# FERRARI: FailurE RepRoduction through automatic test cAse generation and stack tRace analysIs

Valeria Pontillo*, Maarten Vandercammen, Sarah Verbelen and Coen De Roover

*Software Languages (SOFT) Lab — Vrije Universiteit Brussel, Belgium*

Abstract

As cloud services grow increasingly complex, their REST APIs become more susceptible to failures caused by unforeseen interactions between API calls or bugs in service implementations. Debugging these failures involves two key steps: reproducing the failure and identifying the root cause. In distributed environments, reproducing failures is particularly challenging due to the need to reconstruct the intricate sequence of API calls that lead to the issue. Identifying the defect then requires analyzing stack traces, which can be difficult and time-consuming due to the volume of logs and the variability of interactions across the system. While tools like RESTler can automate testing and uncover bugs, the manual analysis of stack traces to pinpoint the root cause of failures remains challenging and time-consuming. To streamline this process, we propose FERRARI, an extension of RESTler that automates the mapping of stack traces to generate targeted test cases for failure reproduction. FERRARI introduces a novel similarity scoring mechanism to quantify how closely the behavior of generated test cases matches the conditions of the initial failure, enabling efficient reproduction and diagnosis. By filtering irrelevant test cases and focusing on high-similarity candidates, FERRARI reduces the number of requests sent while preserving precision. In this way, FERRARI helps developers by reducing the number of stack traces that need to be manually analyzed. Our evaluation demonstrates that FERRARI achieves the same level of failure reproduction as baseline fuzzing strategies but with significantly fewer requests and error logs, offering a scalable and effective solution for developers.

**Keywords**

REST API, Software Testing, Failure Reproduction.

## 1. Introduction

In modern software development, cloud services are commonly accessed through REST APIs, which offer flexible and scalable interactions between clients and servers. These expose various endpoints that facilitate communication and data exchange, making them a critical component of distributed systems and microservice architectures [1]. However, due to their complexity, REST APIs are prone to failures, often resulting from unforeseen interactions between API calls, invalid input sequences, or bugs in the service implementation. When such failures occur, developers must engage in the challenging task of detecting and fixing the cause [2, 3].

*BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21-22, Namur, Belgium*

*Corresponding author.

Valeria.Pontillo@vub.be (V. Pontillo); Maarten.Vandercammen@vub.be (M. Vandercammen); Sarah.Verbelen@vub.be (S. Verbelen); Coen.De.Roover@vub.be (C. De Roover)

0000-0001-6012-9947 (V. Pontillo); 0000-0001-7192-5666 (M. Vandercammen); 0009-0006-7036-4311 (S. Verbelen); 0000-0002-1710-1268 (C. De Roover)

CEUR Workshop Proceedings (CEUR-WS.org)

Debugging REST APIs consists of two essential steps. The first crucial step is *reproducing the failure*. This is particularly important in REST API environments, as the distributed nature of cloud services and the stateful interactions between multiple API endpoints can make it difficult to recreate the exact conditions that triggered the failure [4, 5]. REST APIs are designed to handle stateless transactions, yet the interactions between sequential API calls can introduce dependencies that create stateful behaviors. As a result, reproducing failures in such systems often requires reconstructing the sequence of requests that led to the issue. Unfortunately, if the request sequence or the service's internal state is unclear, reproduction becomes a significant challenge, delaying both diagnosis and fixing.

The second step is *identifying the defect*. Once the failure has been reproduced, the developer needs to trace the cause of the failure through the system's internal state and execution logs. This step can be extremely complex in distributed cloud environments, as it involves correlating API calls with backend service states and navigating through the vast number of possible program states. Additionally, the asynchronous nature of cloud services and the diverse behavior of API interactions, such as cross-service communication, versioning issues, and stateful versus stateless design, can obscure the root cause, further complicating the debugging process. Identifying the defect typically involves analyzing stack traces, which provide a snapshot of the function calls leading to the failure [6]. However, the amount of data and the various API interactions make this process labor-intensive.

To streamline this process, many tools have been developed to automate parts of the debugging process. One such tool is RESTler [7], a stateful REST API fuzzing tool designed for automatically testing cloud services through their REST APIs. This tool systematically explores the API space by generating and executing test cases that consist of various input sequences to expose potential bugs, including security vulnerabilities and reliability issues.

