

On the faults found in REST APIs by Automated Test Generation

BOGDAN MARCULESCU, Kristiania University College, Norway

MAN ZHANG, Kristiania University College, Norway

ANDREA ARCURI, Kristiania University College and Oslo Metropolitan University, Norway

RESTful web services are often used for building a wide variety of enterprise applications. The diversity and increased number of applications using RESTful APIs means that increasing amounts of resources are spent developing and testing these systems. Automation in test data generation provides a useful way of generating test data in a fast and efficient manner. However, automated test generation often results in large test suites that are hard to evaluate and investigate manually.

This paper proposes a taxonomy of the faults we have found using search-based software testing techniques applied on RESTful APIs. The taxonomy is a first step in understanding, analyzing, and ultimately fixing software faults in web services and enterprise applications. We propose to apply a density-based clustering algorithm to the test cases evolved during the search, to allow a better separation between different groups of faults. This is needed to enable engineers to highlight and focus on the most serious faults.

Tests were automatically generated for a set of 8 case studies, 7 open-source and 1 industrial. The test cases generated during the search are clustered based on the reported last executed line and based on the error messages returned, when such error messages were available. The tests were manually evaluated to determine their root causes and to obtain additional information.

The paper presents a taxonomy of the faults found based on the manual analysis of 415 faults in the 8 case studies, and proposes a method to support the classification using clustering of the resulting test cases.

ACM Reference Format:

Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2020. On the faults found in REST APIs by Automated Test Generation. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (August 2020), 42 pages. <https://doi.org/tbd>

1 INTRODUCTION

The Representational State Transfer (REST) is an architectural style for distributed hypermedia systems [16]. REST is an abstraction that ignores details of implementation and focuses on roles of components, constraints upon the interactions between components, and their interpretation of the data. This allows a degree of flexibility that has led to the widespread use of REST to provide web services (e.g., Google¹, Amazon², and Twitter³).

The flexibility that has enabled the widespread use of RESTful web services, however, also makes testing such systems more difficult [11, 12]. While some constraints are defined as part of REST, most of the business logic is up to individual developers. Moreover, REST allows web services to call upon other web services [16], further improving their usefulness and flexibility, but complicating testing.

¹<https://developers.google.com/drive/v2/reference/>

²<http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html>

³<https://dev.twitter.com/rest/public>

Authors' addresses: Bogdan Marculescu, Kristiania University College, Oslo, Norway; Man Zhang, Kristiania University College, Oslo, Norway; Andrea Arcuri, Kristiania University College and Oslo Metropolitan University, Oslo, Norway.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

1049-331X/2020/8-ART111 \$15.00

<https://doi.org/tbd>

Automation provides a useful way of generating large amounts of test data in a fast and efficient manner, and efforts to apply automation to the problem are ongoing. In particular, search-based techniques [18] have been applied to an increasing extent to the problem of software testing [19]. However, many of the faults found as a result of these efforts may not be critical. It is important to provide engineers with an overview of the different failure types that automated systems have revealed, and the tools to identify and prioritize the more important of those faults. An automated test generation tool could find tens or even hundreds of faults, but many of those could be of low priority and similar to each other. Critical faults could get lost among those, if engineers do not have the time to manually analyze all the found faults.

In this paper, we propose a taxonomy for software faults found by automatically generated tests in RESTful web services. This was based on manual analysis of 415 detected faults. In addition, we propose a method to automatically support such a classification, by clustering test cases based on the similarities between their error messages. The goal of this method is to provide engineers with a summary of the categories of faults contained in the test cases and support their decision regarding the relative priority of found faults.

The clustering of test cases is implemented as an extension of EvoMASTER [3, 5, 9], an open-source test case generation tool that supports both black-box and white-box testing [6]. EvoMASTER uses the Many Independent Objectives algorithm (MIO) [4] to generate test cases. Lines of code and potential faults are labeled as separate objectives by MIO, and tests are generated to maximize the number of objectives covered by the test suite. EvoMASTER is primarily focused on RESTful APIs. It generates and runs HTTP calls with generated input data in order to maximize code coverage and fault detection. Besides evolving HTTP calls, the tool can also create SQL data directly into the databases [8].

In this paper, EvoMASTER is used to generate test suites for a total of 8 case studies (7 open-source and 1 industrial). The resulting test suites are used to create the taxonomy presented in this paper and to evaluate the applicability and usefulness of the proposed clustering.

The rest of the paper is structured as follows: in Sections 2 and 3 we discuss the background and related work, respectively. Section 4 describes the research questions and method. Results are presented in Section 5 for the proposed taxonomy, and Section 6 for the results related to automated classification support. Section 7 contains a discussion on the results and considerations related to future work, while Section 8 discusses the threats to validity. Finally, Section 9 presents our conclusions.

2 BACKGROUND

The Hypertext Transfer Protocol (HTTP) is a protocol that defines a set of rules for data communication over a network. HTTP communication is carried out as an exchange of structured messages: a client sends a message requesting that a specified *action* to be performed on specified *resources*. HTTP actions are defined by a set of HTTP verbs (e.g. POST, PUT, GET, or DELETE) that allow users to create, retrieve, update, and delete the specified resource (CRUD operations). The server returns a response to the client's request, accompanied by a status code. The response can be:

- **1xx**: the client's request has been received, continuing process. This is used for actions like switching of protocol (e.g., from HTTP to HTTPS).
- **2xx**: the client's request has been received and successfully carried out. This can be simply a return code, or can be accompanied by additional information, including resources values.
- **3xx**: the client is required to perform additional actions in order to complete the operation. Typically used for redirection.
- **4xx**: the client's request contains errors or cannot be successfully performed. Statuses in this family indicate that the error lies with the client: requesting resources that cannot be found, or that are unavailable due

to some other limitations. In practice, a client would expect such a response to also contain precise error information, in order to allow them to access the service correctly.

- **5xx**: the client's request appears to be valid, yet the server was unable to complete it. Failure to complete an apparently valid request is considered a fault in the server. Note that such a message is not expected to contain detailed information. As a 5xx message would seem to indicate a failure of the API, detailed error messages might contain information about the internal state of the API, such as stack traces. This would be undesirable, as it would constitute a security vulnerability.

Representational State Transfer (REST) is an architectural style that allows the development of web services on top of HTTP. A web service using the REST style (RESTful web service) should, ideally, follow the guidelines proposed by that style: a client-server architecture that separates the user interface from data handling; a stateless communication approach; and limitations on the semantics of operations associated with each of the actions or HTTP verbs. However, REST is an architectural style rather than a protocol. In practice, while such guidelines are proposed and recommended, they are not enforced. This leads to the possibility of non-conforming APIs being developed and used.

OpenAPI⁴ is a programming language-agnostic interface description for REST APIs, the purpose of which is to allow machine and human clients to understand the capabilities of a web service, and the means of interacting with it, without requiring access to its source code, additional documentation, or other data sources. OpenAPI describes an API service using a structured document that specifies what action are available on the service, what input parameters are required by the service and what responses the client can expect, as well as any authentication information might be needed. Earlier versions of OpenAPI (i.e., earlier than 3.0.0) were also known as Swagger. In this study, we refer to both earlier and current versions as OpenAPI in the interest of clarity.

Since OpenAPI is a broadly adopted industry standard, frameworks for building RESTful APIs (e.g., Spring⁵) increasingly include support for automatically generating OpenAPI descriptions from the source code. The automated generation of OpenAPI specifications has a number of benefits over manual development. By generating the OpenAPI specifications automatically, such frameworks ensure that the resulting document is updated every time changes are made to the underlying source. Thus, automatically generated OpenAPI specifications are more likely to be maintained up to date, and to conform to the actual behavior of the API than manually developed and maintained equivalents. The downside is that some of the semantic information related to specific operations may be lost by automatically generating the OpenAPI specification. For example, an operation that returns a collection of objects with a specified schema or that accepts only a limited number of values may be associated with an OpenAPI schema that just specifies a collection of generic objects. The trade-off between the ease of generation and maintenance of automatically generated schemas, and the fine tuning possible with manually developed counterparts, means that both approaches can be encountered in existing RESTful APIs.

When building RESTful APIs, it is a common practice to use *wrapped responses*. Instead of just returning JSON body payloads, those can be “wrapped” inside a JSON object having at least these two fields: *data* and *errors*. If there was no error, then the body payload will be inside the *data* field. Otherwise, error messages will be present in the *errors* field (while *data* will be left to null). The reason to do this is that 4xx/5xx status codes do not provide enough information to the users, especially generic error codes like 400. For example, the default behavior of the Spring framework is to wrap failed calls when automatically generating responses for 5xx errors.

For security reasons, to prevent any information regarding the internal state of the system from being accessible outside the API, detailed info in the wrapped responses could be omitted (e.g., full stack-traces when the failures are due to thrown exceptions). Then, the only information accessible to the client is that a 5xx error was encountered, but all other information regarding the state of the API would be hidden. Such behaviors can be configured for

⁴<https://www.openapis.org/>

⁵<https://spring.io/>

testing, and a more verbose and informative output can be provided for the purposes of testing. However, this would not be available when conducting black-box testing of remote services. This causes difficulties for black box testing, especially testing micro-services where direct access to the code of external APIs being called upon is partially or completely unavailable.

EvoMASTER⁶ [3, 9] is an automated tool that uses evolutionary algorithms and dynamic program analysis to automatically generate system level test cases for RESTful APIs. EvoMASTER starts from a population of random tests and improves them using the Many Independent Objectives (MIO) algorithm [4], to maximize measures such as code coverage and fault detection. Code coverage is evaluated at runtime by instrumenting the JVM code. Potential faults considered for fault finding are based on HTTP responses with 500 status codes and discrepancies between the API responses and what is expected based on the OpenAPI schemas.

In searching for system level tests for RESTful API, EvoMASTER identifies lines of code, branch coverage, and potential faults as all possible objectives for the search. Each candidate test case is evaluated with respect to each of the objectives. Once an objective is met, a single, best, test case is maintained, and the search continues. The search stops when all objectives have been met, or, more likely when the limit of available resources is reached. The resources available for the search can be defined in terms of time available for the search or in terms of maximum number of HTTP calls evaluated for fitness.

Individuals are test cases, each consisting of one or more HTTP calls to the underlying System Under Test (SUT), along with the relevant parameters. Mutation operators act upon the individual parameters, starting from objects and down to the level of basic types. A distinction is made between necessary and optional parameters, to enable successful calls.

3 RELATED WORK

The popularity of RESTful APIs has led to a significant amount of frameworks that support development and deployment, and more recently, in research on testing such systems. Efforts are mostly focused on black-box automated test case generation, based on the OpenAPI schema definitions [10, 13, 20, 30].

QuickREST, proposed by Karlsson et al. [20] uses the OpenAPI specification to derive properties of inputs, and then uses the specification as an oracle to determine the correct behavior of the system. The authors describe the faults they identified as “input validation bugs”. One of the problems identified by the authors is that APIs are under-specified, i.e., the restrictions that are defined by the OpenAPI specification are looser than the valid inputs of the underlying API. The result is that the generators have to be modified by the user, especially in order to generate valid inputs.

Viglianisi et al. proposed RESTTESTGEN [30], a tool that uses the OpenAPI specification, along with runtime information, to identify dependencies between resources and the corresponding operations. These dependencies are then expressed as a *Operation Dependency Graph*.

Other efforts involve using the OpenAPI specification to derive models of the RESTful API, that can then be used to generate test cases, as proposed by Ed-douibi [13]; or as a basis for fuzzing RESTful APIs, as proposed by Atlidakis et al. [10].

Sampath et al. [26] seed five types of faults, which they describe as data store, logic faults, form faults, appearance faults, and link faults. Due to the way EvoMASTER interacts with the system under test (SUT) and evolves test cases, the classification is not entirely applicable. EvoMASTER does not focus on appearance, but rather on the direct interaction with the API, and bases test data generation on what the OpenAPI schemas deem valid.

Elbaum et al. [14] identify both the need for better testing for web systems and the need to automate it, but rely on existing user-session data. The benefit of evolutionary algorithms in general, and EvoMASTER in particular, is

⁶<https://evomaster.org>

that the software tests that are generated automatically are often different from those developed by test engineers or users.

Recent work by Hajri et al. [17] seeks to accomplish a similar goal: automating classification of test cases. That work, however, is applied in the context of use case-driven testing, and focuses on product lines. Nevertheless, it does emphasize the need for automated support to enable developers and test engineers to better understand the faults that are being found and to more effectively sift through large amounts of automatically generated data.

Nguyen et al. [24] also investigate the issue of using clustering to facilitate the process of understanding, classifying, and ultimately fixing faults. Their work uses the DBSCAN clustering algorithm to classify 300 test case failures from 3 projects. The paper proposes a classifier, but no generalizable taxonomy of faults that has been found.

Usman et al. [28] present the application of taxonomies for software engineering and describe some of their uses and benefits. Ralph [25] further elaborates on the uses of taxonomies. The two studies identify the following benefits: providing a common terminology for knowledge sharing, enabling a better understanding of the relevant concepts in a particular field and of the relationships between them, and support for identifying gaps in existing knowledge or for decision support. Overall, taxonomies “increase cognitive efficiency and facilitate inferences” [25] and allow the inclusion of counter-intuitive processes and relationships that are, nevertheless, based on data. Basing an understanding of the expected and encountered faults in RESTful APIs on the data derived from EVOMASTER should enable the development, and later refinement, of a useful fault taxonomy.

From this review of existing work, two major knowledge gaps emerge. First, it is not clear what types of faults can be found with automated testing. The studies cited rely on user session data or seeded faults. A taxonomy of faults found by automated means would provide several benefits. This would allow test engineers and developers to understand and classify the faults found. It would also allow for a more thorough understanding of the capabilities and limitations of automated tests, and the extent to which they overlap with those of other types of testing. Finally, it would allow a better understanding of where automated testing tools could fit in the overall software engineering process, and how the tests generated by automated means could best be used. The second gap is that emerges is the absence of a more general means of supporting developers in classifying test data, especially test data that comes from novel sources. Classifiers do exist, but they are trained on either user data or on seeded data. The behavior of such classifiers when encountering novel test data is not entirely known. The need to provide even rudimentary support for root cause analysis of novel faults starts with an assessment of how similar these faults are to each other and to known faults.

4 METHOD

Section 3 identified two main knowledge gaps: the kinds of faults that are found as a result of automated testing, and the lack of a mechanism to help test engineers classify and analyze the large amounts of test cases and test data that could be generated by automated test tools. To address these gaps, we propose the following research questions.

4.1 Research Questions

- (1) What faults are actually found in RESTful systems by automated testing?
- (2) What types of faults are more common across the SUTs being studied?
- (3) How can automated support for the classification of faults be provided?
 - (a) To what extent is automated support for fault classification useful in distinguishing between different faults?
 - (b) To what extent is a subset of test cases derived from sampling each automatically identified category (i.e., cluster) representative of the categories of faults that are found in practice?

4.2 What faults are actually found in RESTful systems by automated testing?

Ideally, we would want to have available an overall taxonomy of software faults. Such a taxonomy would allow us to classify the types of faults that automated systems find in RESTful APIs and compare them against the categories of faults that other approaches find. This would provide a useful means of evaluating automated testing tools in terms of the classes of faults they discover.

To the best of our knowledge, no such taxonomy exists, so creating a taxonomy of faults found by automated testing in RESTful systems would provide an answer to this question. To create such a taxonomy we have run an automated software testing system, EvOMASTER, on a set of eight case studies. The case studies are all RESTful APIs. Seven are open-source systems, out of which four are original and three are artificial (i.e., showcase examples). The eighth is an industrial case study. While we do not claim that the case studies we have selected are a representative sample of all RESTful APIs, they do provide a good starting point for formulating an initial taxonomy.

While such a taxonomy is, undoubtedly, incomplete and subject to revisions and additions, it may provide a first step towards a more systematic understanding of the faults present in software systems in general, and in RESTful web services in particular.

