

Other most common suggestions are related to how effectively generated tests cover their business logic. Based on the feedback, most tests generated by EVOMASTER properly cover the input validation, while only a few tests cover their real business logic and important usage scenarios.

One of the advantages of manually written automated tests is easy-to-read, e.g., test suites are often well grouped and do not contain many tests. Another advantage is specific,



(a) Diverging stacked barplot for responses of rates on readability and quality of generated tests using 5-points likert-scale collected in 2021. Note that positive responses (i.e., “high”) extending to the right and negative responses (i.e., “low”) to the left from the diverging point.



(b) Diverging stacked barplot for responses of rates on readability and quality of generated tests using 5-points likert-scale collected in 2023. Note that positive responses (i.e., “high”) extending to the right and negative responses (i.e., “low”) to the left from the diverging point.

Figure 8: Answers provided by industrial partners about readability and quality of tests generated by EvoMASTER (QB1-2)

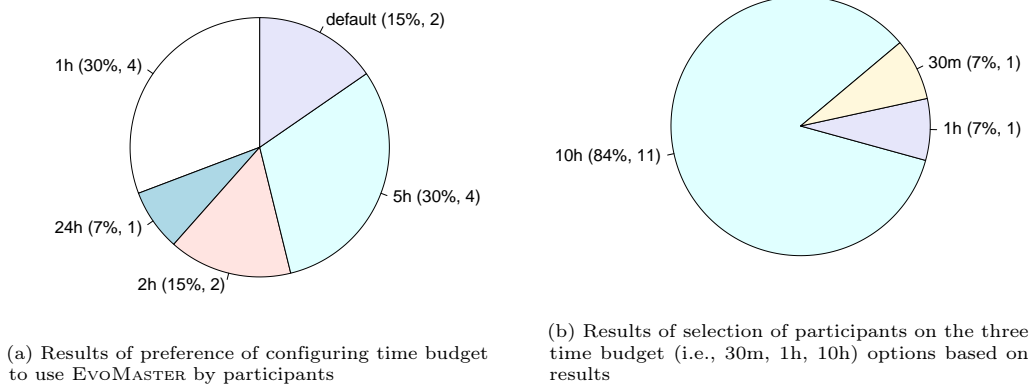


Figure 9: Results of feedback on time budget from industrial participants

i.e., the manually written automated tests are defined to address their business logic, and that has more meaningful combinations of endpoints than automatically generated tests. In addition, the tests could have assertions on database that is currently missing on the tests generated by EVOMASTER.

Furthermore, code coverage and efficiency to produce tests would be better to be improved. Besides, the suggestions on the generated tests, we also received one request to provide information about what code is not covered in terms of each interface, so that it would be easier for them to write manual tests for those missing cases.

*QB5: Would you keep the generated system-level tests?* and *QB6: If yes, how would you like to keep and use them?* Based on the results shown in Table 9, with two experiments involving 18 answers, 16 participants showed positive feedback in keeping the automatically generated tests, i.e., 100% of the respondents in 2021, and 84% of the respondents in 2023. The 2 participants who responded “No” are from one developer who gave *Very Low* rate on quality and one product manager who gave *Very Low* rate on readability (see Table 9).

Regarding *QB6*, the tests could be used together with manually written automated tests, and they would like to further use the generated tests for robustness testing. In addition, they also want to apply them for regression testing in their CI systems (i.e., each time when the services are updated). The tests could also be used to validate and monitor their testing environment for ensuring that all services are up and running.

In the experiment with EVOMASTER v1.6.1, we designed two further questions to collect responses related to time budget that practitioners would like to use.

*QB7: If you are using EVOMASTER, for how long (eg, minutes, hours) do you typically run it for generating test cases? (Default --maxTime option is 1 minute)* Based on results in Figure 9a, 1-hour and 5-hours time budget are the most preferred options specified by the participants, i.e., 30%. There are 15% participants who would like to use the *default* setting and 2-hours time budget. One participant responded 24 hours along with a comment as “Unless the tool can achieve high coverage, there would not be problematic to apply the time budget such as more than 24 hours.”

*QB8: Based on results achieved by EVOMASTER, which option of time-cost do you*

prefer based on the trade-off with the achieved code coverage? This question is designed to collect options of time budget by considering its performance (we provided results shown in Table 7 in the questionnaire). With the three time budget settings, as EVOMASTER with 10 hours could achieve better results, 84% participants opted 10 hours, and 7% participants opted 30 minutes and 1 hour.

Based on the responses of *QB7* and *QB8*, from points of view of practitioners, they might concern more on quality of tests generated by EVOMASTER than time cost, as the tests are produced automatically.

**RQ2.2:** *Regarding results obtained at Meituan, with the recent version of EVOMASTER (v1.6.1), most of the responses on readability are positive (i.e., 62%) or neutral (31%), while the rates on quality are 38% negative, 54% neutral and 8% positive. There exist some limitations (i.e., business logic coverage, test readability and assertions on database state) that should be addressed for further improving the quality. Regarding options of time cost that EVOMASTER takes to generate tests, 1-hour and 5-hours are the most preference based on answers of 13 practitioners. However, practitioners concern more on quality of generated tests compared to time cost (i.e., high time cost could be acceptable as long as it helps to improve the quality). In addition, it is challenging to conduct such as a time-intensive user study on effectiveness of a tool at companies with which the researchers do not collaborate. All participants outside of Meituan claimed that they did not have enough time to inspect tests generated by EVOMASTER, thus, it is not feasible to generalize results we found at Meituan to them.*

### 5.3. Results for RQ3: Integration

Table 4 (see *QCs*) shows the questions we defined in our questionnaire and interview for inquiring potential integration and usage of EVOMASTER in industrial settings. We reported feedback of major barriers in adopting EVOMASTER in User Study 2021 and User Study 2024 (Section 5.3.1). Since the first user study conducted in 2021, EVOMASTER has in 2023 been integrated into the testing pipelines at Meituan. In Section 5.3.2, we share the collaborative process of integrating EVOMASTER into industrial pipelines followed by a discussion about how practitioners would like to use EVOMASTER based on responses collected in the three use studies in Section 5.3.3.

#### 5.3.1. Results for RQ 3.1

In the first user study at Meituan, we asked *QC1* (*What are the major barriers from your point of view in adopting the EVOMASTER tool?*) for collecting potential barriers of integration of EVOMASTER. #1-#4 in Table 10 presents four issues and problems answered by eight industrial participants. The major barrier (#1) was a lack of native support for RPC APIs used by Meituan. Without this support, automating driver generation becomes challenging as it requires manually creating an additional REST layer for RPC APIs. Moreover, using REST APIs for testing of RPC APIs caused side effects in fault detection, as the API calls were made via REST rather than RPC. Due to the large volume of data in enterprise databases, it poses challenges in resetting enterprise APIs (#2). In addition, adopting a new technique into the development process may involve a learning curve for testers (#3). To use tests generated by EVOMASTER, employees

at Meituan were concerned that the tests needed to be compatible with the enterprise’s existing testing framework (#4).

In User Study 2024, we received responses from three out of five participants, i.e., 1 DPM, 1 SWA, 1 Tester/QAM. Two of them concerned the effectiveness of EVOMASTER in fuzzing their enterprise APIs. Unfortunately, they did not identify extra benefits when applying EVOMASTER by themselves as discussed in Section 5.2. The remaining response pertained to learning curve as #3 identified at Meituan.