Despite the powerful fuzzing capabilities of RESTler, identifying specific causes of failures based on stack traces—log outputs that reveal the sequence of function calls leading up to an error—remains a challenge. Manually analyzing stack traces is a time-consuming process, as developers must often sift through numerous test cases to find ones that yield similar errors or bugs. Moreover, the complexity of cloud services, with their distributed nature and numerous points of interaction, makes this task even more difficult.

To overcome these limitations, we propose FERRARI, a RESTler extension that automates the process of mapping stack traces to potential causes by generating targeted test cases. FERRARI takes as input the stack trace to reproduce as well as a trace of all methods invoked during the computation. The stack trace provides information on the methods involved at the moment an error occurs, capturing the sequence of method calls that led to the failure. In contrast, a method trace offers a complete record of all methods executed during the program's runtime, enabling a comprehensive view of the system's behavior, including interactions and state changes that may not be evident in the stack trace alone. These inputs are necessary to understand the flow of events that generated the failure and allow us to reproduce this error. Then, FERRARI is able to create and execute a series of test cases that aim to replicate the failure encountered. By comparing the input stack trace to the results of these test cases, FERRARI narrows down the possible causes of failure, allowing developers to identify the defect more quickly and accurately. This approach leverages the fuzzing capabilities of RESTler but tailors them to enable the reproduction of failure and the discovery of new bugs based on stack traces. The contribution of FERRARI is automating the mapping and analysis of stack traces to generate new test cases focused on specific bugs. In this way, developers are able to reproduce the failure as well as reduce the manual effort.

**Structure of the paper.** Section 2 explains the main functionalities of RESTler and the most closely related works. Section 3 presents the architecture of FERRARI. Section 4 reports a running example to show how FERRARI works, while Section 5 shows a preliminary evaluation of our adaptation. Finally, Section 6 concludes the paper and outlines our future research agenda.

## 2. Background and Related Work

This section provides an overview of RESTler and the literature related to our proposal.

### 2.1. RESTler: A Stateful REST API Fuzzing Tool

RESTler [7] is an automatic REST API fuzzing tool designed to test stateful cloud services through their APIs. The tool analyzes the API specification (typically in Swagger/OpenAPI format) and generates sequences of requests that are tested against the service. The primary functionalities of the tool include:

**Dependency Inference:** RESTler automatically infers dependencies between request types. For example, if one request (A) produces an output needed by another request (B), it understands that B should only be executed after A. This dynamic learning process helps optimize test coverage by exploring valid service states while minimizing redundant or invalid requests.

**Dynamic Feedback Analysis:** During testing, RESTler collects and analyzes responses from the service to learn from previous test executions. It uses this information to adjust its test generation, avoiding request sequences that the service has already rejected, thus optimizing the test coverage and improving the efficiency of the fuzzing process.

**Test Sequence Generation:** RESTler generates test cases that consist of sequences of multiple API requests. By considering both the inferred dependencies and dynamic feedback from earlier test runs, RESTler explores deeper states of the service, increasing the probability of uncovering bugs that simpler stateless fuzzing tools might miss.

**Handling Complex Service Dependencies:** RESTler supports custom annotations to resolve complex dependencies that might not be fully described in the API specification. This functionality is particularly useful in large, production-level cloud services that may require specific request sequences.

**Search Strategies:** RESTler employs different search strategies to more efficiently explore the sequences of requests. These strategies include a breadth-first search (BFS) [8] approach, a faster variant (BFS-Fast) that aims to cover all request types with fewer sequences [9], and a random walk strategy [10] to explore deeper, less common service states. These strategies allow RESTler to scale its exploration based on the complexity of the service being tested.

The empirical evaluation of RESTler shows that it is particularly useful in cloud environments, where services expose a wide range of functionalities through REST APIs. In addition, RESTler supports production-level services with complex API interactions, making it a valuable tool for identifying security vulnerabilities and ensuring service robustness.

## 2.2. Literature on Failure Reproduction

Fault reproduction is fundamental for software testing, especially in complex systems where reproducing a reported failure can aid in debugging and providing patches. Many approaches and tools have been proposed to address this challenge, ranging from test case generation to more sophisticated methods like fuzzing and symbolic execution [11, 12].