4.3 Automated support for the classification of faults

This question is focused towards usability of automated software testing systems. Automated software testing systems have the potential to generate large amounts of test data. In particular, the automated software testing system used for the present study, EvOMASTER, seeks to maximize the amount of code coverage of the overall test suite. This leads to comparatively large test suites, that are difficult to analyze and classify manually. As a result, the question arises: how can automated support be provided to help test engineers classify the resulting test cases? For the rest of this section, we will refer to the classes of faults proposed by automated means as “clusters”, to differentiate them from the classification resulting from manual analysis.

Should a form of automated support be provided, to what extent is that support capable of distinguishing between different classes of faults, what characteristics affect its performance, and how can such an automated support mechanism be evaluated on an arbitrary case study?

In order for such an automated support mechanism to be useful, it would first have to be consistent between runs, i.e., the clusters of faults suggested should be consistent on repeated executions. It is to be expected that some differences between runs will exist, as the search itself is non-deterministic and any further clustering mechanism would have the search results as its starting point. Nevertheless, an evaluation of the stability of the clusters provided by any automated classification support mechanism is necessary.

Second, any automated support mechanism for fault classification would have to rely on a set of criteria that is general enough to be applicable, but specific enough to differentiate between different faults. This is a difficult balance to accomplish, but a useful starting point could be the process by which developers and test engineers conduct the classification manually: i.e., starting from available data such as any error messages provided as a result of software failures. Providing useful defaults values for settings and parameters would be an essential component in proposing an automated mechanism for fault classification.

Third, given a sensible set of default values, there is a need to evaluate an automated fault classification mechanism. More precisely, can the classification mechanism discriminate between different faults. This can be judged by assessing the *cohesion* and *separation* of the clusters of faults that the automated mechanism proposes, as evaluated against the classification provided by manual means.

- **Separation** is a measure of the degree to which faults of the same class are present in the same cluster. Even if imperfect, any automated clustering should be able to ensure that similar faults are grouped together in the same cluster. If this is the case, it could mean that there is some similarity between faults that may

be identified and used automatically. If similar faults are not in the same cluster, it could indicate that clustering is not a useful tool.

- **Cohesion** is a measure of the degree to which faults in the same cluster are similar to each other, i.e., they are of the same class. If a cluster contains many faults of different classes (but the previous point is true, i.e., those classes are not present in other clusters), it could indicate that the clustering being used currently is not sufficient. This could mean that the current clustering approach could be further investigated to determine more suitable parameters, or that more clustering approaches could yield better results.

There are several benefits to providing even an imperfect means of automated clustering of faults found in RESTful API. Even an imperfect automated clustering could provide some of the benefits of the fault taxonomy, in particular the cognitive efficiency and the ability to quickly evaluate test suites and test results, at a fraction of the cost of manual classification. This could be used for summarizing the findings of the search-based tests to test engineers and could be useful in terms of defining how the automated testing systems fit in the software development/testing process. Transferring relevant information about the categories of faults found, in an effective manner, could be an important step in enabling usage in industry.

In particular, it would be interesting to see if a set of parameters can be found that offer a balance between identifying all the fault types and reducing the number of clusters to only relevant categories. If a singular set of “best-fit” parameters cannot be determined for all case studies, then a means of determining appropriate values would be needed. In addition, automated clustering could be used as an initial evaluation of the performance of the search-based system on a new case study.

Overall, a determination of the usefulness of clustering can only be performed in an industrial setting. But experimental results could show whether or not clustering is potentially viable, as a first step towards a more applied validation.

4.4 Empirical Study Setting

To answer our research questions, we used EvoMASTER as our automated software testing tool. As discussed in Section 2, EvoMASTER is an open-source⁷, white/black box, search-based tool that automatically generates system level test cases for RESTful APIs. EvoMASTER has several parameters that can be configured. For this study, we used its default settings. The only exception is that we used a different search budget, as the current default of 1 minute is only for demonstration purposes.

For the purpose of this study, we define a “potential fault” (more commonly referred to simply as a fault) to be an HTTP call that results in a failed result, i.e., a returned status code 500. Note that a return status code of 500 is not always due to a software fault. For example, if a service *A* relies on a service *B* to complete its requests, if for any reason *B* is down, then *A* would return a 500 (*Internal Server Error*) code, although there is no software fault in *A*. The code 500 just means that the service cannot complete the requests, due to a problem that is not dependent to the client (which otherwise would have resulted in a code in the 4xx family).

However, when the tester has full control on the environment of the SUT, 500 has high chances of representing an actual software fault. For example, when there are uncaught software exceptions (e.g., due to null pointer accesses), instead of crashing the whole HTTP server, most frameworks (e.g., Spring) would just craft a 500 response. In practice, the exact decision as to whether or not a potential fault is an actual fault is left to the discretion of the test engineers.

In addition, besides looking at 500 status codes, we consider an HTTP call that results in a code that is not supported by the OpenAPI specification to also be erroneous, even though such calls may not result in system failures (e.g., returning a 400 when such code is not explicitly stated in the schema). Note that codes in the 5xx family are technically unexpected. However, since we are already considering 500 as a potential fault, the

⁷<https://github.com/EMResearch/EvoMaster>

SUT	Classes	LOCs	Endpoints
<i>rest-ncs</i>	9	602	6
<i>rest-scs</i>	13	859	11
<i>rest-news</i>	10	718	7
<i>catwatch</i>	69	5442	13
<i>feature-service</i>	23	2347	18
<i>proxyprint</i>	68	7534	74
<i>scout-api</i>	75	7479	49
<i>ind0</i>	75	4573	20
<i>Total</i>	342	29554	198

Table 1. Overview of case studies used in this study. The table shows, respectively, the number of Java classes, the number of lines of code (LOCs), and the number of REST endpoints for each SUT.

“Expectation Faults” category only includes responses that have status codes that are not supported by the OpenAPI specification, but are not 500.

The result of the search is a runnable, system level, test suite. All tests in the suite are independent of each other, and all tests in the suite should pass, as they capture the current behavior of the system, even when returning a 500 status code.

In this work we used seven open-source case studies (see Table 1), from the EvoMASTER Benchmark (EMB): a collection of open-source RESTful APIs⁸. The benchmark has been used in previous evaluations of EvoMASTER itself and of modifications or additions relating to search-based software testing [3, 5, 31].

Two of these open-source case studies, the REST Numerical Case Study (*rest-ncs*) and REST String Case Study (*rest-scs*), are artificial examples of numerical and string problems, previously used in studies on unit testing [2, 7]. REST News (*rest-news*) is an artificial API developed as part of a course on enterprise development⁹. The remaining four, *feature-service*, *proxyprint*, *scout-api*, and *catwatch* are real, open-source, RESTful web services, hosted on GitHub.

In addition to the seven open-source case studies, we also used an industrial RESTful API that will be referred to in this work as *Ind0*. This case study provides a useful example of the industry standard and highlights any potential differences between industrial and open-source case studies, from the perspective of automated software testing in general, and EvoMASTER in particular. Details regarding the relative complexity of these systems can be found in Table 1.

Search-based methods include a significant stochastic element, as evolution is not a deterministic process. For the purposes of evaluation, we ran EvoMASTER on each of the SUTs for a total of 10 runs, with identical parameters and under identical conditions. During these runs, some variation could be observed in terms of coverage, generated tests, and potential faults discovered.

Table 2 shows the coverage obtained by running EvoMASTER 10 times. The values presented are the mean coverage for each category, and the minimum and maximum coverage encountered in the set of 10 runs. **Targets Covered** shows the number of search targets identified and covered during the search (which is an aggregated value of all criteria EvoMASTER optimizes for, including code coverage, HTTP status coverage and fault finding), **Line coverage** and **Branch coverage** show the mean, maximum, and minimum percentage values as measured by the code instrumentation.

⁸<https://github.com/EMResearch/EMB>

⁹https://github.com/arcuri82/testing_security_development_enterprise_systems

SUT	Targets Covered		Line Coverage (%)		Branch Coverage (%)	
catwatch	1158.10	[1126.00,1207.00]	33.95	[32.82,34.77]	17.74	[16.73,18.68]
features-service	465.80	[411.00,528.00]	40.19	[35.21,45.77]	12.20	[9.32,16.10]
ind0	689.60	[495.00,868.00]	15.94	[9.82,21.34]	5.54	[3.16,7.18]
proxyprint	2004.50	[1764.00,2085.00]	27.66	[24.57,28.66]	11.62	[7.19,12.55]
rest-ncs	620.67	[616.00,627.00]	87.68	[87.41,88.11]	65.33	[63.79,67.24]
rest-news	340.50	[321.00,360.00]	53.26	[51.09,54.89]	26.06	[24.15,29.24]
rest-scs	733.10	[651.00,782.00]	71.50	[64.78,75.08]	42.11	[34.69,47.62]
scout-api	1834.50	[1749.00,1938.00]	38.70	[36.83,40.55]	20.52	[19.94,21.11]

Table 2. Overview of the coverage obtained during the experimental runs. The table shows, for each of the SUTs, the average coverage and (in the square brackets) the minimum and maximum values obtained for the 10 runs. **Targets Covered** shows the number of search targets covered during the search, where a target is an identified search objective. **Line Coverage** and **Branch Coverage** show the number of lines of code and branches, respectively, covered during the search, expressed as a percentage of the total.

SUT	Generated Tests		HTTP Calls		Endpoints with 500		Endpoints with Failed Expectations	
catwatch	47.40	[44.00,54.00]	169.60	[127.00,223.00]	6.00	[6.00,6.00]	8.00	[8.00,8.00]
features-service	38.80	[36.00,43.00]	74.10	[57.00,84.00]	14.00	[14.00,14.00]	16.70	[16.00,17.00]
ind0	84.70	[68.00,97.00]	111.10	[79.00,286.00]	18.50	[14.00,19.00]	18.60	[15.00,19.00]
proxyprint	259.50	[244.00,268.00]	583.20	[455.00,707.00]	40.90	[39.00,43.00]	48.20	[46.00,51.00]
rest-ncs	61.78	[56.00,65.00]	285.00	[238.00,340.00]	0.00	[0.00,0.00]	5.00	[5.00,5.00]
rest-news	29.62	[26.00,34.00]	111.00	[89.00,136.00]	2.00	[2.00,2.00]	4.75	[4.00,5.00]
rest-scs	86.10	[53.00,99.00]	415.20	[257.00,538.00]	0.00	[0.00,0.00]	9.30	[8.00,10.00]
scout-api	204.30	[189.00,222.00]	581.10	[439.00,688.00]	33.20	[33.00,34.00]	48.00	[48.00,48.00]

Table 3. Overview of the generated tests. The table shows the average value for each of the SUTs, as well as the maximum and minimum values obtained for the 10 runs. The column **Endpoints with 500** shows the number of endpoints that have returned calls with the 500 status code. The column **Endpoints with Failed Expectations** shows the number of endpoints that have returned only Expectation faults.

Table 3 shows some of the aggregate information regarding the test cases generated by EvoMASTER during the 10 experimental runs. The values displayed for all categories are: mean value, and minimum and maximum encountered (respectively, in square brackets).

Generated Tests shows the number of test cases that EvoMASTER generated. Note that tests are only kept as part of the test suite if they contribute in the covering of one or more of the test targets (i.e., removing any of these tests would result in lower target coverage). **HTTP Calls** shows the number of calls included in each of the test suites generated. Note that each test case can contain between one and ten HTTP calls.

Endpoints with 500 shows the mean, minimum, and maximum values for the number of endpoints that received answers with the 500 status code, while the **Endpoints with Failed Expectations** column shows the same values for calls that, while not causing a 500 response, fail one or more of the expectations. Note that the case studies presented here have OpenAPI schemas that are generated automatically from the code. This means that all the failed expectations described here are the result of calls that have response codes that are not explicitly supported by the OpenAPI schema. This could be the result of a genuine fault revealed as a mismatch between expected and actual behavior. Alternatively, the same behavior can be observed as a result of using framework

SUT	Total 500 Faults		Unique Paths	Unique Faults	Faults in Selected
catwatch	37.80	[27.00,49.00]	6	13	27
features-service	30.70	[24.00,36.00]	14	18	28
ind0	50.70	[32.00,108.00]	17	34	49
proxyprint	128.80	[104.00,155.00]	44	58	107
rest-ncs	0.00	[0.00,0.00]	0	0	0
rest-news	4.62	[3.00,7.00]	2	2	6
rest-scs	0.00	[0.00,0.00]	0	0	0
scout-api	188.70	[133.00,246.00]	37	70	198
Total					415

Table 4. Overview of the potential faults that were identified with EvoMASTER for each case study for the 10 runs. For each case study, the test case with the highest coverage was selected for analysis. The column **Faults in Selected** shows the number of 500-status faults (and not expectation faults) that were contained in the test cases selected for manual analysis. The column **Unique Paths** indicates the number of unique combinations of endpoint and HTTP verb present, while the column **Unique Faults** indicates the number of unique combinations of path and last executed line present in the faults included in the manually analyzed faults.

default responses for handling certain calls or endpoints. These are technically faults and do result in unwanted behavior, therefore they are recorded here.

An important element in the evaluation of search-based techniques is ensuring that comparable resources are available for each run and for each of the included test cases. To ensure a fair and generalizable evaluation, the search budget for all runs was measured in terms of the number of fitness evaluations available, as recommended by Črepinšek et al. [29]. For this work, the cost of evaluating the fitness of a test case is dependent on the number of HTTP calls it contains. As a result, all EvoMASTER runs on all case studies had the same search budget of 100,000 HTTP calls.

For each case study included, the run with the highest target coverage (which does not necessarily imply the highest number of faults) out of the 10 runs per SUT was selected for analysis. Each resulting test suite was manually analyzed. We determined the root cause for each potential fault, including tracking where in the SUT code the fault appeared to exist. For the same endpoint failing with a 500-status code response, we distinguish between possible faults based on the last executed line in the business logic of the SUT (i.e., an endpoint can fail in different ways and throw exceptions due to different faults in its code). A tentative class was assigned to each fault, based on the identified root cause. The classes found in the test suites formed the basis for the taxonomy of faults presented below, while the method for analyzing and determining the root cause forms the basis for the proposed fault classification method.

Table 4 shows the number of faults present in the test suites generated by EvoMASTER over the 10 experimental runs. The column **Faults in Selected** shows the number of faults resulting from calls with responses with code 500 that were present in the test cases selected for manual analysis, for each of the case studies. Note that, while other types of faults will be discussed, the manual analysis focuses mostly on these 415 faults.

Distinguishing between different faults is an interesting discussion of its own. Note that, during the search, the same fault may be discovered more than once. If the new test case covers additional search objectives, it will be kept, even if one or more of the faults it finds are duplicated. To get a better understanding of the distinct faults found, EvoMASTER maintains statistical information on the number of separate paths that have been found that get faulty responses (i.e., responses with the code 500).

However, two calls on the same endpoint may fail in different ways, which can potentially be seen as the calls have different last executed lines. Table 4 shows the number of faults present in the test suites selected for manual analysis, according to the path and the combination of path and last executed line.

5 TAXONOMY

The purpose of this section is to provide an initial overview of the fault classification, as developed as a result of this study. The classification procedure type is qualitative, based on the root cause analysis of individual faults. The data sources that form the basis of the taxonomy consist of test cases automatically generated by EvOMASTER for the set of case studies present in the benchmark. The benchmark consists of seven open-source case studies and one industrial case study. As discussed in Section 4.4, the test cases were obtained by running EvOMASTER 10 times for each case study, using a search budget of 100,000 HTTP calls, and selecting the run with the highest number of covered objectives. Comparison based on the number of fitness evaluations ensures that a comparable amount of effort was expended for each case study and for each run, in accordance with the guidelines proposed by Črepinšek et al. [29].