**RQ 3.1:** *We received responses regarding the learning curve as a major barrier to adopting EVOMASTER in both user studies. In User Study 2021 at Meituan, we also received three additional responses concerning the lack of native support for RPC APIs, the proper reset of enterprise APIs with large volumes of data in databases, and compatibility with the enterprise’s testing framework. The lack of native support for fuzzing RPC APIs was identified as the most important barrier. From outside of Meituan, the remaining responses related to the effectiveness of EVOMASTER, as they failed to identify extra benefits achieved by EVOMASTER.*

#### 5.3.2. Results for RQ 3.2

To promote the adoption of an academic prototype (such as EVOMASTER) into the development pipeline of an enterprise, it requires efforts from both the enterprise and the researchers.

From the viewpoint of industry practitioners, they need to conduct an initial assessment of the tool, similarly to what we did in the first user study (User Study 2021), before allocating more resources, such as employees and hardware. In order to showcase potential benefits to our industrial partner, in User Study 2021, we invited the industrial partner to assess our approach together, i.e., perform a manual review on tests generated by EVOMASTER with the best configuration (i.e., 10 hours) and coverage reports achieved by executing the tests. With the review, the industrial partner found extra benefits achieved by the tests outputted by EVOMASTER, as we discussed in Section 5.2.1. Such benefits gave our industrial partner confidence to take effort to use our approach and promote further its integration into their development/testing process.

As the authors of EVOMASTER, we addressed the most important challenge (i.e., #1 in Table 10) by designing the first approach in the literature for enabling white-box fuzzing of modern RPC APIs in 2022 (see received date of Reference [25]). In the process of integrating EVOMASTER, our industrial partner also made great efforts to support the integration of EVOMASTER into their testing pipelines, i.e., addressing the challenges of #2-#4 in Table 10. After integrating EVOMASTER, we found another issue shown as #5 in Table 10. As the enterprise web services often use their in-house techniques, we have extended EVOMASTER in terms of customization for allowing users (such as our industrial partner) to use their own techniques, such as custom authentication setup and schema to identify whether a request has succeeded. Note that such extension is meant to support customization for any users when needed.

To enhance the application of EVOMASTER at Meituan, in 2023, we conducted the second user study (i.e., User Study 2023) after EVOMASTER enabled a native support of fuzzing RPC APIs. Compared to User Study 2021, we obtained more resources in User Study 2023. For instance, experiment for assessing the effectiveness of EVOMASTER was conducted in a pipeline that allowed for 10 repetitions, rather than a single execution on

Table 10: Major updates our industrial partner and we made for enabling integration of EvoMASTER since 2021

#	Problems	Current status	Efforts
1	<i>Since EvoMASTER does not have a native support for Apache Thrift, this requires a manual configuration between Thrift and REST. This would be a major adoption barrier from their point of view.</i>	This have been resolved. We have proposed a novel approach for supporting a white-box fuzzing of RPC APIs [25]	Researchers
2	<i>Considering that there is no a viable solution to reset their databases yet, in order to generate independent tests with EvoMASTER, our industrial partner would need to develop such database reset operation from industrial side (as they use their own customized RDS).</i>	This is partially addressed. Our industrial partner developed a solution to give us access to execute all kinds of SQL commands that are for cleaning databases. But due to massive amount of data in databases and efficiency issues, we currently still cannot enable the reset in fuzzing web services.	Industrial partner
3	<i>From the view of one of the principal software engineers/managers, there might exist a learning curve for their testers to adapt a new technique.</i>	Our industrial partners have enabled EvoMASTER into their pipeline, and there is not any manual efforts needed for running EvoMASTER on their industrial services. Thus, testers do not need to learn how to start and configure EvoMASTER.	Industrial partner
4	<i>From the view of one of the principal software engineers/managers, there is a need to extend EvoMASTER test case writer to adopt their testing framework.</i>	In [25], we defined a model specific to handling RPC interactions. Thus, based on tests specified with the model, our industrial partner is now extending EvoMASTER to produce JUnit tests which use their testing framework. However, as EvoMASTER produces self-contained JUnit tests (using EvoMASTER driver), the tests can still be executed on their pipeline, and they currently use the tests generated by EvoMASTER before the customized test writer is implemented.	Industrial partner
5	<i>As the industrial APIs might employ some techniques developed in-house, EvoMASTER should provide some capability to industrial partners for further customization when needed.</i>	Since 2021, to better adopt to industrial settings, EvoMASTER has been extended to provide interfaces that allow users to customize methods to set up authentication, identify whether the SUT is up and running, implement own test case writer, and define a schema to categorize responses.	Researchers

an employee's machine. More employees participated in the user study: 1 employee *vs.* 10 for usability, 5 employees *vs.* 15 for effectiveness, 8 employees *vs.* 19 for integration and existing challenges.

**RQ 3.2:** *Integrating an academic prototype into an industrial process requires collaborative efforts from both enterprises and researchers. Industrial partners must conduct an initial assessment of the prototype before allocating resources, while researchers need to demonstrate the prototype's benefits to the industrial partners. To promote an integration of EVOMASTER, we conducted User Study 2021 with employees at Meituan, and industrial partner identified extra benefits achieved by EVOMASTER. Considering the benefits, our industrial partner is willing to integrate our tool in their development and testing processes. All integration related barriers we identified in the first user study have been resolved by our industrial partner and us. In 2023, EVOMASTER has been successfully integrated into the CI pipelines at Meituan, positively impacting hundreds of engineers and testers each day. To further enhance the application of EVOMASTER at Meituan, our industrial partner is willing to invest enterprise resources for conducting another user study.*

#### 5.3.3. Results for RQ 3.3

To further investigate how our industrial partner would like to use EVOMASTER, we asked *QC2* (*Given your current infrastructure setup, how would you like to have automated system-level test generation framework integrated?* as shown in Table 4) to all participants, across the three user studies.

In User Study 2021, the preferred option for the industrial partner is to adopt EVOMASTER into their CI pipeline. As EVOMASTER has been integrated into their CI pipeline, in User Study 2023 the results were as follows:

*AQC2-1* (Integrate EVOMASTER for test generation into the CI pipeline using defined criteria) Participants at Meituan would like to use EVOMASTER to generate tests automatically in their testing pipeline

- (a) when they create a pull request (QAM);
- (b) when code updates;
- (c) when extra check is required to perform on services, such as before deployment into production, during smoke testing;
- (d) when there is a lack of manually written automated tests or low coverage is achieved by the tests (such as less than 20%);
- (e) as a scheduled task for testing services (additionally, such generated tests could be further used for validating and monitoring their testing environment, i.e., all services in the environment should be up and running).

*AQC2-2* (Integrate EVOMASTER into DevTools) In addition, two development managers and one developer also suggested to integrate EVOMASTER into their DevTools at Meituan;

*AQC2-3* (Integrate EVOMASTER for test generation into the CI pipeline for specific code) Moreover, three participants (development manager, QA manger and

engineer) emphasized that they would like EVOMASTER to generate tests only for updated code;

*AQC2-4* (Incorporate EVOMASTER into their regular testing workflows) One participant (QA engineer) mentioned that she/he would like to replace manually written automated tests with tests generated by EVOMASTER.

Compared to the responses obtained in 2021, we received more diverse responses in 2023. This might relate to User Study 2023 involving participants from a wider range of positions and the CI integration of EVOMASTER. Since EVOMASTER was integrated into the CI pipeline at Meituan, as of April 8 2025 it has been used so far to test 1,727 distinct services. According to feedback from Meituan, its primary use is in regression testing and in testing newly developed requirements.