A well-known approach in this area is automated test case generation, which aims to reproduce faults by leveraging a set of inputs to reach erroneous states. Tools like AFL (American Fuzzy Lop)[1] [13] and its improvements [14, 15, 16] use genetic algorithms to mutate inputs and observe how programs respond, aiming to crash the system or expose vulnerabilities.

Symbolic execution [17] is another technique used to discover faults by systematically exploring the execution paths of a program. Tools such as KLEE [18] and SAGE [19] employ symbolic execution to generate inputs that explore different paths in the program. STAR [20] uses backward symbolic execution to reproduce a failure. Starting at the code location specified at the top of the stack trace, STAR symbolically executes the application *backward* until reaching the entry point of the topmost function in the trace. Upon reaching that point, STAR attempts to find and jump to the call-site of this function. Although generally effective at reproducing failures, STAR, like other symbolic execution tools, suffers from path explosion, especially in large systems with complex branching. Furthermore, symbolic execution tools generally require the source code of the application under test to be fully instrumented so that the tool can construct precise path conditions.

Search-Based Software Engineering (SBSE) has also made significant contributions to automated test case generation, with the aim to optimize test suites for various objectives, such as coverage, fault detection, or failure reproduction [21, 22, 23]. Tools like EvoSuite [24] generate entire test suites for Java applications by employing genetic algorithms to maximize coverage criteria. Although EvoSuite excels in generating comprehensive test suites, its primary goal is often focused on achieving high coverage rather than reproducing specific failures. Some extensions and adaptations of search-based approaches [25] have shown promising results in tailoring test case generation to reproduce specific behaviors. However, these are often limited by challenges such as stateful dependencies or complex environmental setups.

Recently, fuzzing techniques have evolved to address some of these limitations. Stateful fuzzing tools like RESTler [7] focus on testing APIs by understanding dependencies between multiple API requests. This approach is particularly valuable when bugs occur due to interactions between different components or states. Similarly, Corradini et al. [26] presented Restats, a test coverage tool for REST APIs that supports eight state-of-the-art test coverage metrics in a black-box manner. While RESTler and Restats excel in finding bugs in API-driven cloud services, their focus is on discovering bugs rather than reproducing already-reported failures.

[1] https://lcamtuf.coredump.cx/afl/

A different research angle in the domain of fault reproduction is represented by test case minimization, which plays a key role in reducing the size of test cases that can recreate a fault [27]. Techniques such as delta debugging [28] or hierarchical delta debugging [29] systematically reduce the input size of test cases while preserving the failure-inducing behavior. These approaches are widely adopted in reproducing simpler bugs but may struggle with stateful or environment-dependent failures.

The concept of replay-based fault reproduction has also been explored, where systems such as ReCrash [30] or Execution Reconstruction [31] capture and replay execution traces leading to the fault. This method guarantees faithful reproduction but often incurs high overhead and may be limited to specific types of bugs.

In conclusion, while many tools and techniques exist to either discover or reproduce faults, there remains a gap in tools that automatically generate test cases that faithfully reproduce reported failures in diverse, stateful systems. Our tool builds on these prior works by focusing specifically on this gap, aiming to reduce the manual effort involved in reproducing complex bugs in real-world environments.

## 3. Architecture

An overview of FERRARI is shown in Figure 1. Specifically, the computation begins with three input files, each generated through the analysis of a REST API. The first input, i.e., the method trace, is created by a trace agent[2] that captures all methods called during the execution of the API. The trace agent monitors the API calls and logs all method invocations leading up to the failure, providing a complete view of the interactions between the client and the REST API.

The second input is the stack trace, which represents the exception thrown during a failure. The stack trace captures information about the type and cause of the error. Alongside it, the method trace logs the sequence of method calls that were executed prior to the failure, giving detailed insights into the program's execution flow that led to the error.

The last input is the REST API definition file (referred to as the OpenAPI specification) that describes how to access a cloud service through its REST API, including what requests the service can handle, what responses may be received, and the response format.

To generate meaningful test cases, we use FERRARI with these three inputs. FERRARI is built on top of RESTler and uses it to systematically explore the API space by generating and executing test cases with varying input sequences. These test cases are designed to simulate conditions similar to the original failure (reported in the stack trace input), potentially uncovering new bugs related to the error. After executing each test case, FERRARI allows us to compare the stack trace given as input and the stack traces generated to identify potential new bugs. Specifically, we implemented a similarity score mechanism to compare the sequence of method invocations with the input stack trace—more details on the computation are reported later in Section 4. This similarity score quantifies how closely the behavior observed in the test case matches the conditions of the initial failure. A high similarity score suggests that the test case has likely reproduced the failure, giving the developer a starting point for further investigation.