The taxonomy was developed according to the method proposed by Usman et al. [28]. The first activities described by the method are an identification of the knowledge area, objective, and procedure for the taxonomy. The knowledge area is the automated software testing of RESTful APIs. The objectives are to define a set of categories that enable test engineers to classify the faults found in RESTful APIs. This type of system under test also defines the scope of the taxonomy.

The taxonomy was designed using a facet-based classification structure. The authors were unable to find an existing taxonomy of software faults, especially as applied to RESTful APIs. Thus, the hierarchy or tree classification structures might not be appropriate, as we cannot assume that the current study would yield a complete enough understanding of the topic to allow such structures. Moreover, the taxonomy presented here is based on the use of one automated testing tool, EvOMASTER, and on one set of case studies. What is needed is a classification structure that would be flexible and accommodate the inclusion of new types of faults, new case studies, and new findings and connections.

The faults found were analyzed according to several facets. The first set of facets were focused around identifying where in the SUT a potential fault is found. This includes the lines of code where a potential fault is identified, both in the test code and in the SUT, the endpoint that was called, and the HTTP verb that was used. These facets help identify where in the code the potential fault can be found. A second category of facets were focused around identifying the root cause of the fault, including the error message and the type of exception thrown (if any), as well as the result of a manual root cause analysis. This information was then used to determine a tentative class for each of the potential faults found.

The fault classification provides an overview of the types of faults found during the search, on the benchmark case studies. It is important to note that this is a first attempt at this kind of classification and, while useful for providing an overview of the types of faults encountered so far, it is by no means complete or final.

The faults identified so far fall under four major categories:

- **Configuration and Execution Faults** - faults related to the setup of frameworks, testing scaffolding, or other external systems.
- **Schema Conflicts** - mismatches between the OpenAPI schema and the underlying SUT
- **Data Integrity Faults** - initialization, data consistency, database injection faults
- **Faults in the Business Logic** - faults that reflect problems in the code dealing business logic of the underlying SUT.

Note that the faults are being structured based on their root causes. Thus, **Faults in the Business Logic** relate to the SUT itself, the code included there, and behavior relevant to the business logic. This category is the most

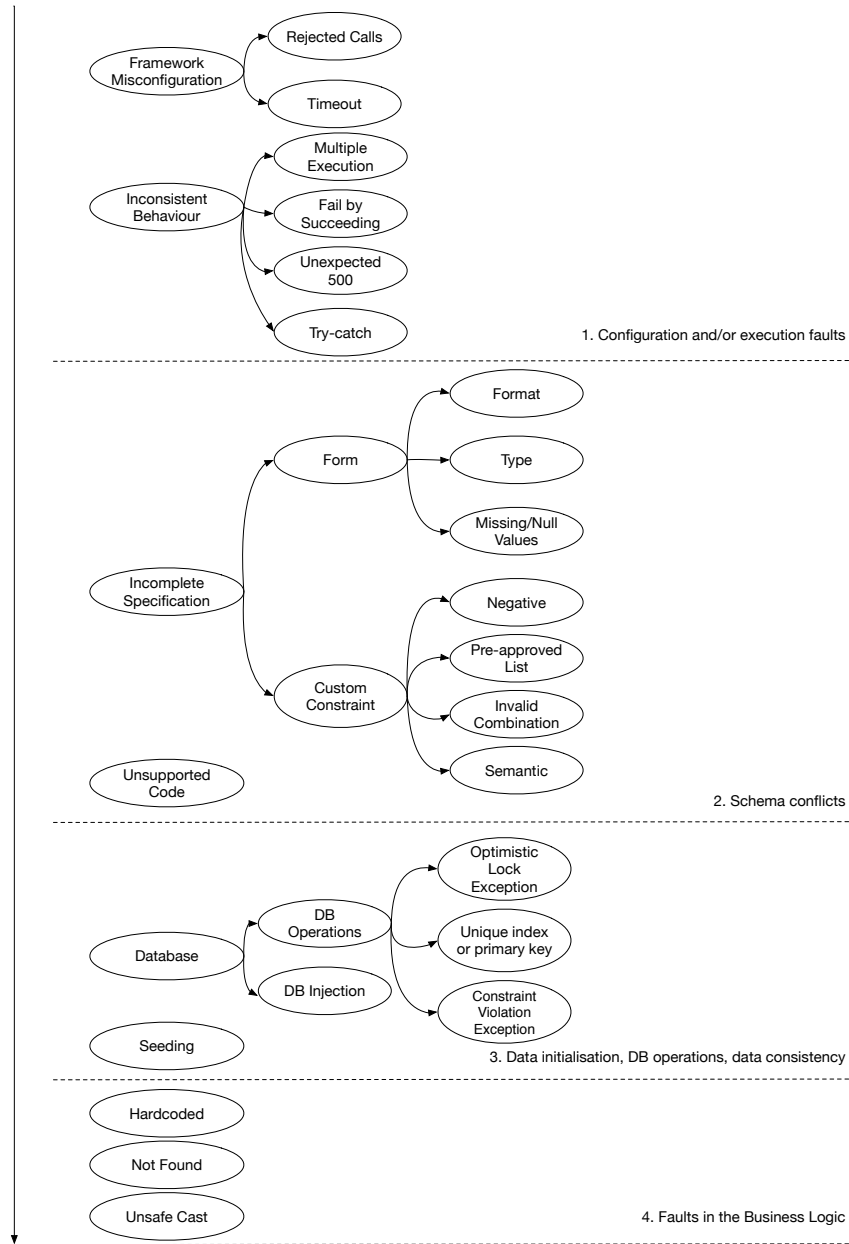


Fig. 1. An initial diagram of the types of faults discovered in the benchmark case studies using EvoMASTER.

generalizable category of faults, as the SUT itself may have a varied sub-systems along with all the software faults associated with them. **Data Integrity Faults** are also quite generalizable. This category refers to fault that have an impact on the integrity of data, in particular the integrity of databases. The data is likely the subject

of domain specific constraints, though the source of the root cause of data corruption may be more difficult to ascertain for certain. Whether the data has been corrupted due to the testing infrastructure and data that could be encountered in practical use and was not suitably handled by the SUT would have to be determined on a case-by-case basis.

Schema Conflicts, on the other hand, are specific to RESTful APIs. The category described the types of faults that arise when the schema, in this case the OpenAPI schema, is not complete or is in conflict with constraints expected by the underlying SUT. These issues are, as described here, specific for RESTful APIs, but conflicts between the schema and the behavior of the system are of concern for any web service. **Configuration and Execution Faults** are also specific to the RESTful APIs, as they describe particular frameworks and interactions between the user and the system that are typical for this type of system.

5.1 Configuration and Execution Faults

This category comprises faults that can be traced to how the SUT, EvoMASTER, or the interaction between them, have been setup. These are difficult to diagnose, as the behavior of the SUT during the search might vary greatly from the behavior of the resulting test cases. Moreover, more complex RESTful APIs can call upon other services. This raises a high potential for misconfiguration faults between the SUT and any external services it might use. Such faults would also be classified under this heading. The benchmark case studies we have used to develop the taxonomy did not use external services, so such faults are not present in the taxonomy at present. However, configuration and connection issues are quite likely, especially as services get more complex and call upon more external services. From the perspective of SUT quality, external services play a vital role in providing the necessary functionality, but they are treated as external to the SUT.

- **Framework Misconfiguration.** These are faults that are due to peculiarities in the various frameworks being used. For example, the industrial case study (*Ind0*) uses a framework that identifies all strings containing a particular character, in this case “;”, as potentially malicious and rejects them, causing a response with a 500 code (instead of correct code in the 4xx family). During the search, EvoMASTER does not exclude those characters from potential mutations, leading to strings containing those characters to be generated (and then rejected by the framework). In the specified example, the fault can be attributed to one of two causes. Either the character in question is mistakenly identified as a potential security issue in the default settings of the framework, or there is a misconfiguration of that framework by the developers of the SUT.

In practice, **Framework Misconfiguration** faults manifest as **Rejected Calls** and **Timeouts**. **Rejected Calls** are calls that are rejected by one of the frameworks being used. An example of this type of fault can be seen in the industrial case study (*Ind0*), where inputs containing particular characters are rejected by the framework as suspected malicious code. This is a very clear example of a fault that manifests at the framework level, as the SUT code is not even executed in these cases.

Timeouts are somewhat rare type of faults that results in specific calls being timed out. By default, EvoMASTER waits up to 10 seconds to get a response to its HTTP requests. Depending on the setup and configuration of the system, some calls may be timed out before they have a chance to complete. This can include calls that execute slowly and are timed out before completion, calls that are timed out and rejected for other reasons (for example, a large number of calls in a short period of time), or limitations on the server configuration (for example, a maximum number of calls being allowed per connection). An example of such a fault can be seen below.

It is interesting to note that these faults do not result in a 500 return code, but rather timeout on the TCP connection. This leads to an interesting dilemma. These are not successful calls, so EvoMASTER groups

Try-Catch Fault Example

```

try{
    ValidatableResponse res_0 = given().accept("*/*")
        .get(baseUrlOfSut + "/files/_postfix/9246-1-0-875/evomaster_1693_input/CpvxDTU01ivNi.
            cdn5S3TpA")
        .then();

    expectationHandler.expect(ems)
        .that(sco, Arrays.asList(200, 401, 403, 404)
            .contains(res_0.extract().statusCode()));
} catch (Exception e){
}

```

Fig. 2. An example of a **Try-Catch** fault. During the search, the call timed out. As a result, it is wrapped in a try-catch block. This allows the test suite to pass, but highlights that an issue has occurred during execution.

them along with potential faults, alongside calls with 500 code and expectation violations. However, such faults lack any return code. An example of this can be seen in Figure 2.

As a result of these peculiarities, such faults fall outside of the 415 faults that were selected for manual analysis. Nevertheless, they have been analyzed and are interesting to discuss, since they are clearly not successful calls and may be indicative of undesirable behavior on the part of the SUT (e.g., performance issues or side effects of the instrumentation). While the root cause of such faults is not immediately apparent, and they seem to be rare in the case studies discussed here, the argument can be made that they are symptoms of deeper problem and, as a result, we have included them in the taxonomy.

However, it might well be that the execution, for a given input, is expected to be slow, and taking more than 10 seconds. Furthermore, such performance issues might had been due to other processes running on the OS during the search, and not replicable when running the generated tests afterwards. In such cases, it would not be a fault, which makes this category hard to analyze.

- **Inconsistent Behavior.** This is a catch-all term for a variety of faults that arise from different SUT behaviors between the search and execution of the generated test cases. These inconsistencies make such problems difficult to replicate and diagnose. Some of the sub-categories included here do not, technically, meet the definition of a “potential fault”. They are, however, worth noting as potentially revealing other faults. It is left to the test engineers to make a definitive determination on the status of the items included in this category.

Included in this category (and observed in the benchmarks) are:

- **Multiple Execution** inconsistencies - where repeated executions of the same generated test case (or of the same call) appear to result in different behaviors. While none of the different behaviors might fall under the strict definition of a potential fault, such inconsistencies could be worth a more in-depth look.
- **Fail by Succeeding** or unexpected 2xx calls - call that failed during the search, but pass when the resulting test case is executed later. This is not necessarily a fault in the SUT. For example, such behavior can be the result of issues in the instrumentation of the SUT code or a shortage of available TCP ports during the search process that does not affect the later execution of the test case. However, since the call failed during the search, EvoMASTER generated the appropriate code to handle such failures.
- **Unexpected 500** - these are calls that succeeded in during the search and fail during the subsequent execution of the generated test case.
- **Try-Catch** - In some cases, calls are suspended or are timed out during the search. This results in missing responses that, technically, do not have a status code. In these cases, EvoMASTER wraps the calls in try-catch blocks, in order to ensure that the generated test cases are valid and pass. While these are not

technically faults, such calls do fall under the category of **Inconsistent Behavior** and may be worth investigating further.

Note that it is not entirely clear if the faults/unexpected behavior in this category are due to the SUT itself, configuration issues around handling the calls, or other reasons, like handling of resources in the OS. Notionally, there should be a consistent behavior between different runs of the same test case, both between the search and subsequent executions, and between different subsequent executions. The difficulty in replicating and diagnosing these faults makes it hard to provide a more detailed description or to classify them in a more detailed manner.

EvoMASTER does a reset of the SUT state between executions of the test cases, based on code provided by the users. To simplify the writing of these functions, EvoMASTER provides as a library different utilities, e.g., to clean the state of the databases, if any. As a result, **Multiple Execution** and **Fail by Succeeding** faults can be the result of faults in EvoMASTER itself or in the reset functions of the driver associated with the SUT. Issues such as incomplete database clearing, reliance on external services for initialization of the internal state, or improper handling of necessary TCP ports can all lead to problems of this type. Given the importance of the driver linking EvoMASTER to the SUT, and the importance of the reset function of that driver, correct identification and classification of this type of fault seems a relevant part of analyzing faults revealed by automated methods.

5.2 Schema Conflicts

Schema Conflicts are types of faults that consist of mismatches between the validity of certain inputs, as defined by the OpenAPI schema, and the validity of the same inputs, as handled by the SUT.

In Figure 1, these faults are further divided into further sub-categories. First, faults related to form. This includes mismatches between the formats of particular data as accepted by the OpenAPI and the acceptable formats for the same data in the SUT. These faults could be fixed by updating the OpenAPI to better match the data formats that are acceptable to the SUT, or to provide the SUT with additional checks and exception handling.

The second sub-category deals with custom constraints, i.e., constraints that are specific to the SUT and have not been captured in the OpenAPI. This could be because such constraints cannot be automatically derived by tools that generate OpenAPI schemas from the code, or have not been expressed manually. The exact tradeoff between the benefits of improved OpenAPI schemas against the costs of manually updating such documents is not the focus of this work, but situations where a less comprehensive automated OpenAPI schema is preferred appear to be quite common. This may mean that, rather than fixing the OpenAPI specification to better match the semantics of the system, it may be more expedient for developers to include additional checks and exception handling in the SUT itself.

- **Format.** This is a type of problem that is caused by a mismatch between the definitions of acceptable formats in the OpenAPI specification and the definition of the same formats in the SUT. For example, the industrial case study (*Ind0*) defines a date as having a maximum of 2 digits for month and day. Technically, this allows one digit months and days. However, the parser used by the SUT rejects single-digit values for month and day, leading to an exception being thrown and a crash. The constraints are expressed as a regular expression, with the goal of returning a response in the 4xx family when the input does not match the format of a correct date. However, the input regular expression is underspecified, and allows some inputs that do not describe correctly formatted or valid dates to be accepted by the framework (e.g., although being just 2 digits, the value 42 does not represent a valid month number). Once such inputs are accepted by the regex, they are handled by the business logic, that try to parse the input into a date object. Since the business logic relies on the previous checks for validity, no additional checks are performed,

Fault Example

```
ValidatableResponse res_6 = given().accept("*/*")
    .header("Authorization", "Basic_am9hcXVpbToxMjM0") // manager
    .contentType("application/json")
    .body("{\"tBfdp8M\"}")
    .put(baseUrlOfSut + "/printshops/15/pricetable/editstapling")
    .then()
    .statusCode(500) // io.github.proxyprint/kitchen/controllers/printshops/
        PriceTableController_255_editStaplingPrice
    .assertThat()
    .contentType("application/json")
    .body("status", numberMatches(500.0))
    .body("error", containsString("Internal_Server_Error"))
    .body("exception", containsString("java.lang.NumberFormatException"))
    .body("message", containsString("For_input_string:_{\"tBfdp8M\"}"))
    .body("path", containsString("/printshops/15/pricetable/editstapling"));
```

Fig. 3. An example of a **Type** fault. The schema requires that a JSON object be used to provide the input, as the body payload. The OpenAPI schema specification places no additional constraints on the content of that JSON object, and thus the default value is that the object should contain a string. The SUT, however, expects a numerical value and does not safely handle invalid inputs, resulting in a `NumberFormatException` and a response with a 500 code.

and the attempted parse fails by throwing an exception, which leads the Spring framework to generate a response with a 500 code.