In User Study 2024, we received a response from one out of five participants (i.e., SWA) expressing interest in integrating EVOMASTER into their DevTools for generating regression tests, similar to *AQC2-2*.

**RQ 3.3:** *Based on 27 responses across two use studies at Meituan, as EVOMASTER has been integrated into their testing pipelines, they would like to use EVOMASTER in their daily tasks (such as Pull Request), trigger new test generation by EVOMASTER when needed (such as lack of manually written automated tests or low coverage achieved by the tests), or run EVOMASTER as a scheduled task to test systems in the microservices. In addition, industrial participants suggest to integrate EVOMASTER into their DevTools, have tests generated only for updated code, and replace manually written automated tests with tests generated by EVOMASTER. From outside of Meituan, we received a response from one out of five participants (i.e., SWA) who also expressed interest in integrating EVOMASTER into their DevTools as we found at Meituan.*

#### 5.4. Results for RQ4: Existing Problems and Challenges

The research task for RQ4 aims at understanding existing testing difficulties in industry, and discuss potential solutions that tools like EVOMASTER could help to tackle. To achieve this goal, we defined three questions as shown in Table 4 and asked them to all industrial participants in the three user studies.

*QD1: What is one of your current major issues/time consuming activity with manual testing that you would like to have automation for?*

Regarding responses from employees in the two user studies at Meituan, currently, their testing tasks are mainly performed manually, e.g., manually write automated tests, conduct manual testing by performing users' behaviors from client sides with UI, or analyze the SUT by replaying its historical execution. All these manual tasks are time consuming, then they would like to have techniques to support automation for the following tasks.

*AQD1-1* Based on the structure of their services and their business logic, they are willing to promote automation testing by considering at the level of APIs (such as multiple APIs), business logic, and UI (from Principal Software Engineer).

*AQD1-2* Hundreds of the connected services are running on one platform, and the developers in a team typically do not know the details of services which they



are not responsible for. Automated test data preparation is very appreciated as database and external services setup is very time-consuming for them (from all positions except product manager).

*AQD1-3* They would like to have some kind of automated recommendations for possibly related services and database state for the SUT, which could help their testers (especially for new employees) to setup their testing scenarios (from Director, QA managers and engineers).

*AQD1-4* It would also be very helpful if there is an automated solution to provide information about what they misconfigure and what data is missing in the database (from developers, QA managers and engineers).

*AQD1-5* As industrial requirements are often updated frequently (adding new features, or update existing features), they would like to have automated assessment on how updated requirements impact on current implementation (from development manager), have generated tests according to requirements (from QA engineer), have generated tests according to updated code (from development manager, developer, QA manager and engineer), have automated solution to maintain tests due to frequent updates (from QA engineer).

*AQD1-6* Since industrial APIs might have various critical features, they would like to have tests that can identify problems in terms of security, service characteristic and core services (from QA manager).

*AQD1-7* They would like to have automatically generated tests which achieve more than 50% line coverage (from QA manager and engineer).

In User Study 2024, we received responses from three out of five participants outside of Meituan. Two of these participants (from QA manager and Tester/QA engineer) expressed willingness to automate test generation for API testing and business logic, which aligns with *AQD1-1*. One of these participants (from SWA) showed interest in automating the generation of unit tests and regression tests, as manually writing these tests is time-consuming, similar to *AQD1-5*.

*QD2: What kinds of faults are harder to detect in the system?*

Based on experience of participants, the hard to detect faults collected at Meituan in User Study 2021 and 2023 are related to:

*AQD2-1 data in the database:* the services might perform some analysis on the existing data in the database, then results of the analysis could be used by other services. In this case, if there exists some dirty/invalid data in the database, it could crash the other processes.

*AQD2-2 associated services:* in order to test the system regarding a business logic, there could be multiple services under test. An error might occur in one service (e.g., wrong data saved in database) but the fault might manifest only in other services (e.g., when reading such data, and expecting it to be correct). Considering the complexity of the whole system (hundreds of services), locating this kind of faults is hard.

- AQD2-3 dependent frameworks:* to build their platform, they also used software frameworks, and these frameworks might have bugs which could result in errors of their services.
- AQD2-4 non-functional fault, concurrent faults, and faults relating to extreme scenarios:* such faults could be detected during testing, but it is difficult to reproduce them before locating them.
- AQD2-5 faults relating to non-core business:* it might not be easy to detect faults in implementation of non-core businesses as there typically exists rare tests for them.
- AQD2-6 faults relating to complex computation:* they need to handle complex computation based on business needs. But it is impossible for them to test all possibilities as they are way too many. Thus, it would be difficult to find the faults existing in the possibilities which have not been tested.
- AQD2-7 non-deterministic faults:* since some faults occur non-deterministically, it is hard for them to reproduce the faults and fix them. This is a common issue in the testing of complex distributed systems.
- AQD2-8 interaction with external environment:* Meituan’s business has close interactions with humans (who are responsible for diverse tasks, e.g., quality inspector, storekeeper) and physical devices referred as System Actors. The testing is performed typically with an assumption that the System Actors operate properly. However, when it is not true, then some faults would only be detected once the services are run in production.

In User Study 2024, four out of five participants from outside Meituan stated that identifying faults associated with multiple services with respects to business logic and scenarios is challenging, consistent with *AQD2-2*. The remaining participant pointed out that detecting faults related to exception handling in complex production environments can also be difficult, such as verifying whether data is properly rolled back in the event of a database exception, which might relate to *AQD2-1* and *AQD2-8*.

*QD3: What are the most important challenges that you meet in testing?*

In the two user studies, based on the responses to *QC3*, there are six challenges that employees at Meituan face in testing:

- AQD3-1* Based on the business Meituan deals with, starting from a user request, there could go through lots of services that could result in very complex combinations of various inputs and different states of the system. Considering such complexity, it is hard to ensure quality of tests wrote by developers, and most faults are identified by the QA department.
- AQD3-2* In addition, it is hard to define the testing scope regarding such closely interacted services.
- AQD3-3* Some combinations of the services could be reached only under certain states of the system. To generate and explore such states is also challenging, e.g., constructing real scenarios would require a rich troves of data in the database.

Currently, as mentioned by a principal software engineer who is responsible for developing the testing tools of Meituan, their team would need to construct data into their testing environment according to real scenario in production, e.g., based on real data from their services in production. Such data preparation in the database usually takes at least four hours each week.

*AQD3-4* Moreover, their platform has a high demand for time-efficiency and security (e.g., payments) that would bring further challenges in testing. For instance, their database services are accessed by many services, and there could be slow SQL queries in one service that would negatively affect the database services, and so result in failures (e.g., timeouts) in other services.

*AQD3-5* Furthermore, how to better test scenarios which is related to external environment is challenging for them, as discussed also in *QD2*. Due to the rapid increase of business requirements, often there is limited time for performing manual testing. Applied technology components and architecture could also be updated, and such updates might lead to conflicts and mismatches that bring further challenges in testing.

*AQD3-6* Last but not least, there exist complex computations that cannot be tested sufficiently.

Regarding *QC3*, we received four responses from outside of Meituan. Three of the respondents indicated that the most important challenge is ensuring coverage of code, business logic, business scenarios, and requirements with their currently applied techniques (from 1 SWA and 2 Testers/QA engineers), associated with *AQD3-1*. One of the respondents also noted that balancing the testing budget is a challenge for them (from QA manager), however, such a challenge might heavily depend on the company.

