Onweer: Automated Resilience Testing through Fuzzing

Gilles Coremans
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
gilles.coremans@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

***Abstract*—Micro-service architectures remain a popular choice for cloud applications. However, micro-service applications are subject to faults such as dropped requests between services which are difficult to catch during development in testing environments. Automated testing tools have been proposed to test the resilience of micro-service applications against faults in testing, but these require manually specifying execution scenarios. In this paper, we propose an approach to resilience testing which integrates fault injection into the fuzzing process. In this way, system executions and faults can be simultaneously explored, allowing fully automated resilience testing. We describe our approach as well as a prototype implementation called ONWEER, and evaluate its performance on a standard micro-service benchmark application.**

***Index Terms*—Quality analysis and evaluation, Reliability, Testing tools, Service-Oriented Architecture.**

I. INTRODUCTION

Micro-service architectures remain one of the more popular strategies for organizing large software systems. Splitting up a large application into many smaller micro-services offers many advantages [1]: services can be developed, deployed, and scaled independently from each other. The ability to develop and deploy micro-services independently allows different teams to more easily work on different parts of the application, increasing development efficiency. Furthermore, the ease with which micro-services can be scaled up or down means that micro-service applications can adapt to changing loads without long-term over-provisioning.

However, splitting components into independent services introduces additional complexities when these services need to communicate with each other [2]. Distributed communication is subject to *faults* such as requests being delayed, duplicated, or lost. More severe faults are also possible, such as services crashing and losing their state. As micro-service applications are often deployed across several physical machines, such faults are unavoidable. Thus, to ensure correct operation of micro-service applications, they must be made *resilient* to faults.

Generally, this is accomplished by carefully written fault handling logic which takes any faults into account and prevents a fault from causing a failure of the system. For example, if a connection error occurs, this fault could be handled by retrying the request until it succeeds. While such fault handling logic is essential to the functioning of micro-service applications, it is also difficult to write correctly. Faults can occur at any point in the execution, and can have a variety of effects. Furthermore, due to the large amount of edge cases introduced and the possibility of faults interacting across multiple different micro-services, it is difficult to determine whether an application is resilient or not. Finally, micro-service applications are usually tested in environments that differ from the production environment, often running on a single machine. In these environments it is rare for faults to occur, and thus the fault handling logic is difficult to test.

All of these factors mean that micro-service applications often have bugs, called *resilience defects*, when exposed to faults. For example, a badly written fault handler which retries a request may inadvertently cause that request to be processed twice. This can cause issues ranging from chat requests being sent twice to payments being processed twice.

To find and resolve resilience defects, *chaos engineering* [3] was developed. Chaos engineering involves intentionally introducing faults into a production system and observing whether metrics such as request latency or number of users logged in are negatively affected. If they are, one can conclude that the system is not resilient to the faults, and the issue can be located and solved. Organisations using chaos engineering often deploy their applications at a very large scale, and thus faults can be injected affecting only a small part of the system or even only a small section of users. This minimizes the impact on the perceived reliability of the system.

However, chaos engineering is rather costly to deploy: it is generally performed by dedicated site reliability engineers, requires detailed system metrics, and requires organisational buy-in [4]. Furthermore, testing in production remains risky even if it is controlled. For example, one can imagine that a medical company may not be able to introduce faults in any part of critical production systems.

For this reason, there has been research interest in *resilience testing* which can be performed in testing or staging environments and which is more automated, requiring less specialized engineers. Gremlin [5] is one of the earliest tools in the field, and relies on *recipes* which allow developers to specify which faults to inject when. This was later followed by tools such as Filibuster [6] and Chaokka [7], which use the structure of the micro-service system to automatically explore all possible faults efficiently.

However, which faults can occur and their effect on the system depends on the system's execution path; not every request will produce the same sequence of requests sent internally between services. Existing resilience testing tools rely on manually specified execution scenarios. This means that it is possible for these tools to miss resilience defects if the execution scenarios have insufficient coverage of the system, reducing the automation offered. Thus, an ideal resilience testing tool should not only automatically explore all possible faults, but also explore executions along which these faults can occur.