Note, the **Format** category does not include inputs that are generated as strings and later parsed into inputs (those fall under **Mismatched Type**, see below). Note that this differs from the **Semantic** category in that the problem is not that some aspect of the meaning of the given data is not captured, but rather the format is incompletely specified. The SUT expects the incoming data to be valid, for example by relying on Spring annotations, and conducts no further validation of said data. However, if some invalid data does get passed, either due to existing validation being incomplete or faulty, the SUT does not handle the exceptions that are raised and this results in a crash.

- **Type**. In most cases, the OpenAPI describes the expected type of input. However, there are situations when the type of input is not clearly specified. This can lead to clear type exceptions being thrown. For example, the call in Figure 3 requires that a JSON object be sent as body payload. This object should contain a numerical value, but the OpenAPI specification is just that of a string. When a randomly generated string is provided as an input, this leads to an exception. Note that this differs from the **Semantic** category: the **Semantic** category contains cases where additional meaning (that cannot be captured by the API schema) is missing. The exact definition of what dates are valid, for example, may be too cumbersome to include in the schema. This category contains cases where the mismatched type could be present in the schema without additional generation or maintenance effort.
- **Missing or Null Values**. This category describes situations where the SUT business logic expects some input or parameter to be present, but they are not mandatory in the schema. This could result in EvoMASTER generating a set of parameter values that are valid according to the OpenAPI schema, but not according to the SUT implementation. The validity of the data seems often assumed by the underlying SUT, and so the business logic would try to use particular parameters without additional checks, thus leading to exceptions being thrown.

Note, this refers only to inputs that are generated by EvoMASTER and passed to the SUT as inputs in the HTTP calls. A fault category that has similar symptoms, but a different root cause, is discussed below, under **DB Injection**.

- **Negative.** This is a particular type of custom constraint fault where the input value is expected to have a particular mathematical property, with non-negative numbers being the most obvious type. The most common example we have encountered is that of an input being defined as an offset for data display. The offset is meant to be a non-negative number, but no such restriction exists in the OpenAPI schema. Thus, when a value that does not meet the condition is passed, it is valid according to the schema, but invalid according to the SUT. This leads to an exception being thrown and then to a crash. Note: this is a particular type of custom constraint fault.
- **Pre-Approved List.** In several cases, an API schema defines a particular input only as a string, and EvoMASTER generates input values accordingly. However, the underlying logic of the SUT expects that particular input to be one of a pre-defined list of acceptable values. Since the evolved input string is highly unlikely to be present on the list of acceptable inputs, this often leads to an exception being thrown and then to a crash (as the SUT is expecting only valid inputs). Examples of this can be found in *rest-news* (an input is expected to be one of 247 valid names for a country), and *proxyprint* (inputs are expected to be one of a small list of acceptable Colors, Ring Types, or Cover Types). Note, this may be a limitation of the current definitions of API schemas. It is worth noting, however, that this kind of fault does significantly reduce the number of valid inputs that EvoMASTER produces and leads to resources (fitness evaluations) being spent on faulty inputs, without providing a clear gradient for improvement. While this may be more of a limitation of EvoMASTER, it does provide a hurdle to be overcome in terms of automated test data generation. It is interesting to note, however, that these limitations are not represented in the OpenAPI schema in any way, as all the checks are conducted in the SUT back end.
OpenAPI does allow users to define enum constraints, to define a list of acceptable values for a particular input. However, in the case studies discussed here, the specification did not include such constraints. Values on pre-approved lists can include a diverse set of items, from supported formats, acceptable parameters and settings, to valid data such as country names.
Even if those constraints were present in the schema, EvoMASTER would still send invalid values on purpose (for robustness testing), albeit with low probability. This is to check that the SUT would respond correctly, i.e., with a status code in the 4xx group.
- **Invalid Combination.** This category describes inputs that are expected to be in particular combinations (e.g., constraints involving the relations between more than one parameter). Currently, OpenAPI schemas do not support multi-parameter constraints. However, extensions have been proposed to handle such issues [22], with black-box testing tool like RESTTest [23] (from the same authors) being able to generate tests with valid combinations.
scout-api showcases several such examples. A tag can be denoted by id, or by a name and a group, but these constraints are not present in the OpenAPI schema. Hence, while the generated values are not themselves invalid, their combination could be. Another example, also from *scout-api* is that of the “myFavourites” input value. The option to select favorites is not currently supported. Setting that input to any value (including “false”) causes the SUT to throw an exception and crash. The SUT expects certain constraints on the acceptable combinations of input values that are not reflected in the OpenAPI schema resulting in unwanted behaviors (in this case, crashes with code 500).
- **Semantic.** This type of fault is usually due to a particular type of semantic meaning that some input data has (or should have) that is not captured by the OpenAPI specification.
An example of this is invalid (but not malformed) dates. The industrial case study (*Ind0*) contains some checks to ensure that dates being passed are valid. For example, maximum of 2 digits for month and day, 4 digits for year. A date of the form “8040-13-35” is, according to the regular expression that checks the inputs, valid. However, when the SUT tries to parse that date, an exception is thrown, leading to a crash. In this case, the constraints defined in the input validation do not capture the semantics of the input, and

thus allow invalid inputs to be generated. Since exactly specifying the meaning of particular inputs may require extensive time and effort, this problem could be fixed by adding additional checks to ensure that the semantics of the received data are correct, including exception handling in the case that semantically incorrect data is being processed by the SUT.

- **Unsupported Code.** The last, and arguably most numerous, type of fault is related to unsupported HTTP response codes. The OpenAPI schema allows each call to define supported codes, and their meaning. Some responses may be unsupported, though returned by the SUT. Note that the oracle for what constitutes “supported responses” is derived from the OpenAPI schema. If the schema is not maintained, incomplete, obsolete, or reliant on automatically generated defaults, rather than user defined behavior, even successful calls may be technically unsupported and therefore be classed as faults according to this category. Technically, all calls with response code in the 5xx group represent server issues (but not necessarily related to software failures in the SUT). As such, it is not unreasonable that developers would not put such response codes in the expected behavior of their endpoints. Therefore, in this fault category we do not consider those status codes in the 5xx group (as anyway the ones with 500 are already considered as faults).

In general, schemas for RESTful APIs are a good starting point for understanding what operations are possible, what inputs are expected, and what results can be provided. However, there seems to be a certain amount of uneven handling of these schemas. Manually generated schemas are difficult and costly to maintain, and can quickly become obsolete and unsupported. Automatically derived schemas have the drawback of not always being able to capture the meaning of the code they are being generated from. It is possible to refine automatically generated schemas, using annotations to guide the automated tools to allow for refinements, especially where the additional time and effort is outweighed by the importance of the endpoint or input in question. For testing purposes, the more accurate and up to date a schema is, the more useful it is in supporting testing activities, but this has to be weighed against the cost and effort of maintaining such schemas.

5.3 Data related faults - initialization, data consistency, database injection faults

This category consists of a more generic type of data related faults. These can be issues with the initialization, handling, or seeding of test data. Data can be seeded for test purposes either by using endpoints designed for the purpose or by direct injection into the database [8]. Direct injection might lead to inconsistent data, missing connections, or simply null values. These are usually covered under the category **Invalid Combination**. The data related category focuses more on direct faults when interacting with the database. These faults can be:

- **DB Operations.** These are faults that are directly linked to database operations. One example is the **DB.OptimisticLock**, identified as a result of a `OptimisticLockException`, category: during the search, parallel access to the database by several calls can lead to inconsistency in the data (e.g., when a previous HTTP call timeouts, and the following HTTP calls are executed while previous request-handling threads are still running). **DB.Key** faults, identified by the `JdbcSQLException` with a “Unique index or primary key” message, have been encountered when trying to seed a database for the second time, or while the database is not empty.

The category **Constraint Violation Exception**, manifesting as a `ConstraintViolationException`, is a more general category of faults. These occur when checks on the database result in exceptions being thrown, but those exceptions are not handled by the SUT, resulting in failures and in responses with the 500 code. These faults can be the result of a number of different causes, ranging from issues with the test scaffolding (e.g., incomplete resetting of the database state), to SUT specific issues (e.g., assumptions being made about the validity of inputs, improper handling of potential errors and violations, mismatches between the constraints expected by the SUT and the constraints defined for the database).

The case study *proxyprint* shows an interesting case of **DB Operations** faults. The *proxyprint* API exposes an endpoint with no inputs, called “/seed”, the purpose of which is to seed the database with some valid data for the purposes of testing. However, there is no constraint to stop an automated tool from calling that endpoint again during the search. As a result, trying to seed the database a second time with the same data results in failure.

The white-box approach employed by EvoMASTER allows for a better understanding of the internal state of the SUT and for the development of additional tools to support in identifying and handling such test cases. Tracking connections and transactions, aborting those transactions that are hanging or incomplete, and maintaining a clear record of the state of the database could ensure that the SUT reset is done correctly and conditions between test case runs are consistent.

- **DB Injection.** These are faults traceable to the incorrect injection of data into the database, and can be traced to insufficient information available to EvoMASTER. When EvoMASTER injects data into the database, the SUT-specific checks and handling are skipped. This means that injected objects might be inconsistent: references to relevant objects or values for particular fields (that normally would be either checked or filled in by the SUT) may be null or invalid. Thus, when trying to use those objects, exceptions are thrown and the SUT crashes. Though not technically SUT errors, it is worth noting that such situations do arise. Note also that these errors sometimes resemble other errors, in terms of behavior (required values are null resulting in Null Pointer or Illegal Argument Exceptions). The main difference is that, in this case, inconsistent data is injected directly into the DB.

This type of fault could be encountered by any type of automated tool that directly interacts with the database, injecting data without using the mechanisms provided by the SUT. This approach is usually necessary when the SUT provides no other means of generating that data. This could include information that is collected by the SUT (e.g., *catwatch* does a lookup for the ids of GitHub projects) or is provided by other services (e.g., *proxyprint* provides no valid way for creating a printshop object, outside of a few test examples). Note that some databases do provide a way to define constraints on the data (e.g., using CHECK in some SQL databases), and EvoMASTER can generate data that satisfy such constraints. However, the case studies included in our work specify most of the constraints at the SUT level, rather than directly in the database. While suitable means to provide test data to a system should be encouraged, DB injections will continue to be useful for achieving better code coverage for SUTs where such means are not in place. Thus, the identification of **DB Injection** faults will continue to be relevant. However, most of these kinds of faults would result in “false positives”, as not really due to faults in the SUT that its developers need to fix.

- **Seeding.** The **Seeding** subcategory emerged from a fault encountered with the *proxyprint* case study. If a the developers of the SUT already having some test data that easily reusable, EvoMASTER can use such data during the search. When starting a test case in *proxyprint*, the database is seeded with a collection of test users via an admin endpoint provided by the SUT for this purpose. This ensures that the users in question are consistent, have the appropriate credentials and permissions, and pass the required authorization checks. However, the SUT is reset between test case executions and the databases are cleared of information. These test users, however, do not appear to be available for any purpose other than authentication into the system. Calls to endpoints that look those users up in the database, or validate some information regarding those users, fail to find the required database entries and a `NullPointerException` is raised. Since the seeding process is repeated during the SUT state reset process, the cause of the fault appears to be that the seeded data is incomplete.

Long term, this could be solved by the SUT providing specific endpoints to allow data, especially test data, to be added to the databases and to allow specific states to be set. However, such mechanisms cannot be assumed to be in place, and automated tools need to rely on other tools to initialize SUTs with the required data and to reset SUTs to the initial state between test case executions.

In addition to clearing data from databases and resetting the internal state of the SUT, automated systems also need to ensure that database transactions that are incomplete between test case executions are aborted, to ensure a consistent behavior for the different test cases (aborting all hanging transactions, and release all acquired database locks, is working in progress that is still not fully supported in EvoMASTER). This category does indicate improvements that can be made to the testability and maintainability of the SUT. Since faults in this category are reported by EvoMASTER, and, until such time as these types of test case problems are addressed, the existence of this category would be of interest to developers and users of EvoMASTER. Note that some of these faults might be related to actual issues in the SUT. For example, if due to software faults some transactions are left hanging after a HTTP call is completed, several of such calls could lead the SUT to run out of memory or out of available threads to handle such transactions.

Data consistency faults are usually found at the interaction between the SUT and the database system. When a means of seeding the database with valid data is not provided by the SUT, direct injection of data into the database allows the search to continue. If the database injection causes the data to be inconsistent, this usually results in one or more fields to be missing. This can be seen in the category **Invalid Combination**. If the database interaction, either when injecting data, when using a provided seeding endpoint, or during regular operation, results in the database itself throwing an exception or causing an exception to be thrown, then those are considered data related faults.

5.4 Faults in the Business Logic

This category of faults captures general unwanted behaviors of the SUT that can be observed as a result of the interaction with EvoMASTER. These behaviors are often crashes resulting from more generic software faults, rather than faults typical to RESTful APIs. This category has the potential to be the most diverse, and arguably interesting, of the categories of faults. It comprises faults that are present in the business logic of the SUT itself, and are unlikely to be connected to other subsystems or services.

While all faults can have an impact on the correct behavior of the SUT, these faults are most likely to be problems with the business logic and to vary between different systems, requirements sets, and domains. They are also faults that are most likely to require system and domain knowledge for a correct diagnosis.

Note that this category describes faults that result in crashing behavior and originate in the business logic code. Faults that are the result of misinterpretation of requirements, domain specific misinterpretations, or fault that relate to the degree of compliance between business logic code and desired behavior are difficult to ascertain from automatically generated test cases.

- **Hardcoded.** One type of fault found is that of a value being invalid due to hardcoding. The case study *catwatch*, for example, selects a sublist of 10 valid results of particular actions to display. Normally, these results rely on external scripts that provide data for the SUT. However, under testing circumstances, this external data might not be available. This leads the list of valid results to be less than 10 elements. In this case, the hardcoded value of 10 for the sublist (whilst not technically invalid) is lower than the size of the full list. This leads to an exception and a crash. Note that this could also be interpreted as an environment problem: as the testing environment does not fully capture the expected interactions of the SUT under normal conditions. In general, however, hardcoded values are considered to be bad practice, especially when they can lead to unexpected behaviors.
- **Not Found.** This category describes faults caused by the improper handling of missing resources. A fault in this category of failures is the result of an operation being conducted on resources that do not exist: either the search for a particular resources comes up empty, or the resource of the specified name or identifier, does not exist. An example of correct behavior, in this case, would be for the SUT to return a response with a code 404, indicating that it encountered a user fault, and in particular that that fault is the user's attempt

to access a resource that cannot be found. If instead of handling the problem, the SUT throws an exception (because its code assumes such resource to exist), this can lead to a crash and a code 500 response. The reasons for the particular resource not being found can be varied, but the SUT should not crash because a non-existing resource was requested.

- **Unsafe Cast.** During the search, a `ClassCastException` was encountered as a result of an unsafe cast (*proxyprint*). The example can be seen in Figure 4. The examples found do not appear to be related to the inputs being generated by EvoMASTER, but rather unsafe casts in the internal workings of the SUT. In the example cited, a request is cast into a `MultipartHttpServletRequest` in an unsafe manner (without any checks that the request object is, indeed, of that type or that the cast can be performed). When the object is incompatible, this results in a crash.

5.5 Taxonomy Discussion

Note that two categories of faults are mentioned that can be thought of as being outside the regular RESTful API:

- **Framework Misconfiguration** issues are not faults in the business logic of the SUT, but rather in the configuration of services that provide users and test engineers with access to the SUT. As such, it is important that faults of this nature are identified and properly fixed by test engineers to ensure successful testing and deployment of the system. Moreover, misconfigurations may lead to security breaches or other unwanted behavior.

Other frameworks or libraries that are used by the SUT may also contain faults. While the development team for a particular SUT may not be in a position to address those faults, automated test tools still provide benefits in identifying these faults, and providing test cases that replicate these faults. Even if such faults are rooted in the interactions between the SUT and other frameworks (rather than being present in the business logic of the SUT itself), they are potentially of interest to test engineers.