**RQ4:** *Most testing tasks are performed manually for testing the business logic, which shows an urgent need for effective automation support. Due to complexity of microservices and business features, there exist various challenges on, e.g., identifying various faults, locating faults, defining test scope, handling external services and databases, preparing test data, treating frequent update of implementations with technique and architecture update, testing complex computation, testing under uncertain environment (such as humans), and balancing the testing budget.*

## 6. Lessons Learned and Challenges

### 6.1. Testing Setup

In the two user studies conducted with our industrial partner, Meituan, the first challenge we faced was to set up our tool in the industrial environment (as discussed in Section 5.1). Unlike open-source SUTs (like the ones collected in EMB [8, 9]), in industry when dealing with microservice architectures, relationships among services are more complex, accesses to databases are more restricted, and data in the database is more sensitive. Thus, it is hard to handle all dependent resources and enable a proper reset of automated test generation in such industrial setting.

For external services, there could be a possibility to handle them with mocking techniques, and further manipulate responses with search from mocked services to maximize the coverage of testing targets (e.g., code coverage and fault detection). However, considering the complexity of these dependencies, it would be challenging to setup them properly in an automated way. Furthermore, this would further enlarge the search space for test case generation. Note that these issues are not specific to just EVOMASTER, but they would likely apply to any fuzzer used in this testing domain (i.e., microservices).

Regarding the reset of databases, as discussed with the industrial partner, they could help us to isolate complete databases for testing with EVOMASTER. As the first setup, they could provide empty databases with an interface to clean all data. Thus, before executing every individual, we could reset the SUT to a certain state for generating independent tests at the end. However, based on the answers collected during the survey and interviews, it might not be sufficient to generate tests starting with empty databases, and, such tests cannot be executed in their main testing environment (i.e., cannot perform a reset of SUT in the main testing environment). Considering the existing data in the database, they are intentionally added for covering their business scenarios in real practice. Table 2 shows the number of rows of existing data in the databases the SUTs directly interact with in their testing environment. There exist 256,024 rows for *CS1 2021*, and 1,742,574 rows for *CS3 2023*. If the testing is involving many services, it is impractical to clean all data and re-add them before every single test. To better adopt EVOMASTER into this kind of industrial setting, there would be a challenge about how to utilize such existing data in the fuzzer (e.g., during test sampling) that could further result in a good coverage on real scenarios. This would be related to the data preparation challenge discussed in Section 5.4, and have a proper reset of SUTs.

### 6.2. Testing Criteria

By investigating code coverage reports and analyzing responses from industrial practitioners, we found that it might not be sufficient to define testing targets only with code coverage, and fault detection, e.g., more code coverage might not result in covering more business scenarios. For solving industrial testing problems, besides code coverage and fault detection, there would be a need to enhance the fitness function from more dimensions, e.g., business logic, time constraints, database performance and system security, and how to measure such dimensions and cope with them together is very challenging.

When analyzing the industrial services for the experiments in User Study 2021 at Meituan, we found that, in order to understand and monitor their system behaviors, the services are built with various monitoring features. For instance, regarding the business logic, there exist a tracing infrastructure (named *Tracer*) inspired by Google Dapper [89] that enables tracking a complete path taken through every services from a user request, as the example in Figure 2. With the code instrumentation done in EVOMASTER, such tracked paths could provide additional information for measuring coverage related to the business logic. As a timely concern on database performance from our industry partners, it would be important to further enhance the fitness function with respects to SQL execution time for exploring slow SQLs in the SUT.

### 6.3. Fault Localization

In microservices architecture, a request from the client could go through a long and complex list of services. The service where a fault is observed might not be the root of the

fault, which could be in any interacted service with respect to this request. Thus, locating the root of the fault is often challenging. In the context of white-box testing, EVOMASTER manages to refer a last executed statement for an identified bug, i.e., exception thrown during execution and a server error identified based on responses customized by users if they exist [25], when testing a single web application. However, to better locate faults in microservice architectures, more information would be required. To help locating faults, it is required to have a more comprehensive study on faults in microservices with various aspects (e.g., security) and define more intelligent rules to identify the faults. Moreover, non-deterministic faults often appear in microservice architectures, due to the nature of distributed systems. Characterizing the causes of such non-deterministic faults (such as [65]) is required to be investigated in the context of microservice architectures.

#### 6.4. Assertion and Readability

Our current strategy to produce assertions in the generated tests is based on responses. However, as testing of an industrial application, test assertions would be required to be more comprehensive. As pointed out by our industry partner, we plan to firstly extend our assertions with respect to databases. In addition, with this study, we found that some content in responses are non-deterministic. A proper handling is required for producing assertion to deal with such non-deterministic content, to avoid generating flaky tests.

One major concern from testers/developers about automatically generated tests is readability. A more readable output would help them to identify problems and locate faults. EVOMASTER integrates a test clustering technique for splitting tests into different files with respect to faults in the context of REST APIs [90]. First, a further handling for RPC would be needed. We now support splitting tests, which identified thrown exceptions, into a separated file. Considering hundreds of tests (for example, for *CS1 2023*), more clustering strategies would be required, e.g., splitting test suites grouped by interfaces, business scenarios. Moreover, we could also simplify the generated tests by moving over-informative inputs and assertions into a separate file, as suggested by some of the testers at Meituan. But whether such move would have side-effects on debugging will need to be investigated. It would also be helpful to name tests with meaningful info, such as RPC interface name, or the scenario that the test examines. Responses in industrial APIs could be very complex and huge, and not every properties are important to them. Assertions need to be optimized, but how to determine the optimization (such as based on “importance”, but how to calculate “importance”?) is required to study. Furthermore, test readability can be one of the objectives to optimize for [66].

#### 6.5. Generalization of Results

It is challenging to carry out such a time-consuming user study that requires to use the tool, run the experiment, and answer the questionnaire in companies where there is no formal collaboration agreement [29]. This is a common problem in software engineering research: “*running user studies can be difficult, and researchers may lack solution strategies to overcome the barriers, so they may avoid user studies*” [28].

One of the authors contacted few acquaintances in their professional network in China. Five practitioners from different companies responded positively to participate in the third user study in 2024. However, with this small sample size, it may not be sufficient to generalize our results obtained at Meituan to other companies. Nonetheless, we discuss

our findings by comparing responses from user studies conducted at Meituan and those conducted outside of Meituan in this section.

Regarding responses related to *usability*, results at Meituan showed that the employees did not meet much difficulties in using EVOMASTER. Although the writing drivers has the most *Difficult* rates, the employees still managed to complete it. However, from outside of Meituan, only one participant out of five succeeded in writing a driver to enable white-box testing with EVOMASTER, and all participants highlighted the difficulty in configuring such a driver and stated the need for additional documentation and tutorials.

Regarding the most important feature to assess the *effectiveness* of tests, code coverage received more votes than fault detection in the two user studies conducted at Meituan. In User Study 2024, which involved five practitioners from five different companies, code coverage also received slightly more votes than fault detection, similar to the observations at Meituan. Regarding the important properties to rate *readability* and *quality* of tests, we observed the same winner in both user studies conducted at Meituan and outside of Meituan, i.e., grouping tests for readability, and coverage of business logic for quality. Regarding responses to assessment of *effectiveness*, as all five participants outside of Meituan stated that they did not have enough time to inspect the generated tests, it was not feasible for us to include their results and perform further analyses on them.