Automatic exploration of the execution space of a program is a field which has seen a large amount of research, often focused on automatic bug finding. One of the most successful and popular approaches is *fuzzing*, a technique whereby random inputs are generated for the system under test [8]. This technique is fully automated, and requires relatively little knowledge of the system under test, and with a sufficiently large time budget can explore many different program behaviors. By providing enough random inputs, many different program behaviors can be found and thus the program can be tested. To increase the efficiency of this approach, fuzzers often rely on *feedback* from the system under test, such as the response given to an input (called *blackbox* fuzzing) or code coverage (called *greybox* fuzzing). Instead of generating purely random inputs, feedback is used to determine which inputs are interesting, and these inputs are then incrementally changed by *mutators* to create new potentially interesting inputs.

We propose combining resilience testing with feedback-driven fuzzing to create a more automated, more effective resilience tester. One could consider the faults to be injected as part of the input used by a fuzzer, and thus similarly mutate them according to the feedback obtained from the system. This solves two problems at once. First, execution paths are automatically found by the fuzzer, freeing developers from the need to provide execution scenarios. Second, the fault space can be efficiently searched by mutating interesting requests to inject faults during their execution. Whether these requests with fault injection are interesting can then be determined using feedback from the system, and if so, further mutations to the input or faults can be performed.

This paper makes the following contributions:

- We present a novel approach which reformulates resilience testing as a fuzzing problem, uses coverage feedback to determine which sequences of faults are interesting to inject, and traces the system under test to find fine-grained fault injection points.
- We present a prototype implementation of this approach, called ONWEER.
- We evaluate ONWEER on a standard micro-service benchmark application, showing it can increase coverage, cover fault handlers and find resilience defects.
- We provide the source code of our implementation, the benchmark systems used for evaluation and the evaluation data in an appendix [9].

```
1  @PostMapping("/ping")
2  public ResponseEntity<Ping> pingPost(
3              RestTemplate rest, @RequestBody Increment i) {
4      long ctr;
5      ResponseEntity<Pong> ponge;
6      try {
7          ponge = rest.postForEntity("http://pong/pong",
8                                         i, Pong.class);
9          ctr = counter.addAndGet(i.increment());
10     } catch(HttpClientErrorException e) {
11         // Handle 400 response from pong
12         logger.warn(e);
13         return ResponseEntity.badRequest().build();
14     } catch(RestClientException e) {
15         // Retry on connection error
16         logger.error(e);
17         ponge = rest.postForEntity("http://pong/pong",
18                                        i, Pong.class);
19         ctr = counter.addAndGet(i.increment());
20     }
21     Pong pong = ponge.getBody();
22     return ResponseEntity.ok(new Ping(ctr, pong.id()));
23 }
```

Listing 1: The implementation of the `ping` example service.

```
@PostMapping("/pong")
public ResponseEntity<Pong> pongPost(
            RestTemplate rest, @RequestBody Increment i) {
    long inc = i.increment();
    if(inc >= 0 && inc < 10000) {
        long ctr = counter.addAndGet(inc);
        return ResponseEntity.ok().body(new Pong(ctr));
    } else {
        // 400 status code if out of range
        return ResponseEntity.badRequest().build();
    }
}
```

Listing 2: The implementation of the `pong` example service.

These contributions show that our approach enables more automated and efficient resilience testers than currently exists in the state of the art.

II. MOTIVATING EXAMPLE

We will illustrate the need for the integration of fuzzing and fault injection in an example. Consider a simple micro-service system consisting of the two services depicted in Listing 1 and Listing 2. It consists of two services, `ping` and `pong`, which must keep a counter in sync. The counter may be incremented by any amount by sending requests to `ping`, which must then communicate this increment to `pong`. However, if the increment is out of range (not between 0 and 10000), `pong` will return a 400 status code, indicating an invalid argument, which must be handled by `ping`.

Implementing these micro-services is tricky, because faults such as connection errors may occur in the request sent from `ping` to `pong`. Thus, `ping` implements a basic retry mechanism.