- **EvoMASTER related faults.** While every effort is made to ensure the quality of the tools being used, some faults may be false positives, due to faults in the data generation or in the testing scaffolding. As an example, the current implementation of EvoMASTER's process for seeding the database bypasses the code in the SUT. In some cases, objects being added to the DB are checked for consistency or have additional information added alongside. When these checks are bypassed, the database may be left in an inconsistent state. While this example relates specifically to EvoMASTER, it is useful to identify false positives when possible. We argue that faults in the scaffolding around the SUT, being in the frameworks used for deployment, other services being used, or the testing infrastructure should still be included in the classification and reported. This would enable test engineers to get a better understanding of the limitations of EvoMASTER (as well as other automated testing tools). Moreover, when prioritizing and deciding on the resources and fixes for particular faults, it would be useful to identify correctly and accurately when the faults are false positives, i.e., the result of the testing infrastructure being used.

The taxonomy presented in Figure 1 shows the hierarchy of the faults that have been found in the case study SUTs by automated means, in this case by EvoMASTER. Test cases generated by EvoMASTER reveal a diverse set of fault categories, from configuration and framework issues to faults in the business logic of the SUTs in question. Since the taxonomy is based on real faults revealed by test cases generated by EvoMASTER, it provides an overview of the faults that we actually found in the case study RESTful systems by automated means.

The taxonomy presented in Figure 1 shows the wide range of fault categories that have been found in the benchmark case studies, using EvoMASTER.

5.6 Classification process

Part of taxonomy definition (see Usman et al. [28] for the methodology and Ralph [25] for uses and philosophy) is providing a methodology for classifying new instances according to the current taxonomy. This methodology will be described in this section. For this study, the classification method was applied manually by one of the authors, and reviewed by a second.

When analyzing a new potential fault, the starting point is the test case that EvoMASTER has generated. Test cases generated by EvoMASTER are *self-contained* (i.e., they contain all the necessary SUT state setup, database injections, and any other operations needed to execute them). Furthermore, the generated tests are *independent* (i.e., they can be run without affecting or depending on each other, either independently or as part of a test suite, and in any order, without any impact on their outcome[32]), as they are run individually during the evolution process and the state of the SUT is reset between test case executions.

Since test cases are independent, each new test case can be run in isolation and analyzed as a distinct entity. Each test case consists of an optional database injection section and a sequence of HTTP calls, along with the checks and assertions made on the response to each call. Potential faults are those calls that have a response with status code 500 or with status codes that are not supported by the OpenAPI schema.

A call that has a response in the 4xx code family could be deemed a potential fault if the endpoint it accesses does not explicitly support a response with that code. Since they do not result in crashes, they are identified as expectation faults, classified as such, but no additional information about them is provided. No exceptions are thrown for these faults, and therefore no stack traces or last executed line information is generated. While these are faults, indicating disagreements between the OpenAPI schema and the observed behavior of the SUT, they could be considered low priority (e.g., faults related to incomplete documentation). Table 4, column **Total Expectation Faults** shows the total number of expectation faults that are found during the ten experimental runs, across the case studies included in the benchmark. Note that, while technically a response with a 500 code is also not supported, we exclude those faults from the **Unsupported Code** category.

Most of the analysis focused on faults that cause crashes, i.e., responses with the 500 code, with a total of 415 such faults being manually analyzed. As a result, the classification process is described considering examples of faults that cause crashes.

Note that the examples discussed for the classification process are included in the 415 that form the basis for the taxonomy, and they have been included in the analysis already. They are presented here solely as examples of the classification process, not as additions to the taxonomy itself.

Each call that is identified as a potential fault is analyzed on its own, and several potential faults with different root causes and leading to different failures may exist in the same test case. Thus, for each potential fault:

- (1) determine if it is a 500-code or expectation fault. A 500-code fault would be accompanied by a last executed line from the SUT (recorded with EvoMASTER's instrumentation). The last executed line provides a good starting point for diagnosing and classifying the fault. Expectation faults that do not result in a 500 code can, usually, be classified according to the expectation that has been broken. For faults related to the 500 status code response, the classification process will continue with a manual analysis of the root causes of the fault.
- (2) where available, the error messages accompanying the 500-code fault also provide useful information in terms of precisely what kind of anomalous behavior or fault to look for. Where the SUT is configured to allow it, the test cases that cause 500-code faults contain checks on the error type, the exception being thrown, and the error text message (in addition to the code). These provide useful insight into the fault category. When such information is not available to the test case, investigating the last executed line using the debugger and collecting the internal stack traces for any exceptions being thrown provides the actual exception and message.

Fault Example

```

ValidatableResponse res_6 = given().accept("*/.*")
    .header("Authorization", "Basic_bWFmYWxkYToxMjM0") // employee
    .post(baseUrlOfSut + "/printdocument")
    .then()
    .statusCode(500) // io/github/proxyprint/kitchen/controllers/consumer/
    PrintRequestController_107_processSumitedFiles
    .assertThat()
    .contentType("application/json")
    .body("status", numberMatches(500.0))
    .body("error", containsString("Internal_Server_Error"))
    .body("exception", containsString("java.lang.ClassCastException"))
    .body("message", containsString("org.springframework.security.web.servletapi.
        HttpServlet3RequestFactory$Servlet3SecurityContextHolderAwareRequestWrapper_cannot_be_cast_to_org.
        springframework.web.multipart.MultipartHttpServletRequest"))
    .body("path", containsString("/printdocument"));

```

Fig. 4. An example of an HTTP call in a test case generated by EvoMASTER. Checks on the internal values of the response are conducted to ensure consistency across several runs. In addition to the status code, information about the call, and the contents of the response, the test case also includes (in a comment next to the status line) the last executed line of the SUT.

- (3) once a fault has been identified, the process of determining its origin can begin. Describing the fault (what goes wrong, what types of exceptions are thrown, and where do the faulty values originate) can help practitioners in determining the main category of fault: **Schema Conflicts** are usually linked to faulty inputs, either provided to individual endpoints or injected into the database.
- (4) once a clear description of the fault has been provided, the analyst can begin identifying what error classes apply and if there are any particular types of invalid input that apply.
- (5) for the purpose of analysis and reporting, a tentative class is selected. This is based on the analyst's evaluation of which of the error classes is the root cause of the fault, and it is also where the individual fault is classified in this taxonomy. The classification starts from the most general applicable class, and continues into the sub-classes until either the relevant fault category is found, or the analyst determines that not of the existing categories are relevant.
- (6) if no relevant category exists, a new category is created. This will be as a sub-category of the most specific applicable category reached at the previous step. The initial description of the new category is derived from the description of the fault to be assigned to it, but will be updated as more information and more examples of such faults become available. The goal is to ensure that the new category is clearly defined, distinct from other categories, and that diagnostic traits for classifying relevant faults in this category and unambiguous.
- (7) while not part of the classification process per se, this at this stage, the fault, along with all the additional information, and the tentative classification, can form the basis for decisions on how to follow up the discovery of the fault: whether the fault should be prioritized for further investigation and/or fixing, how the task will be assigned (if several roles/areas of responsibility are involved), and where further investigations may begin.

By the end of this process, the information needed to describe the new fault according to each of the facets should be available, and a decision can be reached regarding which existing class the fault can be classified into or if a new class of fault needs to be defined to better describe this new fault.

The code example in Figure 4 shows an HTTP call, part of a test case generated by EvoMASTER, that is identified as a potential fault. First, the call has a response with a 500 response code, identifying it as a potential fault. In addition to the code, the call has the last executed line (seen in comment, after the status code assertion), as well as checks on the contents of the call. The exception and the message give some information about what when

wrong and the last executed line points to the last line in the SUT that was successfully executed before the exception was encountered. This allows a test engineer to begin their analysis of the root cause of the fault and to provide their interpretation. With all this information available, the test engineer can decide on a tentative class for the fault, or decide that a new class of fault might be needed. In this case, the root cause of the failure is a unsafe cast between two types of Servlet in the SUT. As a result, the problem is in the SUT itself, and the fault will be classified as a **Fault in the Business Logic**, in particular an **Unsafe Cast**.

Note that some of the facets defined for the taxonomy, as well as the classification process itself, rely heavily on a qualitative analysis of the faults in question. In particular, we focus on determining the root cause of the faults, and providing an interpretation of the fault and the failure it causes. Currently, this qualitative analysis is performed manually and is a time and effort intensive process.

The automatic generation of software tests, however, can produce large test suites containing a large number of potential faults for classification. The prevalence of low priority **Unsupported Code** faults indicates the need for some automated support, to enable test engineers and developers to classify the faults found and focus their efforts on the relevant categories. While conducting the root cause analysis and providing an interpretation for the individual faults is difficult to accomplish automatically, EvoMASTER can provide a large amount of information that help the analysis. This may be unavailable in black-box settings, or when testing external services without access to the codebase.

The follow-up hypothesis is that some of the information that is available, such as the last executed line (i.e., the location in the SUT source code where a fault is traced to) and the error messages could provide support in conducting an initial classification of the faults. With this preliminary classification available, test engineers could focus their time and resources on those faults that are more likely to be of high priority.

Table 5 shows the prevalence of the various categories of faults in the case study SUTs. The data in this table described the 415 found by the test cases generated by EvoMASTER, manually classified and used to build the taxonomy presented in Figure 1.

5.7 Expanding the taxonomy

A concrete process for expanding the taxonomy is difficult to provide, as individual fault classes would have to be added depending of the faults found in particular systems. Entirely new classes may have to be added to describe previously unseen faults, or classes may need to be sub-divided, to account for similar faults that nevertheless need to be differentiated. However, some guidelines can be provided.

First, the process of developing the taxonomy started from concrete faults that were found in the case studies using EvoMASTER. Thus, we recommend that efforts to expand the current taxonomy also rely on concrete code examples that illustrate the new fault classes. These could be automated or manual, but we suggest that the standard approach proposed by EvoMASTER be followed: independently runnable test cases that consist of clearly differentiated HTTP calls. As a consequence, the first step would be to conduct an analysis of the fault in question, according to the process described in Section 5.6.

Once the analysis is complete, and the root cause of the fault has been described, a decision regarding the classification should be made. We recommend starting from the most general applicable class. In the example shown in Figure 4, the root cause of the fault is in the business logic of the SUT, rather than in any of the other high-level categories. Thus, a new high level class is not required, as the **Faults in the Business Logic** class describes the fault accurately enough.

As with the regular classification, the analysis moves deeper into the taxonomy to determine which, if any, of the existing classes applies. If the fault does not appear to fit in any of the existing sub categories, the next step would be to reassess the current category (i.e., whether it is possible that a sister category would be a better fit

SUT	Class	Count
scout-api	NotFound	107
proxyprint	Specification.Custom.MissingOrNull	28
proxyprint	NotFound	28
scout-api	Specification.Custom.Semantic	24
ind0	Specification.Custom.Semantic	23
features-service	NotFound	20
scout-api	DB.OptimisticLock	16
scout-api	Constraint.Null	15
scout-api	Specification.Custom.MissingOrNull	14
ind0	Framework.NotAccepted	13
ind0	Invalid.List	12
proxyprint	Specification.Custom.PreApprovedList	11
proxyprint	Seeding	10
scout-api	DB.ConstraintViolation	9
scout-api	Specification.Custom.InvalidCombination	9
catwatch	Hardcoded	7
features-service	Specification.Form.Type	7
proxyprint	Unsafe.Cast	7
rest-news	Specification.Custom.PreApprovedList	6
catwatch	Specification.Form.MissingOrNull	6
catwatch	Specification.Custom.PreApprovedList	6
proxyprint	Inconsistent.Unexpected500	6
catwatch	Specification.Custom.Negative	5
proxyprint	DB.Key	5
proxyprint	Fail.by.Succeeding	5
scout-api	Inconsistent.Unexpected500	3
catwatch	Specification.Custom.Semantic	2
proxyprint	Framework.NotAccepted	2
proxyprint	MultipleExecution	2
proxyprint	Specification.Form.Type	2
catwatch	Specification.Form.Format	1
scout-api	Inconsistent.Upstream	1
features-service	Specification.Form.Format	1
proxyprint	DB.Injection.Inconsistency	1
ind0	DB.Key	1
Total		415

Table 5. The numbers of faults present in each SUT in each class. Note that *rest-ncs* and *rest-scs* have no faults with code 500, and are, therefore, not shown in this table.

or provide more similar sub-classes). Note that it is not uncommon for some faults to have more complex root causes and to have a more ambiguous or undefined sub-category or specific class.

If none of the current classes describes the fault and its root cause accurately, a detailed definition for the new category of fault will be developed. This new definition would start from the current example, focusing on how the new category of fault differs from existing categories. This differentiation should be especially focused on other sub-categories that are similar in terms of root cause, fault expression, exceptions thrown, etc. This is to ensure that the new category of faults can be suitably differentiated from existing ones and making it easier for faults to be classified between the categories.

Continuing from the example in Figure 4 and its classification. In the previous step, we have determined that the call in that example is a **Fault in the Business Logic**: an unsafe cast in the code of the SUT results in an exception being thrown and thus in a crash. For the sake of this example, we will assume that the **Unsafe Cast** category did not already exist.

The root cause of the fault is that, in the business logic of the SUT, an `HttpServletRequest` is cast to a `MultipartHttpServletRequest` without any additional checks. If the input is not actually of that class, or is not compatible with that class, a fault is thrown and the call fails. Thus, the fault belongs in the **Faults in the Business Logic** higher class: it is clearly a fault in the SUT itself and how the SUT handles its business logic.

The other sub-classes in the **Faults in the Business Logic** class do not appear to describe this fault well, so a new class is needed. The root cause is an unsafe cast leading to a `ClassCastException`, so we have chosen

Class	rest-news	catwatch	scout-api	features-service	proxyprint	ind0	Total SUTs	Total Faults
NotFound			✓	✓	✓		3	155
Specification.Custom.Semantic		✓	✓			✓	3	49
Specification.Custom.PreApprovedList	✓	✓			✓		3	23
Specification.Custom.MissingOrNull			✓		✓		2	42
Framework.NotAccepted					✓	✓	2	15
Inconsistent.Unexpected500			✓		✓		2	9
Specification.Form.Type				✓	✓		2	9
DB.Key					✓	✓	2	6
Specification.Form.Format		✓		✓			2	2
DB.OptimisticLock			✓				1	16
Constraint.Null			✓				1	15
Invalid.List						✓	1	12
Seeding					✓		1	10
DB.ConstraintViolation			✓				1	9
Specification.Custom.InvalidCombination			✓				1	9
Hardcoded		✓					1	7
Unsafe.Cast					✓		1	7
Specification.Form.MissingOrNull		✓					1	6
Fail.by.Succeeding					✓		1	5
Specification.Custom.Negative		✓					1	5
MultipleExecution					✓		1	2
DB.Injection.Inconsistency					✓		1	1
Inconsistent.Upstream			✓				1	1
Total								415

Table 6. Overview of the distribution of classes in the case study SUTs. The table shows the presence of individual types of faults in the case study SUTs. The last column shows the number of SUTs that each fault category is present in. Note that *rest-ncs* and *rest-scs* have no calls with response code 500, and are, therefore, omitted from this table.

to call the new class **Unsafe Cast**. This new sub-class is added as a daughter class to the deepest node in the taxonomy that describes the fault, in this case as a daughter to the current node: **Faults in the Business Logic**. The description of the class will contain the interpretation of the root cause of the current fault.

Note that the goal of the taxonomy is to provide a clear understanding of the categories of faults that are found in practice, using automated testing. As such, the taxonomy is meant to be flexible and robust enough to allow for new categories and subcategories of faults, to accommodate new findings and new systems. It is expected that categories and their descriptions are updated and new information added when relevant.

5.8 Fault Category Commonality

Table 6 shows the presence of the found classes of faults in the case study SUTs, as identified by the test cases evolved by EvoMASTER. The rightmost column displays the number of SUTs each fault category can be found in.