In terms of *integration*, the learning curve seems to be one of major barriers concerned most by practitioners, as observed in both user studies conducted at Meituan and outside of Meituan. In addition, we found that participants outside of Meituan concerned most about the effectiveness of EVOMASTER before making efforts to integrate it, while we did not observe such a concern in the user studies conducted at Meituan. This might be because participants outside of Meituan failed to identify the benefits due to the complexity of conducting the experiment and the lack of approval to participate in the user study during their working hours. Benefiting from research-industry collaboration, all issues identified in adopting EVOMASTER have been resolved or partially resolved by our industrial partner and us. Considering the lack of willingness to integrate EVOMASTER from participants outside of Meituan, we received only one answer from a participant who expressed interest to integrate EVOMASTER into their DevTools for automated regression test generation. This option was also identified in User Study 2023 conducted at Meituan.

Regarding answers relating to *existing challenges* practitioners face, most of the challenges observed in User Study 2024 (i.e., outside of Meituan) have been reported in the user studies conducted at Meituan, except for one, i.e., balancing the testing budget. However, such a challenge remains a fundamental challenge in software development and might also heavily depend on the company.

### 6.6. Unit Testing Tool

In this study, questionnaire we designed as shown in Table 4 covers all questions from an industrial study on unit test generation [24]. This was done to enable meta-analyses. Note that, as none of participants in User Study 2024 fully completed all questions, we excluded their answers in these meta-analyses.

Based on results obtained in the two user studies from our industrial partner, compared with the industrial feedback on the unit testing tools, both test automation tools (such as EvoSuite and EVOMASTER) obtain positive feedback on usability (*QAs*), i.e., easy task to use it. Regarding the generated tests (*QB1-QB5*), the unit testing tool received more

negative answers on inputs and assertions than EVOMASTER. This could happen because, in the context of system testing, combination of inputs for validation and responses dealt with are more complex than the unit testing. Having more sufficient testing on input validation is also important for web services, and manually creating such combinations would be impractical. Regarding assertions, in the first experiment, EVOMASTER utilizes the HTTP responses to create assertions, and assertions are created based on instantiated Java objects in tests for RPC in 2023. The responses in web services are typically defined with certain purposes, e.g., information messages, while such meaningful responses are often not applied in unit testing. In addition, both tools were evaluated as needing to have an improvement on readability of the generated tests, which is also related to the size of the generated test suite files.

Regarding *QB5: Would you keep the generated system-level tests?*, most of the answers for unit testing is No, but we received 100% Yes answers from 5 participants in the first experiment and 84% Yes answers from 13 participants in the second experiment. This result could be related to negative feedback on the generated tests by the unit testing tool. In addition, compared with system testing, for unit testing it would be easier and less time-consuming to create unit tests manually, and that could result in the preference of developers/testers on manually written tests. Regarding the integration (*QC*), both surveys collected the same preferred option with CI, and the major barriers are both related to currently applied frameworks. With EVOMASTER, these major barriers have been resolved by our industrial partner and us.

Due to the different complexity of the addressed problem compared to manual testing, the system testing automation tool (such as EVOMASTER) received a more positive feedback and shows a more promising integration than unit testing generation. However, this is based on answers from employees at one company with which we have collaborated in industry, and more would be needed to be able to generalize this claim.

### 6.7. Summary of Common Challenges

In this section, we summarize important common challenges (denoted as C-#) which Web API fuzzers might face in industrial settings, along with potential solutions and our future work.

C-1 To fuzz industrial Web APIs, besides generating inputs and requests to the SUT, the fuzzer requires to be capable of manipulating responses of the directly interacted services and databases.

**Future direction:** Mocking techniques are a suitable solution to handle external services. However, how to enable it as part of the fuzzer is a challenge that the research community needs to address.

C-2 To enable an automated Web API testing, it requires to reset states of the SUT. However, enabling such reset is very challenging in industrial settings. For instance, databases in industry are very complex and large, e.g., more than 1 million rows for the test data in *CS3 2023*. It is impractical to reset such database at every test execution.

**Future direction:** As discussed with our industrial partner, they plan to implement a solution to flashback states of the databases based on

timestamps, as they think such solution is also useful for them for their other testing tasks. However, the time efficiency of the flashback is of paramount importance, e.g., 1 second could result in at most 3600 requests within 1 hour, which would significantly limit the effectiveness of the fuzzer.

- C-3 In industry, testing data in the databases is typically maintained based on data collected in real production for testing their APIs with various business scenarios. Besides resetting databases, considering the amount of data in these databases, how to make full use of them is another challenge.

**Future direction:** EVOMASTER is equipped with *sql handling* to analyze existing data of SQL databases with JDBC connection. As a white-box SBST fuzzer, it is possible to enable heuristics for optimizing selection and manipulation of existing data which requests relate to. However, considering the large amount of data, how to optimize it in a time effective way would be a potential challenge we might face.

- C-4 An industrial Web API could connect multiple databases using database sharding techniques, which is a common solution for distributed databases in industry. Thus, fuzzers are required to be able to handle multiple connections in their database handling solutions.

**Future direction:** EVOMASTER enables SQL database handling, but such handling is currently built based on one JDBC connection. EVOMASTER could be extended to support multiple JDBC connections, but how to cope with the multiple connections when inserting data (e.g., determine a database to perform the insertion) will be another challenge we need to address.

- C-5 Common testing criteria might not be sufficient to tackle industrial problems. As discussed with our industrial partner, they concern more on metrics relating to their business logic, efficiency, performance and security. Such dimensions are likely also common to other industrial APIs. How to measure these dimensions as testing criteria and deal with them during fuzzing APIs would be an important challenge that research community should address.

**Future direction:** Currently EVOMASTER does not consider performance testing criteria, such as response time, which could be considered as future work to extend EVOMASTER for performance testing. However, such information has been collected during search. As requested by our industrial partner, EVOMASTER now can trace and output logs of all SQL commands executed during testing. Then, with this info, our industrial partner can conduct further performance analysis on their databases based on these logs.

**Future direction:** (specific to industrial microservices equipped with monitoring system) As a white-box fuzzer, it is possible to track paths



of accessing APIs at runtime by instrumenting services for monitoring microservices, if the SUT has any such connections. Then, the tracked information might enable identifying business logic and further employ them as parts of fitness function of EVOMASTER.

C-6 Due to close and complex interactions among Web APIs and databases, locating faults in microservices is a challenge industry currently meets.

**Future direction:** To better locate faults, it is the first step for researchers to understand and characterize faults in microservices, which is going to be an important future work.

C-7 Industrial Web APIs often employ sources of non-determinism in their implementation (such as calling `Random` or accessing current CPU clock) that can result in flaky tests. To avoid producing flaky tests, it is important that fuzzers can properly handle such sources of non-determinism in the generated tests.

**Future direction:** As a white-box fuzzer, EVOMASTER could take advantage of directly handling non-determinism in the source code (e.g., instrumenting a fixed seed for `Random`). However, it requires to first identify all possible non-deterministic sources in the source code.

C-8 Hundreds of tests could be generated when fuzzing industrial APIs, such as 231 tests for *CS1 2021* and 488 tests for *CS1 2023*. Note: a fuzzer could evaluate millions of tests during its search, but the output is usually minimized, e.g., generate minimal test suites with the highest coverage achieved during the search. To better help testers use these generated tests (e.g., for debugging and regression testing), readability of the tests is important in industrial practice.

**Future direction:** Classification is one potential solution to improve readability of the tests, such as group tests based on success or failure responses into different files. The tests with failure in responses could be further classified into different files based on types (such as user error or system error). To enable such classification, a study for understanding faults in microservices is required to conduct first. EVOMASTER has been equipped with a clustering strategy for tests of REST APIs. For RPC, we developed a strategy to put tests which lead to exception thrown into a separated file. However, strategies specific to the domain of RPC APIs and business scenarios are needed.