However, thoroughly testing this retry mechanism is quite tricky. If a developer tests this application without fault injection, they can cover lines 4-9 and 21-22 of `ping`, where nothing goes wrong, and lines 10-13, where `pong` returns a 400 status code, but they will not be able to cover lines 14-20, which are only executed when a connection error occurs.

```
1:  population $\leftarrow$ initialize_population()
2:  **loop**
3:      (seed_sequence, seed_traces) $\leftarrow$ select(population)
4:      candidate_sequence $\leftarrow$
                mutate(seed_sequence, seed_traces)
5:
6:      pre_coverage $\leftarrow$ get_coverage()
7:      responses $\leftarrow \varnothing$
8:      traces $\leftarrow \varnothing$
9:      **for all** request, faults, links $\in$ candidate_sequence **do**
10:         **for all** index, from, to $\in$ links **do**
11:             request[to] $\leftarrow$ responses[index][from]
12:         **end for**
13:         install_faults(faults)
14:         responses $\leftarrow$ responses $\cup$ send_request(request)
15:         traces $\leftarrow$ traces $\cup$ get_trace()
16:     **end for**
17:
18:     post_coverage $\leftarrow$ get_coverage()
19:     **if** post_coverage $>$ pre_coverage **then**
20:         population $\leftarrow$
                    population $\cup$ (candidate_sequence, traces)
21:     **end if**
22: **end loop**
```

Listing 3: An overview of the extended fuzzing algorithm that underlies our approach.

Thus, fault injection is needed to fully test this system. If the developer uses fault injection in a test where the input is in range and injects a connection error fault in the REST request on line 7, they will be able to cover lines 14-20 of `ping` and achieve full coverage. However, they will still have missed two resilience defects present in this system:

- If the developer injects a connection error but the input is out of range, the application will crash with a 500 status code and they find the first *resilience defect* of this application. The first REST request on line 7 is covered by a `try` block to catch the exception thrown when `pong` returns a 400 status code, but the second REST request on line 17 is not. Thus, if the request is out of range and a fault is injected, the exception indicating a 400 status code will propagate and cause a crash.
- If the developer injects a connection error on line 7 and then also injects a connection error on line 17, the application will also crash. As before, the retry block does not have a `try` block to catch the exception thrown when a connection error occurs, and thus the application is only resilient to one fault, not two consecutive faults.

These resilience defects illustrate that in order to thoroughly test micro-service applications, it is necessary to inject multiple faults and to inject faults under a variety of inputs.

III. Overview of the Approach

Our approach to integrating fuzzing and fault injection involves making two significant modifications to the basic fuzzing loop: stateful fuzzing and fault injection. This results in the high-level algorithm shown in Listing 3 which we will describe in detail in the following sections.

A. Stateful Fuzzing

Fuzzing in our approach serves to explore potential executions of the micro-service system such that faults can be injected during a variety of executions of the system. Most micro-service systems are stateful, regardless of whether they use a REST, gRPC or different API. Typically, users of the API must first send a request to create or fetch a resource, and then send subsequent requests to manipulate this resource. Our approach assumes that a schema describing all operations in the API and their arguments is available, such as OpenAPI [10] for REST or protocol buffer definitions for gRPC.

Thus, our approach must be capable of generating stateful sequences of requests to the system. For this, we use an approach inspired by RESTler [11], but implemented as mutators in our fuzzing architecture. The population is seeded with request sequences containing a single request for every operation in the schema. New requests are then added to the sequences by mutators, which may add, remove, duplicate or swap any requests in the sequence.

To make these sequences stateful, it must be possible for values from responses to previous requests to be used as parameters in future requests. For example, upon creation of a new resource, an identifier for that resource is usually returned. This identifier must be passed to future operations on that resource, but if all arguments are randomly generated by a fuzzer this is very unlikely to happen.