The most prevalent type of fault is the **Not Found** category. When a request is made on a resource that does not exist, correct behavior would be to return a response with a code 404. In some cases, however, the SUT seems to have no code handling situations where a request is made on resources that do not exist, and no mechanism for recovery. Thus, the SUT crashes, usually with a `NullPointerException`.

Specification faults are also quite common. They describe a mismatch between the behavior of the SUTs and the OpenAPI specifications, usually due to assumptions being made about the validity of input data. The most prevalent examples are the requirement that specific inputs be part of a pre-approved list, but this restriction not being present in the OpenAPI schema. The *rest-news* case study requires that a particular input be a part of a pre-approved list of 247 countries, and it assumes that the input is correct (i.e., the fault here is that the SUT does not have the business logic aimed at returning a 4xx-code response when such input is invalid). However, that input is identified in the OpenAPI schema as a string, with no additional restrictions being defined. When input values are generated automatically, it is quite unlikely that a valid string (i.e., a string that exists in the pre-approved list) is being generated, and so it is rather trivial to detect this fault.

Note: even if such constraint was present in the OpenAPI (e.g., as an enum), still an automated generation tool could produce invalid inputs on purpose, to see if the SUT behaves correctly (i.e., returning a response with a 4xx-code). This is done for example in EvoMASTER.

Some categories of faults are only present in one case study. This includes all of the database related faults, and most of the **Configuration and Execution Faults**.

In summary, the most common type of fault appears to be the **Not Found** category: a mishandling of requests on resources that do not exist. **Specification Faults** are also frequently encountered, with discrepancies between the SUTs and the OpenAPI schemas describing them being present in all but two systems (*rest-ncs* and *rest-scs* – simple, artificial examples).

Faults in framework configuration and database handling are encountered, but are usually different sub-categories between the different SUTs. The same is true for faults in the business logic. This is expected, as there is considerable variation between the SUTs in terms of the frameworks, databases, and code standards used.

We can conclude that mishandling resources that are not found and mismatches between OpenAPI specifications and the RESTful APIs they describe are a common type of fault that developers can expect to encounter, in a variety of forms.

6 CLUSTERING - RESULTS AND INTERPRETATION

So far, we have described a taxonomy and a way to classify the potential faults in automatically generated test cases according to that taxonomy. However, the classification process is heavily reliant on a root cause analysis of individual potential faults that involves a large amount of manual effort. This is all the more problematic, as EvoMASTER uses code coverage as a fitness function, thus leading to large test suites that maximize coverage. For example, EvoMASTER generated over 200 test cases for the *proxyprint* and *scout-api* case studies (see Table 3). Moreover, as automated testing tools improve and greater coverage is achieved, this would result in even more substantial test suites.

One of the arguments in favor of developing and using such a taxonomy is the cognitive efficiency that results from having a clear and systematic way of describing fault categories. Ideally, test engineers and developers would use such a taxonomy to focus their resources on addressing faults from more serious and relevant categories. An initial, albeit tentative, classification would greatly help in concentrating even the resources required to do a root cause analysis on only those faults that have a potential to be interesting or serious.

The result of this approach is that we could use automated techniques, e.g., clustering, to provide developers with a tentative classification. In a practical sense, clustering is meant as a tool to allow test engineers and other users of EvoMASTER to better understand, analyze, and use the test cases and test suites generated as a result of the search. The clustering, however, is an attempt to conduct an initial classification, to reduce the amount of time, effort, and resources that would be required to accomplish the same task manually. In effect, the clustering is an attempt to get some of the benefits of a taxonomy without the cost of conducting manual classification. A common terminology to allow test engineers to communicate with each other and other departments, to improve

understanding of the test suites that are generated by EvoMASTER, and to better make inferences on the basis of that data. A good taxonomy of faults, accompanied by an initial automated classification could allow test engineers to determine if there are any gaps in the current testing process, and draw conclusions about the existing software quality process. This is in line with Ralph’s stated benefit of increasing “cognitive efficiency and facilitate inferences” [25].

To get an overview of the quality of the clustering, we used the following methodology.

6.1 Clustering test cases generated by EvoMASTER

During the search, new search objectives are constantly added based on lines of code, conditions, and potential faults. The goal of any test suite is to cover as many of these search objectives as possible. A new test case is assessed for fitness with respect to all known and uncovered search objectives. Once an objective is covered, only one relevant test case is saved.

Since search objectives can be arbitrary in number, for a given SUT, the resulting number of test cases is also likely to be quite high. The result could be a large test suite, with a high number of test cases. For any users of EvoMASTER, it would be important to know how similar the test cases being generated are, and how they can be classified to either enable or at least support further efforts at classification and prioritization. An initial evaluation of test performance is based around characteristics relating to access to the RESTful APIs: the endpoint that is being accessed and the operation that is being carried out (i.e., the HTTP verb being used).

While this is useful information for tracking the faults and conducting root cause analyses, there is a possibility that different test cases trigger the same underlying problem through different endpoints. For example, a fault in processing certain input information, like dates, could manifest for a number of different operations. As each endpoint is called and the respective operations are executed, new lines of code will be covered, increasing the overall code coverage and thus making the test a useful addition to the test suite. From an analysis perspective, however, the new test cases do trigger the same fault and are similar to each other, for a given definition of “similarity”.

Thus, if a similarity measure could be proposed, a clustering algorithm could allow EvoMASTER to provide the users with an initial classification of the observed test cases, enabling test engineers to conduct a more effective classification and prioritization, as well as allowing for a better understanding of the overall quality of the SUT. Ideally, a small test suite containing just a representative test case from each failure cluster could be generated (i.e., an *executive summary*), to give test engineers an overview of what types of faults are present in the system they are testing and to enable them to prioritize their efforts and resources.

Thus, we implemented an addition to EvoMASTER that uses a clustering algorithm to classify the test cases based on a given set of distances. For the purpose of this study we have used the Density Based Spatial Clustering of Applications with Noise (DBSCAN)[15], and have implemented distances based on the error message associated with failed runs and the last executed line associated with failed runs. While the examples presented are focused around these choices, note that other clustering algorithms can be chosen, and more distance metrics can be added when relevant. The clustering was implemented as an addition to EvoMASTER. As the individuals developed by EvoMASTER are written out as JUnit test cases, the clustering is performed and relevant cluster labels are attached to each individual test case. Besides the main test suite that contains all the tests which contribute to coverage, a second test suite file (which we call the executive summary) is generated, with only one test representative per fault cluster.

Each EvoMASTER individual, or test case, is a collection of REST actions. Each action consists of an HTTP call with specific parameters: the endpoint used, the HTTP verb, input parameters, body payloads, and response information. However, an individual test case may consist of a number of failed, and potentially faulty, actions, and each of these actions may have a different underlying root cause. Thus, the clustering itself is conducted on

the set of REST actions that is present in all the test cases in the test suite (i.e., all the rest actions present in all the individuals in the final population). Once all the REST actions have been clustered, each test case is then marked with the labels of all the relevant clusters that its component actions are a part of. Since the test case (i.e., individual) is independently executable, it is a good unit of analysis and reporting for the types of faults it contains, and thus the cluster labels that are relevant to it.

6.2 DBSCAN and the distance metrics

The first goal is to cluster those test cases that have at least one call with a 500 return code, as those have been identified as being potential faults and likely of higher priority. Moreover, such calls also have additional information that can be used for classification: for example error messages and a record of the last executed line.

The clustering algorithm chosen is the Density Based Spatial Clustering of Applications with Noise (DBSCAN) [15]. DBSCAN is a clustering algorithm that creates clusters based on the density in particular regions of the space. Thus, DBSCAN requires a distance function to calculate pairwise distances between the individual test cases, a definition of what an acceptable neighborhood is for a cluster, and the minimum number of points that can form a separate cluster.

There are several reasons for choosing DBSCAN as an appropriate clustering algorithm for clustering test cases. DBSCAN does not require that the number of clusters is defined in advance. This is especially useful when applying the algorithm to clustering automatically generated test cases, that are the result of a stochastic search process. By not assuming the number of clusters in the test suites that EvOMASTER generates, we can provide a more robust assessment that can apply to novel problems or case studies, as well as known ones. The number of clusters that results is based on the values of the two parameters: epsilon - a definition of the neighborhood, and minPts - the minimum number of points that can form a cluster.

The algorithm is robust in terms of the shape of clusters (it can find arbitrarily shaped clusters), and in terms of noise and outliers. This is especially important when applying the algorithm to automatically generated test data that are relevant to different SUTs, and are difficult to validate manually.

The final parameter DBSCAN needs to cluster the test cases evolved by EvOMASTER is a measure of distance. To this end, we have defined two measures of distance between test cases that have been identified by EvOMASTER as faulty, according to the previous definition:

- **The textual similarity between error messages.** This is based on one of the first steps performed during manual classification and analysis of faults. The concept is that the error message (if any) attached to individual failed calls should provide developers and test engineers with some information regarding the cause of individual failures. Similar faults will result in similar error messages, and the distance between error messages can be used to cluster the resulting faults. Since error messages often contain specific information about the runtime state of the system, we cannot expect that they will be identical for different faults. Moreover, since the length of error messages may impact the similarity metric, the values are normalized to the length of the error message. A measure of textual similarity between error messages, based on the Levenshtein distance [21], is used to define the distance between two error messages, encoded as strings.

However, there are two potential downsides to this measure of distance:

- **General Error Messages.** Most frameworks that practitioners use to develop RESTful APIs provide some generic fault response. During development, it is expected that practitioners will override the default option with individualized error messages that are more relevant to individual faults and more informative. However, this practice cannot be enforced. If an SUT only provides the default, generic error messages, then no differentiation between faults can be determined.

- **Security Concerns.** It is not considered good practice for individual error messages to be propagated to the users of online systems. As a result, individual failures may be wrapped in other error responses that are more general and less informative. This could also reduce the relevance of distances based on the textual similarity between error messages.

As an example, calls that receive a response in the 4xx family, usually mean some user faults. Information about these faults should be quite detailed to enable the user to provide the appropriate inputs. However, faults in the 5xx family usually do not provide detailed error messages, as information about the internal state of the SUT, in particular information such as stack traces for failures, are considered sensitive information. Some frameworks do allow for testing configurations and more informative errors to be provided for a testing environment. However, those are not the default settings, and developers must enable such configurations to allow for improved testing.

- **The textual similarity of the path of the last executed statement.** As part of the search process, EvoMASTER instruments the code of the SUT to provide information about the runtime state. A part of this instrumentation is to provide failing test cases with a record of the last executed statement before the failure occurred. This is also based on the process by which developers and test engineers classify and analyze failures manually, by looking up the potential source of the fault. This measurement also has potential downsides:

- **Fault location.** The exact location of the last executed line may be imperfect. As an SUT fails, it is not always possible to track down the last executed line precisely. The last executed line recorded in the test cases generated by EvoMASTER refers to the last line of the SUT to be executed. However, this does not count the third-party libraries (e.g., there would be no point in tracking the last statement executed in the HTTP server before sending out a 500-code response). Should the failure be associated with the configuration of external code (e.g., a library or framework) before the SUT code is even executed, the last executed line will be recorded simply as “framework code”. Moreover, an individual fault may be triggered by calls from various modules and methods in the SUT. Execution may continue, even if the SUT is in an incorrect (and ultimately failing) state. The last executed statement might point towards the caller of the faulty method rather than the faulty section of code itself. An example of this is the case of try-catch blocks. The last successfully executed line could be in the “catch” or “finally” blocks, even if the failure is in the “try” block.
- **Precise identification.** The last executed line points to a particular line of code in the SUT. The last executed line, however, may contain several instructions that might have caused the failure. In addition, one issue that has been identified in the case studies presented is that of identical, or near-identical, code being duplicated on different lines. Thus, a fault in the same category, sometimes with the same failing instruction may be traced to different lines of code, while faults of different categories, throwing different exceptions, may share a last executed line.
- **Structure of the SUT.** Currently, the analysis of the last executed line is based on the textual similarity between strings. This is potentially problematic, as the textual similarity metric might be misled by the way modules and submodules are structured in the SUT. However, a tradeoff needs to be taken into account: this is meant to be a fairly general and low-effort approach for clustering the test cases, rather than an in-depth analysis of the structure of the SUT.

Nevertheless, the last executed line provides useful information for identifying and classifying faults, and supporting the classification process using automated tools could provide benefits to software developers and test engineers.

Note that the two distances are not, at the moment, combined into a single clustering effort. The clustering process is run once for each distance metric provided. The results are not combined at the HTTP call level, but rather at the test case level.

Our current implementation assigns cluster labels to each test case, rather than each test case action. Each test case is considered an independent individual, and can be moved in the ordering, assigned to other test suites or files, and generally can be executed in isolation or with other test cases. This allows the possibility to select, isolate, and only run those test cases that are assigned to specific clusters. One example of this being carried out is the reordering of individuals inside the generated JUnit files to enable those test cases that contain calls with a return code in the 5xx family to be first in the order of the output. This is done to highlight those test cases that are likely to contain faults.

However, an individual test case may contain several HTTP calls, including several calls that trigger different faults. Thus, it is entirely possible that an individual test case may be relevant to several fault clusters. During the clustering process, each call is clustered and assigned a cluster number, based on the parameters stated above. Once the clustering process is over, each individual test case receives all the cluster labels of all the clusters relevant to the calls it contains, for all the metrics that have been considered in the clustering process. This choice does, however, impact the analysis process, as individual test cases will exist that belong to more than one cluster.

The clustering is based on the same experimental data as the Taxonomy presented above: the test cases from each of the ten runs of the open-source case studies and the industrial case study were clustered using DBSCAN, as discussed above. The same set of “best” test suites was selected (i.e., the test suite with the highest number of covered targets) for a more in-depth analysis of the clustering.

To analyze the results of the clustering algorithm, we have used the same set of test cases as those used in the development of the taxonomy in Section 5. This allows a direct comparison between the tentative classification obtained manually and the clustering performed by DBSCAN.

6.3 Clustering overview

A test case generated by EvoMASTER usually contains one or more calls to the tested RESTful API. As EvoMASTER evolves more test cases and seeks to cover more search objectives, test cases can become more complex, with more heuristics being added [31]. This leads to increasingly large test cases, often containing multiple failed calls with different root causes. Since clustering is conducted based on information available at call level (error messages and last executed line), this means that each test case can belong to more than one cluster (be assigned more than one label).

Test case minimization, i.e., reducing the number of calls in an individual test case to the minimum necessary to cover new search objectives and illustrate particular faults, may help in making the classification clearer as well. However, calls may rely on the results of previous calls in the same test, and test cases with several failed calls are still possible.

Note that the manually assigned tentative classes are only useful for validation purposes, as we cannot expect this information to be available in practice for anything but a very limited number of cases. The goal of the clustering is precisely to inform developers which of the tests represent those interesting cases where such an evaluation is worth the time and effort.

To get a better understanding of the potential information that clustering can provide for test suites developed by EvoMASTER, we will focus on one case study in detail. For this purpose, we would favor a case study with a relatively high number of diverse faults, to ensure that classification is not completely trivial. As discussed above, some case studies also use frameworks that restrict access to error messages and other internal information. This renders clustering according to the call error message uninformative.

Class	Cohesion	Separation	Mis-classified
Seeding	13.51%	100.00%	0.00%
Framework.NotAccepted	3.28%	100.00%	0.00%
DB.Key	8.20%	100.00%	0.00%
MultipleExecution	3.28%	100.00%	0.00%
Specification.Custom.MissingOrNull	29.51%	64.29%	35.71%
Specification.Custom.PreApprovedList	14.75%	81.82%	18.18%
Fail.by.Succeeding	8.20%	100.00%	0.00%
NotFound	36.49%	96.43%	3.57%
Inconsistent.Unexpected500	8.20%	83.33%	16.67%
Unsafe.Cast	11.48%	100.00%	0.00%
Specification.Form.Type	2.70%	100.00%	0.00%
DB.Injection.Inconsistency	1.35%	100.00%	0.00%

Table 7. Overview of cluster analysis criteria. The data presented here is based on the test case selected for *proxyprint*.