[2] https://github.com/attilapiros/trace-agent

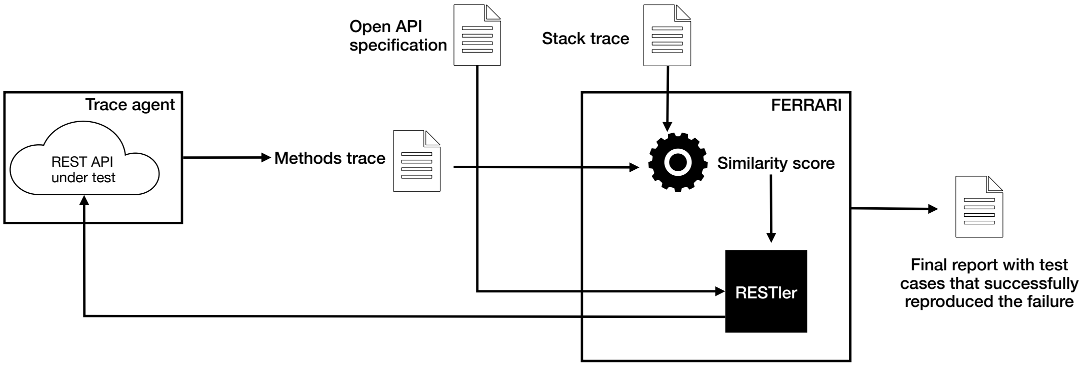

**Figure 1:** Overview of FERRARI architecture. The Figure shows an overview of the entire process, starting from the REST API that a developer aims to investigate.

Finally, FERRARI generates a final report for the developer, highlighting which test case successfully reproduced the error. This report includes the details of the API request that led to the error being reproduced, enabling the developer to examine the inputs that triggered the failure and identify the bugs.

## 4. Running Example

As an example of how FERRARI reproduces failures, we demonstrate how it recreates a stack trace in the open-source Petstore REST API.[3] This API consists of 20 HTTP endpoints for interacting with an eponymous pet store by allowing users to post, update, delete, or fetch information about customers, orders, and registered pets. An OpenAPI specification for this API has also been made available.[4]

The API itself is implemented as a monolithic Java application. We have introduced a synthetic error into the application, with the aim of having FERRARI actively reproduce this failure. Listing 1 depicts the Java handler for `POST /pet/` requests, through which users can save a new pet in the application. This handler accepts a JSON representation of a `Pet` model as input, featuring, e.g., a name and an id of the pet. We have added a check on the posted pet's id (lines 15 – 18) so that, if the id equals a specific, arbitrary value, an uncaught arithmetic error is thrown (line 3), resulting in a stack trace being produced. For the sake of the example, the handler calls two additional methods `petIdIsSmallerThan50` and `petIdIsSmallerThan25` that check whether the id falls within a specific range before raising the error.

```
1  protected void petIdIsSmallerThan25(Long petId) {
2      if (petId == 10)
3          1 / 0; // INJECTED ERROR
4  }
5  protected void petIdIsSmallerThan50(Long petId) {
6      if (petId < 25)
```

[3] https://petstore.swagger.io/

[4] https://github.com/swagger-api/swagger-petstore/commits/master/src/main/resources/openapi.yaml

```
7              petIdIsSmallerThan25(petId);
8          else
9              petIdIsSGreaterThan25(petId);
10     }
11     // Request handler for POST /pet/ requests
12     public ResponseContext addPet(RequestContext req, Pet pet) {
13         [...]
14         PetData.addPet(pet);
15         // START INJECTED ERROR
16         if (pet.getId() < 50)
17             petIdIsSmallerThan50(pet.getId());
18         // END INJECTED ERROR
19         [...]
20     }
```

Listing 1: Synthetically introduced error in the `addPet` HTTP handler for a `POST /pet/` request.

### 4.1. Setting up Failure Reproduction

Listing 2 depicts parts of the stack trace that is produced when the arithmetic error at line 3 in Listing 1 is reached.