## 7. Threats to Validity

*Conclusion Validity.* In this study, we carried out three user studies to assess *usability* and *effectiveness* of EVOMASTER, investigate *potential integration* of EVOMASTER into industrial process, and understand *existing challenges* from the viewpoints of practitioners.

These user studies were conducted with in total 32 industrial professionals with different roles, e.g., a director, QA managers, testers, developers, developer managers and product manager. To collect opinions and perceptions of practitioners, we conducted questionnaire and interviews that comprise 29 questions (Table 4), and summarized our findings based on in total 606 answers of the questionnaire/interviews across the three user studies as shown in Table 5. As the authors of EVOMASTER, we also performed three tasks in this industrial evaluation, i.e., a preliminary study of EVOMASTER on industrial SUTs (such as database and authentication handling), coverage analysis based on generated tests, and share our collaborative process of integrating EVOMASTER into the industrial CI pipeline. To reduce bias in the analysis of the results, we confirmed our findings with our industrial partner, ran the experiment of EVOMASTER on multiple enterprise APIs, and together with our industrial partner performed a manual analysis in the generated tests for assessing the effectiveness of EVOMASTER.

In 2021, as EVOMASTER for experiments could only be run on the machine of an employee at Meituan, we conducted the first experiment with one single run. However, as EVOMASTER has been integrated into the testing pipelines at Meituan in 2023, we conducted the second experiment with 10-time repetitions, to reduce issues with results obtained by chance due to the randomness nature of the search algorithm used in EVOMASTER.

*Internal Validity.* There is no guarantee that the implementation of EVOMASTER is bug-free. However, EVOMASTER has been used to perform many experiments using open-source SUTs [2, 41, 4, 5, 7, 36, 25], and it is also carefully tested, i.e., currently with 735 test suites (including unit tests and end-to-end tests), which covers 60.23% lines of its code base. In addition, EVOMASTER is an open-source project which can be reviewed and examined by anyone who is interested in it. Regarding the questionnaire survey, we have involved 32 industrial participants from eight profiles in industry (as discussed in Section 4) for reducing possible bias in these answers.

*External Validity.* In this work, we trade *generability* (i.e., focus on one single company) and *sample size* (e.g., limited number of engineers and testers, in total only 27) to obtain more *realism*: i.e., we want to *study the actual use of fuzzers in industry by practitioners on their own large, industrial systems*. There might be external threats to the generalizability of our findings, i.e., the results may not directly transfer to other companies. However, this limitation is common in software engineering research, especially when user studies are conducted without formal collaboration with multiple companies [29, 28]. Alternatives would had been to use students in the empirical study, to have a larger sample size. Not only there would be questions on how results could generalize to actual industrial practice, but there would be major barriers to setup such an experiment. For example, how long time (possibly months) it would take for the students to get familiar with large industrial systems, part of a microservice with tens of millions of lines of code. For the same reason, even if Meituan has hundreds of engineers and testers, given a specific API used as SUT, only a small number of these professionals are familiar with the details of such API. Having an engineer from another department being involved in an experiment on an API that they never used / worked on before would not be realistic. This is particular the case for when evaluating the *readability* of the generated tests, as readability of the generated tests is strongly related as well on how familiar the tester is with the tested API. To further clarify our findings in these user studies, we discussed generalization of the results for each RQ in Section 6.5.

In addition, this kind of analyses involving human subjects are time consuming, much more than for example evaluating a fuzzer only “in the lab” on a set of open-source APIs. Also, there is a major difference between empirical studies with students and empirical studies with practitioners in industry, as their cost is much higher, and so usually their sample size is smaller. However, this kind of studies are essential to address the gap between research and practice. Many studies done only “in the lab” by researchers might rely on assumptions that do not hold in practice, making the use of novel scientific techniques not fit for practice. A scientific result with no practical use might not be fitting for an engineering discipline like software engineering. Reporting experience studies in different companies can help building a large enough body of knowledge throughout time, from which meta-analyses can be used to infer general results. To this aim, we used the same type of questions from an existing work on the application of a unit test generation tool in industry (Section 6.6).

Furthermore, the study was performed with Web APIs using the same RPC framework. In the context of RPC, different frameworks could share parts of common domain knowledge (e.g., database interactions and dependencies among services), and the findings in this study might be applicable in other contexts (such as Apache Dubbo). However, five SUTs alone are not enough yet to draw general conclusions. More user studies of this kind are in dire need in the literature.

## 8. Conclusion

It is very challenging to test enterprise applications using a microservice architecture, as they are often composed of a large amount of web services, such as the ones developed at Meituan. Automated techniques such evolutionary search have demonstrated their effectiveness on solving many testing problems. However, there is a lack of empirical evaluation in industrial microservices.

EVOMASTER is an open-source test case generation tool that exploits the latest advances in the field of Search-Based Software Testing for web services. In this paper, we conducted an empirical study on integrating EVOMASTER into real industrial settings. The study was performed three times, in 2021, 2023 and 2024, using three versions of EVOMASTER (i.e., v1.3.0, v1.6.1 and v2.0.0) and involving in total 32 participants. The first two user studies were carried out with 27 employees at Meituan, which is our industrial partner. The third user study was carried out with 5 practitioners from five different companies that we have no formal collaboration agreements with. The user studies were designed from viewpoints of practitioners that comprise questionnaire and interviews for assessing *usability* and *effectiveness*, *integration* of EVOMASTER, and understanding *existing challenges* that practitioners are facing. Benefiting from research-industry collaboration, with an access to information of enterprise APIs and direct communication with their employees, we (as the authors of the academic prototype) discussed our findings and shared our experience in the process of integrating our tool into the industrial pipeline at Meituan.

Results show that EVOMASTER clearly demonstrates its benefits to our industrial partner, Meituan, e.g., unused code identification, test generation, code coverage, and real fault detection. But, there are still many critical challenges posed in this study that are required to be investigated by the research community, like interactions with external services and large distributed databases.

EVOMASTER is in active state of development. Since this pilot study at Meituan was carried out in 2021, we addressed the most major challenges for Meituan by proposing a novel approach in 2022 that facilitates a native support for RPC directly. Once such major challenges pointed out in previous study are addressed, it is important to repeat these surveys/questionnaires with industrial practitioners, to see what next needs to be addressed before this kind of fuzzing techniques can become commonly used in industrial practice as the second user study we carried out at Meituan.

The third user study was performed for generalizing results obtained at Meituan to other enterprises. Results on *usability* and *effectiveness* indicate the challenges in conducting the user study that requires programming proficiency to use the tool and takes time to run the experiment. However, responses to *integration* and *existing challenges* can also be identified as in the user studies carried out at Meituan.

Many studies done only “in the lab” might rely on assumptions that do not hold in practice, making them unusable for industry. This paper gives the important scientific contribution of empirically evaluating a Web API fuzzer in industry, confirming its applicability and usability in this domain. Most of the challenges identified in this study are not specific to EVOMASTER, and would likely apply to any other fuzzer for Web APIs.

Currently, EVOMASTER has been integrated in the CI pipelines of Meituan, where it has been applied to fuzz 1,727 distinct services so far, for tens of millions of lines of code. EVOMASTER is open-source, with each release automatically stored on Zenodo for long term storage (e.g., version v2.0.0 [12]). To learn more about EVOMASTER, visit our website [www.evomaster.org](http://www.evomaster.org).