Thus, our approach stores the responses of every request and uses the concept of *links* between requests in a sequence to utilize these responses in future requests. Links are introduced to sequences by mutators. A link has three properties:

- The “index” property indicates which response in the sequence to take a value from.
- The “from” property indicates which key in the response body to take the value of.
- The “to” property indicates which argument in the request to replace with the value taken from the response with the given index at the given key.

Lines 10-12 of Listing 3 show how links replace values in the request with values from a previous response.

For example, consider a sequence for a REST API consisting of `GET /products, POST /cart/add?pid=5`. The second request has a link with an index of 0, a “from” property of `[0, id]` and a “to” property of `[pid]`. The first request, `GET /products`, returns a JSON object of the form `[{"id": 103}, {"id": 104}]`. Then, before the request `POST /cart/add?pid=5` can be executed, its link will take the response at index 0, and look at the property `id` of the first element in the list. It will then place this value, 103, in the property `pid` of the request, resulting in the request `POST /cart/add?pid=103`, having replaced the randomly generated value of `5` with `103`.

B. Tracing & Fault Injection

Code locations where a fault can be injected are called *fault injection points*. Because faults can only be injected in code which is executed, efficient fault injection requires knowledge of which fault injection points are executed for any given request. Thus, after the execution of every request, our approach collects a *trace* which is stored with the request, as shown in Listing 3 on line 15. The trace records every potential fault injection point and how often it was executed during that request.

Faults are added to requests by a mutator which takes the trace of a request and adds a fault at one of the points listed in the trace. These faults are then installed in the system under test just before the request is executed, as shown in Listing 3 on line 14. The faults will be actually injected when the corresponding fault injection point is executed. Under our fault model, a fault can be injected multiple times at a fault injection point. For example, if a fault is injected three times at some fault injection point, then that fault will be injected the first three times that that point is executed.

Our approach intentionally does not prescribe a highly specific fault model and leaves this to the implementation. The exact behavior of a fault does not matter to our approach, and it even admits combining different kinds of faults. It also allows for very fine-grained faults by using fine-grained traces, but if more coarse-grained fault injection is desired this is also possible within our framework.

Note that a trace is collected for every request, including requests which already inject faults. Thus, it is possible for a request that injects faults to have a trace which may expose new fault injection points. By mutating this request, more faults can be injected and it is possible to incrementally create requests with multiple faults.

C. Greybox Feedback

As seen in Listing 3 on lines 6 and 18-21, our approach uses greybox coverage feedback to determine whether an input is interesting. Many stateful fuzzers only use blackbox feedback methods such as responses from the system and coverage of endpoints, so as to avoid needing to instrument the system under test. However, our approach requires collecting traces to enable fault injection, which likely already requires instrumentation for most systems, making the extra cost of gathering coverage fairly small. Thus, our approach use greybox coverage feedback to increase the efficiency of our approach.

Coverage feedback allows our approach to more easily distinguish which inputs cover new code and thus are interesting, giving a wider variety of inputs to inject faults in. It is also useful to determine which sets of faults are interesting to inject. If a sequence containing faults increases coverage, it is likely because it covers a fault handler. Thus, by adding this sequence to the population, it becomes available for further mutation and our fuzzing approach can attempt to find bugs in this fault handler, either by injecting more faults or by modifying the parameters of the request.