In order to reproduce this stack trace, a trace agent (cf. Section 3) must be attached to the Petstore application when the latter is launched, as the agent performs the necessary instrumentation to enable logging method invocations and uncaught exceptions. The trace agent is furthermore configured to recognize the request handlers for each of the 20 HTTP endpoints listed in Petstore's OpenAPI specification. This enables the agent to record the names of all Java methods that were invoked directly or indirectly as a result of an incoming request. To filter out irrelevant method invocations, the agent only records the invocations of methods defined in packages of the application itself. Invocations to third-party library or framework methods or methods defined in the standard Java libraries are hence not recorded. If further investigation into the third-party dependency is required, developers can reconfigure the trace agent to enable the logging of external method calls selectively.

Each incoming request results in the creation of a new method trace that lists the methods that were called while processing this request. Furthermore, if at any point an uncaught exception is thrown, the agent will also ensure that the stack trace for this exception is recorded in a new error log, separate from the recorded method traces.

Once the application has been set up, FERRARI can be configured to reproduce, e.g., the stack trace depicted in Listing 2. At first, RESTler sends several requests to each of the 20 HTTP endpoints listed in Petstore's OpenAPI specification. After each request that it sends to the API, FERRARI reads the method trace that was produced as a result of the request and compares this to the user-specified stack trace.

### 4.2. Computing a Similarity Score

We hypothesize that, the larger the overlap between the method trace of a request and the stack trace that should be reproduced, the higher the likelihood that this request can be tweaked

```
java.lang.ArithmeticException: / by zero
    at io.swagger.petstore.controller.PetController.
petIdIsSmallerThan25$original$uYjjOSWW(PetController.java:167)
    [...]
    at io.swagger.petstore.controller.PetController.
petIdIsSmallerThan50$original$uYjjOSWW(PetController.java:178)
    [...]
    at
        io.swagger.petstore.controller.PetController.addPet(PetController.java)
    [...]
    at org.eclipse.jetty.util.thread.QueuedThreadPool$Runner.run(
        QueuedThreadPool.java:1034)
    at java.base/java.lang.Thread.run(Thread.java:842)
```

**Listing 2:** An example of an error log that is produced when the synthetic error in the POST /pet/ handler is reached.

further in order to exactly reproduce the failure. The overlap between both traces is quantified in the form of a *similarity score*, which expresses how many method invocations from the method trace also appear, in their correct order, in the stack trace.

For instance, suppose that RESTler sends a request to the POST /pet/ endpoint, with a JSON representation of a Pet model with an id of 0 as input. Once this request has been received and processed by the application, the trace agent creates the method trace depicted in Listing 3 and communicates it to FERRARI. At this point, FERRARI compares the method trace to the stack trace and finds that the three method invocations highlighted in blue correspond with the three method invocations highlighted in the user-specified stack trace (Listing 2). These invocations, in turn, correspond to the method definitions highlighted in Listing 1.

When comparing both traces, FERRARI finds that these three methods overlap and computes a similarity score of 3. In contrast, if RESTler sends a POST /pet/ request with a pet id of 100, the similarity score would be only 1, as only the io.swagger.petstore.controller.PetController.addPet method would appear in both traces. If RESTler sends a request to one of the 19 other endpoints, the similarity score might be 0, as there could be *no* overlap between both traces. Requests that result in a similarity score of 0 are discarded at this point, as they provide no relevant information for reproducing the failure. By filtering out such requests early, FERRARI focuses only on requests with meaningful overlap, thereby reducing unnecessary computational overhead.

The similarity score hence indicates how "close" a request came to reaching the specified failure. However, the similarity score cannot be computed before sending a request, as it requires the generated method trace produced by the trace agent during runtime. This approach ensures that the score reflects the dynamic execution behavior of the system.

```
io.swagger.petstore.model.Pet.setId
io.swagger.petstore.model.Pet.setName
[...]
io.swagger.petstore.data.PetData.addPet
io.swagger.petstore.controller.PetController.addPet
[...]
```

```
7  io.swagger.petstore.controller.PetController.petIdIsSmallerThan50
8  io.swagger.petstore.controller.PetController.petIdIsSmallerThan25
```

Listing 3: An example of a method trace for an invocation of the `POST /pet/` handler.