The case study *proxyprint* has the benefit of having a large number of potential faults (see Table 3) and a large number of faults in the test suite selected for manual analysis (see Table 4). The faults found in the case study are also quite diverse, as shown in Table 6. In addition, the calls in the test suite have informative error messages that allow the clustering algorithm to be evaluated.

The case study *proxyprint*, thus, meets the requirements discussed above and had been selected for a more in-depth analysis, as a promising example of how the clustering algorithm can perform under favorable circumstances.

Table 7 shows the cluster analysis criteria, for the clusters resulting from the *proxyprint* test suite.

The **Cohesion** column shows the percentage of errors of a given fault type that are present in the same cluster. The ideal would be 100% cohesion, as that would mean that all the elements in the cluster are of the same tentative class. The values for cohesion in Table 7 indicate that the clusters proposed by DBSCAN are more general than the manual classification. A cluster proposed by DBSCAN contains more than one type of fault, as identified by the manual analysis. This could mean that the current clustering approach could be further investigated to determine more suitable parameters, or that more clustering approaches could yield better results.

The **Separation** column shows the proportion of faults in each manually identified category that are present in the same cluster. A cluster with 100% Separation would indicate that all the elements of that tentative class are present in the cluster (i.e., no elements of that tentative class are misattributed to a different cluster). The values in Table 7 show high values for separation, indicating that the clusters proposed by DBSCAN do separate between different tentative classes.

We interpret these findings to mean that there are clear categories of faults that can be identified by error message. This is somewhat intuitive, as different errors would have different error messages, and therefore end up in different clusters. This, however, is heavily dependent on two conditions being met: errors should have informative and distinct error messages that are available to the clustering algorithm (i.e., not default or inaccessible messages); and the clustering algorithm parameters should be improved.

As stated previously, a test case contains up to ten HTTP calls, and may reveal different faults, and thus may belong to several clusters. Further work on EvoMASTER, in particular test case minimization, could help in isolating faults that are independent from other calls. In such a case, cluster cohesion would depend more on the parameters of the clustering algorithm, which can be optimized, rather than on the test suite. This would make it easier to identify and isolate faults that cause particular types of errors. Moreover, by identifying faults that cannot be split, it may be possible to identify and highlight faults that are the result of more complex, related series of calls.

The previous results suggest that the clusters do separate the different classes of faults, i.e., the fault classes mostly exist in one cluster, as opposed to being spread across several clusters. For example, the **Unsafe Cast** row in Table 7 has a Separation of 100%. That means that all the faults of that class are (correctly) classified in the same cluster. For the same class of fault, however, the Cohesion is only 11.85%. This means that of all the faults that are grouped in that cluster, only 11.85% are of the relevant class. Ideally, the cluster that represents the **Unsafe Cast** faults would be separated into their own cluster.

This indicates that the clustering can be used, to some extent, to separate different classes from each other. The information in Table 7 also shows the cohesion of the clusters to be low. This suggests that each cluster contains more than one class of fault. This conclusion is verified by a quick check of the raw data. We interpret this result to mean that, while the clustering does an adequate job of separating different classes into different clusters, the clusters themselves are not cohesive. A cluster, therefore, cannot be said to represent a class of fault, but more to be a bin where several classes of fault, that manifest in similar ways or have similarity in terms of error message, are grouped.

More in-depth details on the clustering analysis (e.g., where clustering failed) is provided in Appendix.

Initial results show that clustering can separate between different types of faults, and has the potential to provide test engineers with automated support when classifying large numbers of test cases.

7 DISCUSSION

This section will discuss the results presented above, and detail their implications, both academic and practical.

7.1 Taxonomy

The taxonomy emerged out of a need to understand the types of faults that are found by an automated testing tool applied to RESTful APIs. It is based on empirical data, obtained from seven open-source case studies and one industrial case study. The test cases generated by EvoMASTER provide the examples of faults, and manual analysis of their root causes allow their classification and provides the structure of the taxonomy.

The taxonomy is meant to be a flexible and robust description of the faults that are encountered in practice, not only in the case studies provided or by applying EvoMASTER, but for a more general overview of software faults present in RESTful APIs. The underlying assumptions are that test cases can be independently run, to provide repeatable examples of faults. Access to the code of the target SUT is recommended. This improves the performance of EvoMASTER, as well as providing the necessary information to analyze and classify the faults that are uncovered.

In practice, it is expected that new types of faults, new variants of categories or subcategories, will constantly emerge. This could be the result of adding more case studies with more varied faults, or as a result of the improving quality and increased use of automated software testing tools. For example, those case studies could still have uncovered faults, hidden in parts of the code that is still not executed/reached during the test generation, and that require improved heuristics to generate test data that can reach the execution of those parts of the SUT code. As a result, the taxonomy includes guidelines on adding new categories and refining existing categories, to better

describe the fault that are empirically observed. It is unlikely that faults and fault categories are static, so it is expected that the taxonomy itself will undergo changes, improvement and adaptation to better meet its goal.

An interesting aspect of the taxonomy is that it contains a large variety of faults, ranging from configuration issues in the frameworks used for API development and deployment, to faults in the business logic of the case study SUTs. This suggests that automated software testing tools in general, and EvoMASTER in particular, are able to find a wide variety of faults in the systems they test. This could indicate the potential for improvement in a variety of areas. Faults in the business logic of the APIs can be addressed by the developers of the respective SUTs, as can problems identified with database connections.

The current implementation of EvoMASTER identifies bugs based on two types of criteria. First, any response with a 500 code is considered a possible fault, as this is considered undesirable behavior in general. This is equivalent to a crash oracle, as a 500 response, for our case study SUTs, is often given when an exception is thrown and a crash occurs. In practice, this often means situations where incorrect or invalid inputs are mishandled, resulting in a crash, and a mistaken 500 response instead of a more proper handling and a 4xx response. Since the most common type of 500 fault observed is a mismatch between the OpenAPI schema that is used to generate valid inputs, the most common type of fault found consists of mishandling invalid inputs. Thus, invalid inputs are not identified as such, either by the OpenAPI schema, the frameworks used, or by any checks in the SUT itself. When the invalid inputs are used, crashes occur, for example, mishandling data formats, erroneous types being provided, values or value combinations that have a particular type of semantics that limits their valid range.

However, we do see faults that do not fall into that category: i.e., faults that are found in spite of the input values being valid. Hardcoded values in the SUT business logic appear to lead to crashes even when the input values are, as far as can be determined, valid. The same is true for the unsafe casts. Framework configuration issues are often encountered before the inputs are processed at all, so it is difficult to assess their precise nature, but their severity appears to mark them out as important faults to address. In short, the evidence shows that the tests generated by EvoMASTER do reveal significant faults in the SUTs included in the current case study. Moreover, the selection of oracles used to identify potential faults is being increased. More complex relationships between resources are introduced to guide the search [31], and metamorphic relationships can provide additional oracles to identify potential faults [27]. This will further increase the variety of faults found using automated testing tools.

Mismatches between the OpenAPI schemas and the SUTs they describe are themselves quite interesting, as most of the OpenAPIs for the case studies presented in this work are automatically generated. This suggests that improvements can be made in the tools that automatically derive such schemas from the code. It also indicates that developers need to understand these automated tools and find configurations and parameters that are appropriate for their settings.

7.2 Automated Classification Support

As shown, EvoMASTER is able to generate a large number of test cases for some of the more complex APIs presented in this study. Increased complexity of the SUTs leads to an increase in the search objectives that EvoMASTER will try to cover. Improvements in technology and development of automated software testing systems will also increase coverage of the resulting tests. All these factors seem to indicate that automatically generated test suites tend to increase in size, in order to ensure coverage of the SUT. Already, EvoMASTER generates a large number of test cases (more than 200 for *proxyprint* and *scout-api*), many of which contain several calls and include code for database insertion. Many of the test cases in question showcase frequently encountered, low severity, faults that are often over-represented in terms of their number. All these factors show

the need for automated support in filtering and classifying these test cases, and for providing test engineers with information on what types of faults have been identified in their code.

The variety of the faults found in the case studies included in this work highlights an interesting point in automated software testing in general: solutions that are found should be as robust and as generalizable as possible. For this study we selected the DBSCAN clustering algorithms precisely because of its relative robustness: no assumptions need to be made about the number of different clusters present in the data, and the algorithm itself is quite robust in terms of the topology of the space it clusters. Clustering relies on information being available for use. This means that access to information, to the code of the SUT being analyzed, is essential. This improves both the performance of EvoMASTER itself, by allowing that code to be instrumented and providing better coverage, but provides also for a more detailed analysis of the faults that are found, including information such as the last executed line, information of the exceptions encountered, the stack traces of those exceptions, and debugging information. Thus, we encourage the use of such tools as white-box, where possible, allowing them to leverage all the information available.

The analysis of cluster separation and cohesion indicates that the clustering algorithms does a good job in separating distinct categories of faults, but the resulting clusters are not coherent, i.e., several categories of faults are found in the same cluster, but a category of faults is only found in one cluster. The main sources of information used for clustering in this study, text of error messages and the last executed line, are both likely to be quite varied in structure and offer some difficulties for text based analysis. Error messages often contain object references or values that change between executions, and similarity measurements in these cases are not trivial. Better results could be obtained by combining the text similarity metric with information on the types of exceptions encountered. This would allow the clustering algorithm to separate existing clusters to better describe the underlying fault categories.

It should also be noted that fault clustering relies on the faults being distinct enough to allow for separation to occur. We recommend that developers make full use of test settings and environments provided by the frameworks they use, to allow automated testing tools to access relevant information regarding error messages, thrown exceptions, etc. Moreover, we recommend that informative error messages are provided, to help with automated classification, as well as to enable manual analysis.

The large number of **Unsupported Code** and **Schema Mismatch** faults found seems to indicate that current schema are under-specified. As automated testing tools increase in coverage, quality, and availability, this issue could be a detriment to developers. As shown in this study, automated software testing tools could create large numbers of low severity faults that may be considered false positives. Refining OpenAPI schemas to better describe the RESTful APIs could help avoid such false positives, as well as improving search results, preventing automated systems from wasting search budget on irrelevant areas of the input space.

7.3 Future work

Several potential directions for future work have already been hinted at. As more case studies are added to EvoMASTER, the taxonomy will be further expanded and refined, and future experiments could also provide additional information about the types of faults found. In addition, continuous development of EvoMASTER and increases in the level of coverage it provides could also reveal new fault classes in the SUTs used in this paper.

The initial experience of clustering test cases also provides a useful guide to further developments. First, clustering can be further improved, either by identifying useful default values or a mechanism by which some of the parameters may be tuned. This would result in more useful clustering and more informative labels for them. An overview of the current clustering results reveals that the length of the error messages plays an important part in the distance evaluation, in spite of the efforts made to normalize the distance value by the length of the string. Determining the interaction between the various error messages, the impact of their length on the distance

metrics, and how those distance metrics are later used by the clustering algorithm could provide significant improvements in the resulting clusters. Additional information, such as exception types, can also be used to improve clustering results where available. Note that such information may not be available, for a variety of reasons. These can range from using default framework error handling for all SUT failures to intentionally obscuring the SUT internal state to increase security. In these cases, new distance metrics would have to be developed and evaluated.

Ultimately, the goal is to facilitate the use of automated testing in general, and EvoMASTER in particular, in industry. To this end, the communication between the test engineers using EvoMASTER for practical projects and the tool itself can be improved. Clustering test cases and potential errors by their similarity is one effort to provide the human users of EvoMASTER a means of more efficiently analyzing and prioritizing large amounts of potential failures.

Therefore, a clear line for future work is that of industrial validation and determining where in the software engineering process EvoMASTER provides the most benefit and most contributes to the improvement of software quality. This also involves defining specific preconditions for employing automated test systems and an assessment of the impact they have on the software development process. A key aspect of EvoMASTER is the effort it makes to minimize manual labor in term of setup and running. While some components need to be written manually, running EvoMASTER, from data generation, code instrumentation, repeated runs, and code generation are all automated.

Work presented in this paper could be useful in terms of providing methods for the assessment of EvoMASTER and other automated tools, and making an informed decision regarding the fit of such tools for particular industrial applications. In particular, the amount of resources the system has at its disposal is an important factor in its successful application. EvoMASTER uses an evolutionary algorithm as its basis. Longer running times are often considered better, with more tests being generated, overall a higher level of code coverage, and a higher likelihood of consistent results. However, the exact number of fitness evaluations is dependent on the SUT as well as a host of other factors. Higher complexity SUTs, a higher number of endpoints, and complex relationships between the resources used can all increase the number of HTTP calls required to consistently find faults, and increase the size of resulting test suites. Clustering faults could provide a means of assessing, quickly and efficiently, the performance of such tools in practice.

8 THREATS TO VALIDITY

Conclusion validity. Search-based algorithms include an element of randomness that could have an impact on the outcome of the search. We conducted the search on each of the case studies 10 times, to ensure that the dataset used is not an outlier in terms of performance. The results of all the runs are presented in Tables 2 and 3. The results do not indicate that any of the runs was a particular outlier in terms of performance. To ensure the best chance of finding a wider variety of faults, the run with the highest coverage, as measured by the number of covered objectives, was selected for manual analysis.

Construct Validity. For this study, we used the number of HTTP calls as a stopping criterion for the EvoMASTER search. All runs had the same search budget of 100,000 HTTP calls. All runs has the same stopping criterion [1] and the stopping criterion selected was the number of HTTP calls that are part of the fitness function evaluation [29].

Internal Validity. The taxonomy and the results of this study rely on the manual evaluation and classification of 415 faults found by the current implementation of EvoMASTER. While every effort has been made to ensure the quality of the code, the absence of faults in EvoMASTER itself cannot be guaranteed. EvoMASTER continues to be maintained and supported, and further improvements are being conducted. The implementation of EvoMASTER is open-source, so the code can be reviewed.

Given the manual analysis of the results, the validity of this study could be affected by faults being mistakenly identified and/or classified during the process. To ensure consistency in terms of fault identification and classification, all faults were analyzed by one of the authors. The results were then checked by a second author, to ensure that the results were valid and that no mistakes had crept in the analysis. Where disagreements or differences in classification occurred, the faults were discussed until agreement was reached. No additional mechanisms for ensuring consensus were needed for the classification.

External Validity. External validity is heavily dependent on the case studies that were used as a basis for the taxonomy. To ensure that the case studies selected were representative of as broad a class of RESTful APIs as possible, we used a diverse selection of case studies. Our selection included open-source systems, including both artificial and real systems; as well as an industrial system. However, RESTful APIs are quite varied, and additional types of faults may be found in other systems. The taxonomy is designed to address this threat. The taxonomy not only describes the fault classes as found in the study, together with their descriptions, and a means to classify new faults into those classes, but also provides a method for extending the taxonomy to accommodate new fault classes and categories, and to relate them to the existing structure and hierarchy. Moreover, the structure itself may change, as fault categories may be split or added at the appropriate levels. This allows the current taxonomy to be expanded in order to fully describe faults that are present in other case studies. While further work could always lead to improvements and refinements, the first step towards addressing this threat is to provide those mechanisms allowing new information and evidence to be accounted for without making the findings presented here obsolete.

9 CONCLUSIONS

In this paper, we propose a taxonomy of faults discovered by EvoMASTER in RESTful APIs. The taxonomy reflects 415 faults found and identified in one industrial and seven open-source case studies. It described a diverse set of faults found in the case studies, ranging from RESTful specific framework misconfigurations, and schema mismatches, to generic software faults. The taxonomy we propose aims to enable test engineers and developers to better understand and prioritize software faults and improves cognitive efficiency in analyzing such faults. In addition, we make the argument that the diversity in terms of categories of faults contained in the taxonomy also shows that automated software testing can find a wide range of potential faults.