## Acknowledgments

Man Zhang is supported by State Key Laboratory of Complex & Critical Software Environment (SKLCCSE, grant No. CCSE-2024ZX-01) and the Fundamental Research Funds for the Central Universities. 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] S. Newman, Building microservices, " O'Reilly Media, Inc.", 2021.
- [2] A. Arcuri, RESTful API Automated Test Case Generation with EvoMaster, ACM Transactions on Software Engineering and Methodology (TOSEM) 28 (1) (2019) 3.
- [3] A. Arcuri, J. P. Galeotti, Sql data generation to enhance search-based system testing, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 1390–1398. doi:10.1145/3321707.3321732. URL <https://doi.org/10.1145/3321707.3321732>
- [4] A. Arcuri, J. P. Galeotti, Enhancing search-based testing with testability transformations for existing apis, ACM Transactions on Software Engineering and Methodology (TOSEM) 31 (1) (2021) 1–34.

- [5] M. Zhang, 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)* 31 (1) (2021).
- [6] M. Zhang, B. Marculescu, A. Arcuri, Resource and dependency based test case generation for RESTful Web services, *Empirical Software Engineering* 26 (4) (2021) 1--61.
- [7] M. Zhang, A. Arcuri, Open Problems in Fuzzing RESTful APIs: A Comparison of Tools, *ACM Transactions on Software Engineering and Methodology (TOSEM)* (may 2023). doi:10.1145/3597205.  
URL <https://doi.org/10.1145/3597205>
- [8] Evomaster benchmark (emb), <https://github.com/WebFuzzing/EMB>, online, Accessed August 6, 2024.
- [9] A. Arcuri, M. Zhang, A. Golmohammadi, A. Belhadi, J. P. Galeotti, B. Marculescu, 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.
- [10] A. Arcuri, EvoMaster: Evolutionary Multi-context Automated System Test Generation, in: *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2018.
- [11] A. Arcuri, J. P. Galeotti, B. Marculescu, M. Zhang, EvoMaster: A Search-Based System Test Generation Tool, *Journal of Open Source Software* 6 (57) (2021) 2153.
- [12] A. Arcuri, M. Zhang, A. Belhadi, J. P. Galeotti, Bogdan, Seran, A. Golmohammadi, A. M. López, hghianni, O. D, A. Aldasoro, A. Panichella, K. Niemeyer, Emresearch/evomaster: v2.0.0 (Oct. 2023). doi:10.5281/zenodo.8431056.  
URL <https://doi.org/10.5281/zenodo.8431056>
- [13] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, C. Holler, *The fuzzing book*, <https://www.fuzzingbook.org> (2019).
- [14] X. Zhu, S. Wen, S. Camtepe, Y. Xiang, Fuzzing: A survey for roadmap, *ACM Computing Surveys* 54 (11s) (sep 2022). doi:10.1145/3512345.  
URL <https://doi.org/10.1145/3512345>
- [15] P. Godefroid, Fuzzing: Hack, art, and science, *Communications of the ACM* 63 (2) (2020) 70--76.
- [16] M. Harman, B. F. Jones, Search-based software engineering, *Journal of Information & Software Technology* 43 (14) (2001) 833--839.
- [17] M. Harman, S. A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, *ACM Computing Surveys (CSUR)* 45 (1) (2012) 11.
- [18] P. McMinn, Search-based software testing: Past, present and future, in: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, IEEE, 2011, pp. 153--163.

- [19] G. Guizzo, S. Panichella, Fuzzing vs sbst: Intersections & differences, *ACM SIGSOFT Software Engineering Notes* 48 (1) (2023) 105--107.
- [20] M. Kim, Q. Xin, S. Sinha, 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, ISSTA 2022, Association for Computing Machinery, New York, NY, USA, 2022*, p. 289--301. doi:10.1145/3533767.3534401. URL <https://doi.org/10.1145/3533767.3534401>
- [21] V. Garousi, D. Pfahl, J. M. Fernandes, M. Felderer, M. V. Mäntylä, D. Shepherd, A. Arcuri, A. Coşkunçay, B. Tekinerdogan, Characterizing industry-academia collaborations in software engineering: evidence from 101 projects, *Empirical Software Engineering* 24 (4) (2019) 2540--2602.
- [22] A. Arcuri, J. P. Galeotti, Enhancing Search-based Testing with Testability Transformations for Existing APIs, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31 (1) (2021) 1--34.
- [23] O. Nourry, G. BAVOTA, M. LANZA, Y. KAMEI, The human side of fuzzing: Challenges faced by developers during fuzzing activities, *ACM Transactions on Software Engineering and Methodology* (2023).
- [24] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, 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.
- [25] M. Zhang, A. Arcuri, Y. Li, Y. Liu, K. Xue, White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study, *ACM Transactions on Software Engineering and Methodology* 32 (5) (2023) 1--38.
- [26] I. Salman, A. T. Misirli, N. Juristo, Are students representatives of professionals in software engineering experiments?, in: *2015 IEEE/ACM 37th IEEE international conference on software engineering*, Vol. 1, IEEE, 2015, pp. 666--676.
- [27] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, M. Oivo, Empirical software engineering experts on the use of students and professionals in experiments, *Empirical Software Engineering* 23 (2018) 452--489.
- [28] M. C. Davis, E. Aghayi, T. D. LaToza, X. Wang, B. A. Myers, J. Sunshine, What's (not) working in programmer user studies?, *ACM Transactions on Software Engineering and Methodology* (2023).
- [29] A. J. Ko, T. D. LaToza, M. M. Burnett, A practical guide to controlled experiments of software engineering tools with human participants, *Empirical Software Engineering* 20 (2015) 110--141.
- [30] L. Briand, D. Bianculli, S. Nejati, F. Pastore, M. Sabetzadeh, The case for context-driven software engineering research: generalizability is overrated, *IEEE Software* 34 (5) (2017) 72--75.

- [31] R. T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. thesis, University of California, Irvine (2000).
- [32] A. Belhadi, M. Zhang, A. Arcuri, Evolutionary-based Automated Testing for GraphQL APIs, in: Genetic and Evolutionary Computation Conference (GECCO), 2022.
- [33] A. Arcuri, RESTful API Automated Test Case Generation, in: IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2017, pp. 9--20.
- [34] A. Arcuri, Test suite generation with the Many Independent Objective (MIO) algorithm, *Information and Software Technology* 104 (2018) 195--206.
- [35] M. Zhang, A. Belhadi, A. Arcuri, Javascript instrumentation for search-based software testing: A study with restful apis, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2022.
- [36] M. Zhang, A. Belhadi, A. Arcuri, Javascript sbst heuristics to enable effective fuzzing of nodejs web apis, *ACM Transactions on Software Engineering and Methodology* (2023).
- [37] A. Arcuri, Automated black-and white-box testing of restful apis with evomaster, *IEEE Software* 38 (3) (2020) 72--78.
- [38] A. Panichella, F. Kifetew, P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, *IEEE Transactions on Software Engineering (TSE)* 44 (2) (2018) 122--158.
- [39] OpenAPI/Swagger, <https://swagger.io/>.
- [40] P. K. Lehre, X. Yao, Runtime analysis of (1+1) ea on computing unique input output sequences, in: IEEE Congress on Evolutionary Computation (CEC), 2007, pp. 1882--1889.
- [41] A. Arcuri, J. P. Galeotti, Handling SQL databases in automated system test generation, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29 (4) (2020) 1--31.
- [42] A. Arcuri, M. Zhang, J. P. Galeotti, Advanced white-box heuristics for search-based fuzzing of rest apis, *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2024). doi:10.1145/3652157.  
URL <https://doi.org/10.1145/3652157>
- [43] B. Korel, Automated software test data generation, *IEEE Transactions on software engineering* 16 (8) (1990) 870--879.
- [44] A. Golmohammadi, M. Zhang, A. Arcuri, Testing restful apis: A survey, *ACM Transactions on Software Engineering and Methodology* (aug 2023). doi:10.1145/3617175.  
URL <https://doi.org/10.1145/3617175>