Note that a method trace generally includes method invocations that are not listed in the stack trace. For example, the `PetData.addPet` method is called at line 14 in Listing 1 and appears in the method trace at line 4. Since the function frame for this method invocation was no longer active when the arithmetic error was reached, it does not appear in the stack trace. Hence, the similarity scoring algorithm must take into account that method invocations that appear right after another in the stack trace do not necessarily appear consecutively in the method trace. Although our FERRARI prototype currently, by default, does not intervene in the generation and sending of requests, we aim to modify RESTler's fuzzing algorithm to prioritize sending requests that have previously resulted in a high similarity score.

### 4.3. Comparing Stack Traces

In parallel to parsing method traces and comparing them to the user-specified stack trace, FERRARI also checks whether the previous request created a new error log, for example, because an uncaught exception was raised. If so, FERRARI parses this error log as well and compares it with the specified stack trace. In contrast with the comparison between method traces and the stack trace, this comparison considers both methods defined in third-party libraries and frameworks, as well as methods defined in the application's packages.

However, it should be noted that reaching the same error twice in separate execution scenarios does not necessarily result in producing two stack traces that are exactly identical. For one, there may be some non-determinism in parts of the application. As an example, the Petstore application employs a third-party library to route HTTP requests to their corresponding Java handlers. Non-determinism in the routing may result in different library methods being called before the same failure is eventually reached. Furthermore, if the application relies on run-time reflection, the fully-qualified names of several methods in the stack trace may be partially auto-generated, and may hence change whenever the application is restarted. This is demonstrated in Listing 2, where several method calls feature the auto-generated `uYjjOSWW` identifier.

The stack trace comparison mechanism takes both issues into account by parsing the trace to remove auto-generated identifiers and by performing a *weak* matching of traces. Auto-generated identifiers are removed by simply discarding all characters that appear after a "$" in the method's fully-qualified name. Weak matching considers two stack traces to be identical if both the error message (e.g., `java.lang.ArithmeticException: / by zero`) and the top five method calls[5] are the same. FERRARI saves all requests that result in the creation of an error log that weakly matches the specified stack trace. Once RESTler has finished fuzzing, FERRARI outputs a report that lists these requests so that users can examine with which inputs the failure can be reproduced.

[5] The restriction to the top five method calls strikes a balance between computational efficiency and ensuring that enough contextual information is preserved to meaningfully compare traces. However, this threshold is configurable and can be adjusted by the user depending on the needs of the analysis.

## 5. Evaluation and Discussion

We have performed a limited evaluation of FERRARI on the Petstore REST API to gauge how effective our similarity scoring mechanism is at steering the tester to send those requests through which it can reproduce a specified failure. As mentioned in Section 2, RESTLER, by default, employs a sophisticated strategy for iteratively modifying requests and arranging them into various sequences to thoroughly test a REST API. We compare the default RESTLER with FERRARI that discards requests that result in a low similarity score rather than iterating over these requests. The discarding of requests with a similarity score of 0 occurs immediately after computing the score once the method trace for the request has been compared to the user-specified stack trace. By filtering out these requests early, FERRARI avoids wasting resources on paths that are unlikely to reproduce the failure.

Table 1 compares how often RESTLER reproduces the arithmetic error depicted in Listing 2 with and without discarding low-scoring requests. In both cases, we let our FERRARI prototype check for error logs that are created while fuzzing the API and compare these with the specified stack trace.

| | Requests sent | # error logs produced | # failure reproductions |
|---|---|---|---|
| RESTLER | 5,809 | 3,346 | 405 |
| FERRARI | 737 | 476 | 405 |

**Table 1**
Comparison of the effectiveness at reproducing the arithmetic error in Listing 2 with and without discarding of low-scoring requests.

If we do not discard any requests, RESTLER sends a total of 5,809 requests before terminating the fuzzing, and 3,346 error logs are produced. The vast majority of these error logs are the result of RESTLER intentionally generating malformed requests, resulting in uncaught exceptions being thrown by the application. Such errors are unrelated to the arithmetic error that we wish to reproduce. RESTLER also manages to reproduce the arithmetic error in analysis 405 times.