We further conclude that the large number of **Unsupported Code** and **Schema Conflict** faults indicates that current OpenAPI schemas are under-specified. This could have a detrimental impact in future, as the increased quality and more widespread adoption of automated systems could generate large amounts of false positives. Efforts to improve the quality of provided schema would avoid wasting search budget on generating test cases that are not relevant, as well as reducing the time and effort developers have to spend analyzing that data.

In addition to the taxonomy, we propose an initial means of automatically classifying software faults based on their similarity, as determined from the text of their error messages. If the error messages are informative, rather than generic placeholders, the clustering approach does separate between several categories of faults (although cluster cohesion is still an ongoing problem). This could provide practitioners with low cost, low effort support in analyzing and prioritizing the software faults that are found. To this end, we extended EvoMASTER to output executive summary JUnit files based on clustering of the detected faults. EvoMASTER is open-source, and freely available at www.evomaster.org

ACKNOWLEDGMENTS

This work is supported by the Research Council of Norway (project on Evolutionary Enterprise Testing, grant agreement No 274385).

REFERENCES

- [1] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering (TSE)* 36, 6 (2010), 742–762.
- [2] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification, and Reliability* 16, 3 (2006), 175–203. <https://doi.org/10.1002/stvr.v16:3>
- [3] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.
- [4] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology (IST)* 104 (2018), 195–206.
- [5] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.
- [6] Andrea Arcuri. 2020. Automated Blackbox and Whitebox Testing of RESTful APIs With EvoMaster. *IEEE Software* (2020).
- [7] A. Arcuri and L. Briand. 2011. Adaptive Random Testing: An Illusion of Effectiveness?. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 265–275.
- [8] Andrea Arcuri and Juan Pablo Galeotti. 2020. Handling SQL Databases in Automated System Test Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–31.
- [9] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software* 6, 57 (2021), 2153.
- [10] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (Montreal, Quebec, Canada) (ICSE '19)*. IEEE Press, 748–758. <https://doi.org/10.1109/ICSE.2019.00083>
- [11] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. 2013. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability (STVR)* 23, 4 (2013), 261–313.
- [12] Gerardo Canfora and Massimiliano Di Penta. 2009. Service-oriented architectures testing: A survey. In *Software Engineering*. Springer, 78–105.
- [13] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. 181–190.
- [14] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. 2005. Leveraging user-session data to support Web application testing. *IEEE Transactions on Software Engineering* 31, 3 (2005), 187–202. <https://doi.org/10.1109/TSE.2005.36>
- [15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.
- [16] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Ph. D. Dissertation. University of California, Irvine.
- [17] Ines Hajri, Arda Goknil, Fabrizio Pastore, and Lionel C. Briand. 2020. Automating System Test Case Classification and Prioritization for Use Case-Driven Testing in Product Lines. [arXiv:1905.11699 \[cs.SE\]](https://arxiv.org/abs/1905.11699)
- [18] M. Harman and B. F. Jones. 2001. Search-based software engineering. *Journal of Information & Software Technology* 43, 14 (2001), 833–839.
- [19] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.
- [20] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.
- [21] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.
- [22] Alberto Martin-Lopez, Sergio Segura, Carlos Muller, and Antonio Ruiz-Cortés. 2021. Specification and automated analysis of inter-parameter dependencies in web APIs. *IEEE Transactions on Services Computing* (2021).
- [23] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In *Service-Oriented Computing*, Eleanna Kafeza, Boualem Benatallah, Fabio Martinelli, Hakim Hacid, Athman Bouguettaya, and Hamid Motahari (Eds.). Springer International Publishing, Cham, 459–475.
- [24] X. Nguyen, P. Nguyen, and V. Nguyen. 2019. Clustering Automation Test Faults. In *2019 11th International Conference on Knowledge and Systems Engineering (KSE)*. 1–7. <https://doi.org/10.1109/KSE.2019.8919435>
- [25] P. Ralph. 2019. Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering. *IEEE Transactions on Software Engineering (TSE)* 45, 7 (2019), 712–735.
- [26] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. Souter Greenwald. 2007. Applying Concept Analysis to User-Session-Based Testing of Web Applications. *IEEE Transactions on Software Engineering* 33, 10 (2007), 643–658. <https://doi.org/10.1109/TSE.2007.70723>

Multiple Fault Example

```
List list = namedQuery("User.byIdentity").setParameter("type", type).setString("value", value).list();
```

Fig. 5. An example of a line of code in the SUT (in this case *scout-api*) that can be related to faults in several categories. In this case, problems included crashes due to invalid value type, the values to be assigned being null, or the semantics of setting individual parameters (for example, a user’s attempt to increase their own authorization level resulted in a crash).

- [27] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2017. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering (TSE)* (2017).
- [28] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. 2017. Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method. *Information and Software Technology (IST)* 85 (2017), 43 – 59. <https://doi.org/10.1016/j.infsof.2017.01.006>
- [29] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. 2013. Exploration and Exploitation in Evolutionary Algorithms: A Survey. *ACM Comput. Surv.* 45, 3, Article 35 (July 2013), 33 pages. <https://doi.org/10.1145/2480741.2480752>
- [30] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.
- [31] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based test case generation for RESTful web services. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1426–1434.
- [32] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) (*ISSTA 2014*). Association for Computing Machinery, New York, NY, USA, 385–396. <https://doi.org/10.1145/2610384.2610404>

APPENDIX

This appendix contains additional information about the clustering analyses carried out in this paper.

Failed clustering

In some cases clustering has failed to provide any useful information. For example, *scout-api* wraps all errors by default and removed all information from the error messages, with the exception of the “code 500” text. This results in all the test cases being clustered together, as all the faults appear to have identical error messages. A similar problem occurs with *feature-service*, which removes all text from the error, leaving the system unable to conduct clustering according to the error message at all.

Clustering according to the last executed line proved to be problematic, as the distance between two “last executed lines” components is somewhat uneven. Faults present in deeper classes will be more similar (as class, sub-module, and module names might be the same). The last executed lines do seem to provide interesting information, but more study is needed to determine how precisely to leverage that information. Rather than textual similarity, clustering based on the last executed lines could focus on the structure of the code, highlighting modules and classes that seem to be the root source of failures. This, combined with information from informative error messages could indicate if there is a correlation between classes and failure types.

Where differences are clear, however, the additional labels are useful. For example, for the industrial case study *ind0*, faults that are related to **Framework Misconfiguration** are clearly clustered together by the last executed line, which in that case is `framework` code.

Examples were also found of several categories of faults relating to the same line of code. The code, seen in Figure 5, shows a line of code in the *scout-api* case study, that deals with modifying parameters for a given user. This line of code is associated with a number of different fault categories. Note, all the categories described here involve crashes of the SUT and 500 responses to the client.

First, some problems were due to the formatting of dates being considered invalid by the database constraints. These dates appeared to be correct and passed other checks in the SUT. A second category consists of constraints

that have specific semantics in the context of the SUT. For example, an attempt by a user to increase their own authorization level would fail that that line of code with a “`javax.ws.rs.WebApplicationException: Cannot set authorization level to higher than your own`” exception. All these are traceable to the same line of code, and the exact faults can only be determined as a result of manual analysis. However, pinpointing this line of code as a source of several fault categories could be useful information when prioritizing faults.

Clustering according to the error text provides an interesting perspective: looking at the similarity between error messages does provide useful insight into the types of problems that were encountered. Moreover, similar error messages could be indicative of the same fault in different areas of the code. This could be an indication of duplicated code, or could indicate areas of the code that are reached often and from different endpoints and operations. It is also a fairly intuitive first step, to start from the kind of information that test engineers and developers would also use to conduct root cause analyses and to begin debugging.

However, this approach is meaningful only if the error messages themselves are meaningful. In some of the case studies, all error messages were handled by their respective frameworks with the default values, and with no customization. This leads to individual error messages either being identical and uninformative (thus not allowing for a meaningful classification) or completely missing (with the error being simply the error code). The example presented above, *scout-api*, wraps all the responses from the SUT to prevent internal information from being exposed to users. This means removing items such as stack traces, exception descriptions, and error messages from the response itself. As a result, all responses in the 500 family fall into the same cluster, since they all have the same error message: “Internal Server Error”. Thus, clustering is not successful in aiding classification where error messages are not informative. We will, therefore, focus the analysis on those case studies where the error messages were present, in the expected format (JSON).

Clustering Analysis

The analysis presented in this section will focus on two of the case studies: *proxyprint* and *catwatch*. These are the systems where error messages were informative enough to allow an automated analysis to take place. Table 8 shows examples of cluster assignments according to the error message, for the *proxyprint* and *catwatch* case studies. Note that during clustering, some measure of similarity was used, but error messages are not expected to be completely identical, as they might depend on the input data.

It is interesting to note that informative error messages often contain detailed information about the values that are being used, and include sections of text that would repeat, contain paths for various classes, and contain exact values that might change between separate runs. This leads to an interesting conundrum: error messages are unlikely to ever be identical, but long segments are likely to repeat. This strengthens the validity of clustering, as a means of classification. For example, Table 8 shows 3 examples (marked in bold) showing a similar fault: the values included do not belong to a pre-approved list of values. The error messages in question are different, as they refer to different values but also include the actual value generated by EVO MASTER. This is useful for debugging, but the same values will not be repeated between two runs. In our example, the values are correctly clustered together, albeit not separated from other values.

Table 8 also shows two clear examples of clusters that contain only one type of fault. The `NumberFormatException` in cluster “ErrorText_2” for *proxyprint* and the `JpaSystemException` in cluster “ErrorText_2” in the *catwatch* case studies. This suggests that, for appropriately informative and distinct error messages, separate clusters with individual types of faults may be created.

Sensible default values, that work for a generic SUT, need to be found. Simultaneously tools for validating the clusters, optimizing clustering parameters, and using domain specific knowledge to interpret the findings of, or even improve, the clustering process could provide additional benefits.

SUT	Cluster	Exception	Message
proxyprint	ErrorText_0	org.springframework.web.multipart.MultipartException	Could not parse multipart servlet request; nested exception is javax.servlet.ServletException: org.apache.tomcat.util.http.fileupload.FileUploadBase\$InvalidContentTypeException: the request doesn't contain a multipart/form-data or multipart/mixed stream, content type header is application/json
		org.springframework.dao.DataIntegrityViolationException	could not execute statement; SQL [n/a]; constraint [null]; nested exception is org.hibernate.exception.ConstraintViolationException: could not execute statement
		org.springframework.orm.jpa.JpaSystemException	Null value was assigned to a property [class io.github.proxyprint.kitchen.models.notifications.Notification.read] of primitive type setter of io.github.proxyprint.kitchen.models.notifications.Notification.read; nested exception is org.hibernate.PropertyAccessException: Null value was assigned to a property [class io.github.proxyprint.kitchen.models.notifications.Notification.read] of primitive type setter of io.github.proxyprint.kitchen.models.notifications.Notification.read
		org.springframework.orm.jpa.JpaSystemException	Null value was assigned to a property [class io.github.proxyprint.kitchen.models.consumer.printrequest.PrintRequest.cost] of primitive type setter of io.github.proxyprint.kitchen.models.consumer.printrequest.PrintRequest.cost; nested exception is org.hibernate.PropertyAccessException: Null value was assigned to a property [class io.github.proxyprint.kitchen.models.consumer.printrequest.PrintRequest.cost] of primitive type setter of io.github.proxyprint.kitchen.models.consumer.printrequest.PrintRequest.cost
		org.hibernate.PropertyAccessException	Null value was assigned to a property [class io.github.proxyprint.kitchen.models.notifications.Notification.read] of primitive type setter of io.github.proxyprint.kitchen.models.notifications.Notification.read
		java.lang.IllegalArgumentException	No enum constant io.github.proxyprint.kitchen.models.printshops.pricetable. Item.RingType.evomaster_378_input
		java.lang.IllegalArgumentException	No enum constant io.github.proxyprint.kitchen.models.printshops.pricetable. Item.Colors.bpmRf
		java.lang.IllegalArgumentException	No enum constant io.github.proxyprint.kitchen.models.printshops.pricetable. Item.CoverType.fyU4k0IAU1pXz
		org.springframework.orm.jpa.JpaSystemException	could not execute statement; nested exception is org.hibernate.exception.GenericJDBCException: could not execute statement
		org.springframework.dao.EmptyResultDataAccessException	No class io.github.proxyprint.kitchen.models.notifications.Notification entity with id 233 exists!
		java.lang.ClassCastException	org.springframework.security.web.servletapi.HttpServlet3RequestFactory \$Servlet3SecurityContextHolderAwareRequestWrapper cannot be cast to org.springframework.web.multipart.MultipartHttpServletRequest
	ErrorText_1	java.lang.NullPointerException	No message available
	ErrorText_2	java.lang.NumberFormatException	Name is null For input string: " tBfdp8M "
catwatch	ErrorText_0	java.lang.IllegalArgumentException	fromIndex(314456) > toIndex(0)
	ErrorText_1	java.lang.IllegalArgumentException	toIndex = 10
		java.lang.UnsupportedOperationException	this parameter configuration is not implemented yet .. start date, end date required atm
		java.lang.IllegalArgumentException	offset must be greater than zero but was -1797018441
		java.lang.IllegalArgumentException	sortBy must be empty or have a valid value but was 7cOjajxPYU3OI. Valid values are organizationalCommitsCount, organizationalProjectsCount, personalCommitsCount, personalProjectsCount, organizationName.name
	ErrorText_2	org.springframework.orm.jpa.JpaSystemException	Null value was assigned to a property [class org.zalando.catwatch.backend.model.Project.gitHubProjectId] of primitive type setter of org.zalando.catwatch.backend.model.Project.gitHubProjectId; nested exception is org.hibernate.PropertyAccessException: Null value was assigned to a property [class org.zalando.catwatch.backend.model.Project.gitHubProjectId] of primitive type setter of org.zalando.catwatch.backend.model.Project.gitHubProjectId

Table 8. Examples of clustering according to error messages for the *proxyprint* and *catwatch* case studies. The exception in bold, **Illegal Argument Exception**, showcases one of the difficulties of automated classification: While the error messages are of the same structure, they contain the different faulty inputs, thus pointing to different faults. The examples *proxyprint* → *NumberFormatException* and *catwatch* → *JpaSystemException* showcase how clustering faults by error message could be successful: with appropriately informative error messages, a clear (albeit tentative) classification can be provided to the test engineers automatically.

In addition, the clustering process can be refined by using additional information, where available. If error messages are available, additional information, such as the exception class, can further be used to differentiate between different fault categories. Exception classes are less likely to contain information that will change between runs.

Initial results show that clustering has the potential to be a useful tool in supporting the classification of faults. In particular, there is the potential of correct identification of uncommon faults, with clearly distinct error reporting, that could otherwise be easily overlooked.

Automated support for fault classification seemed able to separate different failure categories into different clusters, when these failures were accompanied by informative, structured error messages. Results show that the clusters themselves are not as cohesive and as representative of one type of fault as desirable. This can be improved by refining the clustering, but is also dependent on the types of error messages that are present in the SUT, and on how informative and structured those messages are.

For each test suite, a subset of test cases is selected, with one representative selected from each automatically derived fault category (i.e., cluster). We denote this subset as the “executive summary”, the purpose of which is to give test engineers an overview of the different types of faults that are found, as classified by the automated system, in the form of an executable test suite. From each cluster, the shortest representative test case is considered. If the test case is already included in the summary, then it is skipped. This ensures that the summary is kept as clear as possible, free of duplicate test cases, and only showing the shortest available examples for each type of fault found.

The effectiveness of the executive summary depends on the quality of the clustering, and thus the quality of the data used. In the current version, the executive summary presents one test case from each identified cluster. This can be useful in separating some of the classes of faults, for example, identifying the **Framework Misconfiguration** faults in the *ind0* industrial case study. However, since the current clusters can contain more than one type of fault, selecting only one representative can be misleading.