- [45] E. Viglianisi, M. Dallago, M. Ceccato, Resttestgen: Automated black-box testing of restful apis, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2020.
- [46] V. Atlidakis, P. Godefroid, M. Polishchuk, Restler: Stateful REST API fuzzing, in: ACM/IEEE International Conference on Software Engineering (ICSE), 2019, p. 748–758.
- [47] P. Godefroid, B.-Y. Huang, M. Polishchuk, Intelligent rest api data fuzzing, in: ACM Symposium on the Foundations of Software Engineering (FSE), ESEC/FSE 2020, ACM, 2020, p. 725–736.
- [48] S. Karlsson, A. Causevic, D. Sundmark, Quickrest: Property-based test generation of openapi described restful apis, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2020.
- [49] N. Laranjeiro, J. Agnelo, J. Bernardino, A black box tool for robustness testing of rest services, IEEE Access 9 (2021) 24738–24754.
- [50] A. Martin-Lopez, S. Segura, 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.
- [51] H. Wu, L. Xu, X. Niu, C. Nie, Combinatorial testing of restful apis, in: ACM/IEEE International Conference on Software Engineering (ICSE), 2022.
- [52] P. Godefroid, D. Lehmann, M. Polishchuk, Differential regression testing for rest apis, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 312–323.
- [53] V. Atlidakis, P. Godefroid, M. Polishchuk, Checking security properties of cloud service rest apis, in: IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2020, pp. 387–397.
- [54] Z. Hatfield-Dodds, 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.
- [55] L. Veldkamp, M. Olsthoorn, A. Panichella, Grammar-based evolutionary fuzzing for json-rpc apis, in: 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT), IEEE, 2023, pp. 33–36.
- [56] L. Veldkamp, Blockchains and security: Grammar-based evolutionary fuzzing for json-rpc apis and the division of responsibilities, Master’s thesis, Delft University of Technology (2022).
- [57] Openrpc, <https://spec.open-rpc.org/>.
- [58] C. Zhang, Y. Yan, H. Zhou, Y. Yao, K. Wu, T. Su, W. Miao, 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.



- [59] M. Brunetto, G. Denaro, L. Mariani, M. Pezzè, On introducing automatic test case generation in practice: A success story and lessons learned, *Journal of Systems and Software* 176 (2021) 110933.
- [60] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, 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.
- [61] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, 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.
- [62] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, 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.
- [63] G. Petrović, M. Ivanković, State of mutation testing at google, in: *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2018, pp. 163–171.
- [64] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, D. Ding, Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study, *IEEE Transactions on Software Engineering (TSE)* (2018).
- [65] W. Lam, P. Godefroid, S. Nath, A. Santhiar, S. Thummalapenta, Root causing flaky tests in a large-scale industrial setting, in: *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, ISSTA 2019, 2019, p. 101–111.
- [66] E. Daka, J. Campos, G. Fraser, J. Dorn, 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.
- [67] D. I. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, A. C. Rekdal, A survey of controlled experiments in software engineering, *IEEE transactions on software engineering* 31 (9) (2005) 733–753.
- [68] J. Siegmund, N. Peitek, S. Apel, N. Siegmund, Mastering variation in human studies: The role of aggregation, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30 (1) (2020) 1–40.
- [69] L. Briand, D. Falessi, S. Nejati, M. Sabetzadeh, T. Yue, Traceability and sysml design slices to support safety inspections: A controlled experiment, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23 (1) (2014) 1–43.
- [70] G. Scanniello, C. Gravino, M. Risi, G. Tortora, G. Dodero, Documenting design-pattern instances: A family of experiments on source-code comprehensibility, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24 (3) (2015) 1–35.

- [71] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated unit test generation really help software testers? a controlled empirical study, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24 (4) (2015) 1--49.
- [72] P. Paulweber, G. Simhandl, U. Zdun, Specifying with interface and trait abstractions in abstract state machines: A controlled experiment, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30 (4) (2021) 1--29.
- [73] A. Santos, S. Vegas, O. Dieste, F. Uyaguari, A. Tosun, D. Fucci, B. Turhan, G. Scanniello, S. Romano, I. Karac, et al., A family of experiments on test-driven development, *Empirical Software Engineering* 26 (2021) 1--53.
- [74] S. Vegas, O. Dieste, N. Juristo, Difficulties in running experiments in the software industry: experiences from the trenches, in: 2015 IEEE/ACM 3rd International Workshop on Conducting Empirical Studies in Industry, IEEE, 2015, pp. 3--9.
- [75] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, E. Aftandilian, Measuring github copilot's impact on productivity, *Communications of the ACM* 67 (3) (2024) 54--63.
- [76] T. E. Vos, B. Marín, M. J. Escalona, A. Marchetto, A methodological framework for evaluating software testing techniques and tools, in: 2012 12th international conference on quality software, IEEE, 2012, pp. 230--239.
- [77] H. Jiang, Y. Chen, Comparison of different techniques of web gui-based testing with the representative tools selenium and eyesel (2017).
- [78] H. Do, S. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, *Empirical Software Engineering* 10 (4) (2005) 405--435.
- [79] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, *Experimentation in software engineering*, Springer, 2012.
- [80] B. Kitchenham, S. Linkman, D. Law, Desmet: a methodology for evaluating software engineering methods and tools, *Computing & Control Engineering Journal* 8 (3) (1997) 120--126.
- [81] B. Kitchenham, L. Pickard, S. L. Pfleeger, Case studies for method and tool evaluation, *IEEE software* 12 (4) (1995) 52--62.
- [82] A. Arcuri, M. Zhang, A. Belhadi, Bogdan, J. P. Galeotti, Seran, A. Gol, A. M. López, A. Aldasoro, A. Panichella, K. Niemeyer, Emresearch/evomaster: 1.6.1 (Apr. 2023). doi:10.5281/zenodo.7821550.  
URL <https://doi.org/10.5281/zenodo.7821550>
- [83] T. C. Lethbridge, S. E. Sim, J. Singer, Studying software engineers: Data collection techniques for software field studies, *Empirical Software Engineering (EMSE)* 10 (3) (2005) 311--341.

- [84] K.-J. Stol, B. Fitzgerald, The abc of software engineering research, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27 (3) (sep 2018). doi:10.1145/3241743.  
URL <https://doi.org/10.1145/3241743>
- [85] R. Likert, A technique for the measurement of attitudes., *Archives of psychology* (1932).
- [86] M. Eck, F. Palomba, M. Castelluccio, A. Bacchelli, Understanding flaky tests: The developer’s perspective, in: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830--840.
- [87] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, R. Oliveto, How does code readability change during software evolution?, *Empirical Software Engineering* 25 (2020) 5374--5412.
- [88] V. Garousi, K. Petersen, B. Ozkan, Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review, *Information and Software Technology (IST)* 79 (2016) 106--127.
- [89] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspán, C. Shanbhag, Dapper, a large-scale distributed systems tracing infrastructure (2010).
- [90] B. Marculescu, M. Zhang, A. Arcuri, On the faults found in rest apis by automated test generation, *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31 (3) (2022) 1--43.