However, if we let FERRARI discard any requests that result in a similarity score of 0, then RESTLER only sends 737 requests. These requests result in 476 error logs, 405 of which match the stack trace with the arithmetic error that we want to reproduce. While the number of failure reproductions (405) remains constant, FERRARI achieves this with significantly fewer requests sent and error logs produced. This highlights the efficiency of the filtering mechanism: it enables FERRARI to focus on promising requests, minimizing redundant error logs and resource usage while consistently reproducing the desired failure.

We can conclude that our adaptation was, hence, comparatively more efficient at reproducing the arithmetic failure in the Petstore application than the baseline RESTLER fuzzing strategy. By discarding low-similarity requests, FERRARI significantly reduces the number of requests sent and error logs analyzed, which in turn reduces the overall time required to reproduce the failure. While this evaluation did not focus on execution time explicitly, initial observations suggest that FERRARI can reproduce a failure more quickly compared to RESTLER.

While our preliminary evaluation focuses on demonstrating FERRARI's efficiency compared to RESTLer, we believe its design and initial results highlight its potential to achieve the broader goals of helping developers identify defects more efficiently and improving the precision of failure reproduction. In particular, by filtering out requests with a low similarity score and focusing on requests that are more likely to reproduce the target failure, FERRARI minimizes the noise generated during debugging. This reduction in irrelevant error logs and test cases significantly narrows the search space for developers, allowing them to pinpoint the conditions leading to a failure more quickly. For instance, developers can immediately focus on the 476 error logs relevant to the target failure instead of also having to analyze those that are not relevant to the error they want to reproduce and that are contained in the 3,346 logs produced by RESTLer. This prioritization lays the foundation for reducing manual debugging effort and accelerates the identification of defects.

Moreover, the similarity score mechanism in FERRARI ensures that test cases are tailored to reproduce specific stack traces accurately. This mechanism not only increases the likelihood of reproducing a failure but also provides developers with high-confidence test cases that closely match the original conditions. This precision allows developers to investigate the failure in a controlled environment, iterating over a smaller, more focused set of scenarios.

## 6. Conclusion and Future Agenda

In this paper, we presented FERRARI, an extension to RESTLer with the aim of reproducing failures in cloud services by focusing on REST API stack traces. Our tool integrates the analysis of specific stack traces with the capabilities of RESTLer to generate test cases, helping developers identify bugs more efficiently. By comparing the stack traces, we automate part of the debugging process, reducing manual effort and enhancing accuracy in locating the cause of failures.

The preliminary results and evaluation from our experiments on the Petstore API demonstrate the potential of our approach. By computing a similarity score, we were able to reproduce failures more efficiently and with fewer requests. Furthermore, the integration of fuzzing and stack trace analysis allows us to efficiently filter out irrelevant error logs and focus on those that are more likely to reproduce the targeted failure.

Our work provides several opportunities for future research. One of the main directions involves extending the applicability of our tool to microservice architectures, where distributed and stateful interactions between components can lead to even more challenging debugging scenarios. Another area of improvement is refining the similarity metrics used to compare stack traces, potentially incorporating more sophisticated techniques to improve the accuracy of the comparison. In that sense, following the work of Fortz et al. [32], we plan to use deep learning models to learn which messages or combinations are important for reproducing failure.

Finally, we plan to conduct a large-scale empirical study across various real-world cloud services and systems to provide more insights into the effectiveness of our tool in different environments. This study would allow us to assess the robustness and generalizability of our tool and optimize it for practical use in large-scale cloud systems.

## Acknowledgments

The authors are partially funded by the FWO SBO BaseCamp Zero project (Code: S000323N).

## References

[1] A. Neumann, N. Laranjeiro, J. Bernardino, An analysis of public rest web service apis, IEEE Transactions on Services Computing 14 (2018) 957–970.

[2] R. A. DeMillo, H. Pan, E. H. Spafford, Failure and fault analysis for software debugging, in: Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97), IEEE, 1997, pp. 515–521.

[3] J. A. Whittaker, What is software testing? and why is it so hard?, IEEE software 17 (2000) 70–79.

[4] G. Canfora, M. Di Penta, Service-oriented architectures testing: A survey, in: International Summer School on Software Engineering, Springer, 2006, pp. 78–105.

[5] M. Bozkurt, M. Harman, Y. Hassoun, Testing and verification in service-oriented architecture: a survey, Software Testing, Verification and Reliability 23 (2013) 261–313.

[6] M. Burger, A. Zeller, Minimizing reproduction of software failures, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, 2011, pp. 221–231.

[7] V. Atlidakis, P. Godefroid, M. Polishchuk, Restler: Stateful rest api fuzzing, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 748–758.

[8] R. Zhou, E. A. Hansen, Breadth-first heuristic search, Artificial Intelligence 170 (2006) 385–408.

[9] M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches, Software testing, verification and reliability 22 (2012) 297–312.

[10] Z. B. Zabinsky, et al., Random search algorithms, Department of Industrial and Systems Engineering, University of Washington, USA (2009).

[11] X. Zhu, S. Wen, S. Camtepe, Y. Xiang, Fuzzing: a survey for roadmap, ACM Computing Surveys (CSUR) 54 (2022) 1–36.

[12] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Computing Surveys (CSUR) 51 (2018) 1–39.

[13] A. Fioraldi, A. Mantovani, D. Maier, D. Balzarotti, Dissecting american fuzzy lop: A fuzzbench evaluation, ACM Trans. Softw. Eng. Methodol. 32 (2023). URL: https://doi.org/10.1145/3580596. doi:10.1145/3580596.

[14] C. Wang, S. Kang, Adfl: an improved algorithm for american fuzzy lop in fuzz testing, in: Cloud Computing and Security: 4th International Conference, ICCCS 2018, Haikou, China, June 8-10, 2018, Revised Selected Papers, Part V 4, Springer, 2018, pp. 27–36.

[15] C. Lemieux, K. Sen, Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage, in: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, 2018, pp. 475–485.

[16] S. Wei, S. Yang, P. Zou, Improvement of afl's seed deterministic mutation algorithm,

in: International Conference on Emerging Networking Architecture and Technologies, Springer, 2022, pp. 347–357.

[17] J. C. King, Symbolic execution and program testing, Communications of the ACM 19 (1976) 385–394.

[18] C. Cadar, D. Dunbar, D. R. Engler, et al., Klee: unassisted and automatic generation of high-coverage tests for complex systems programs., in: OSDI, volume 8, 2008, pp. 209–224.

[19] P. Godefroid, M. Y. Levin, D. A. Molnar, et al., Automated whitebox fuzz testing., in: NDSS, volume 8, 2008, pp. 151–166.

[20] N. Chen, S. Kim, STAR: stack trace based automatic crash reproduction via symbolic execution, IEEE Transactions on Software Engingeering 41 (2015) 198–220. URL: https://doi.org/10.1109/TSE.2014.2363469. doi:10.1109/TSE.2014.2363469.

[21] M. Harman, B. F. Jones, Search-based software engineering, Information and software Technology 43 (2001) 833–839.

[22] A. Arcuri, G. Fraser, On parameter tuning in search based software engineering, in: Search Based Software Engineering: Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings 3, Springer, 2011, pp. 33–47.

[23] M. Harman, S. A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Computing Surveys (CSUR) 45 (2012) 1–61.

[24] G. Fraser, A. Arcuri, Evosuite: automatic test suite generation for object-oriented software, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, pp. 416–419.

[25] A. Panichella, F. M. Kifetew, P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, IEEE Transactions on Software Engineering 44 (2017) 122–158.

[26] D. Corradini, A. Zampieri, M. Pasqua, M. Ceccato, Restats: A test coverage tool for restful apis, in: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2021, pp. 594–598.

[27] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Software testing, verification and reliability 22 (2012) 67–120.

[28] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Transactions on Software Engineering 28 (2002) 183–200.

[29] G. Misherghi, Z. Su, Hdd: hierarchical delta debugging, in: Proceedings of the 28th international conference on Software engineering, 2006, pp. 142–151.

[30] S. Artzi, S. Kim, M. D. Ernst, Recrash: Making software failures reproducible by preserving object states, in: ECOOP 2008–Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings 22, Springer, 2008, pp. 542–565.

[31] G. Zuo, J. Ma, A. Quinn, P. Bhatotia, P. Fonseca, B. Kasikci, Execution reconstruction: Harnessing failure reoccurrences for failure reproduction, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021, pp. 1155–1170.

[32] S. Fortz, P. Temple, X. Devroey, P. Heymans, G. Perrouin, Varyminions: leveraging rnns to identify variants in variability-intensive systems' logs, Empirical Software Engineering 29 (2024) 99.