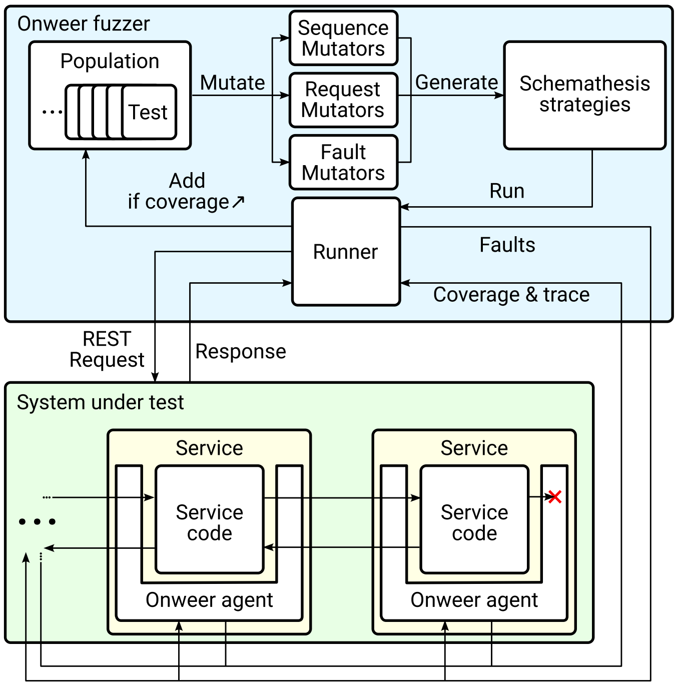


Fig. 1. A diagram showing how ONWEER, the ONWEER agents and the system under test interact.

IV. PROTOTYPE IMPLEMENTATION: ONWEER

We have implemented our approach in ONWEER, a prototype resilience tester. The source code of ONWEER is available in our online appendix [9]. Figure 1 shows the high-level architecture of our tool, which closely corresponds to the algorithm in Listing 3. As seen in Figure 1, ONWEER consists of two components: the ONWEER fuzzer, described in Section IV-A and Section IV-B, and the ONWEER agents, described in Section IV-C.

A. Generating REST Requests

As REST APIs are the most common choice for implementing micro-service systems, we have chosen to build ONWEER as a REST fuzzer. Thus, our tool must be able to generate individual REST requests to build sequences out of. As mentioned in Section III-A, our tool relies on an OpenAPI [10] schema for the system under test to know which operations and parameters are available, and which responses to expect.

To generate individual REST requests from this schema, ONWEER uses the Schemathesis [12] library. This library parses the schema into a Hypothesis [13] data generation strategy per endpoint, which we can use to generate REST requests for each endpoint.

These Hypothesis strategies can be seen as pure functions which take as only input a bitstream and output a REST request. Mutations of the input bitstream result in structural mutations of the output REST request, a property which Hypothesis uses to implement shrinking in property-based testing [14]. However, ONWEER uses this to implement a parametric fuzzer [15]. In this way, it is not necessary to implement complex structural mutators for REST requests, only some simple bitstream mutators.

B. Generating REST Sequences

ONWEER builds and executes sequences of REST requests as described in Section III-A. However, some details are important to note.

ONWEER uses HTTP status codes as an oracle for the successful execution of a request. If a request in a sequence receives a `400 Bad Request` response, the sequence is considered invalid and discarded. If a request receives a `500` status code, the sequence is considered an error and recorded, as these errors are what ONWEER is designed to find.

In our implementation, links can replace any part of a request object with any part of a previous response object. This means that it is possible for links to create requests which do not conform to the OpenAPI schema. This is unavoidable, as OpenAPI schemas do not contain sufficient information to determine the relations between the response of one endpoint and the parameters of another. Some REST fuzzers such as RESTler [11] use heuristics to determine which operations may be safely linked, but this runs the risk of missing valid links and thus valid sequences. However, these invalid requests do not hinder the fuzzing process. Passing a nonconforming parameter is likely to simply result in a `400` Bad Request status code, which allows us to safely conclude that this request is invalid and stop executing the sequence.

C. Tracing, Fault Injection & Coverage

ONWEER implements the tracing, fault injection and coverage features described in Section III-B and Section III-C using ONWEER *agents*. These agents are responsible for gathering coverage information, tracing requests and injecting faults. An agent should be attached to every micro-service in the system under test. Agents communicate with the ONWEER fuzzer through a simple REST API, allowing multiple implementations of the ONWEER agent such that our tool may be used with polyglot micro-service systems.

In terms of the fault injection model, ONWEER implements the abstract fault model described in Section III-B. Tracing of fault injection points and the actual behavior of fault injection is left entirely up to the agents, with fault injection points being arbitrary strings which ONWEER does not interpret in any way. Furthermore, every fault injection point will only be used with the agent where it originates. This means that it is possible to use multiple different agents implementing different fault injection models together.

Currently, we have implemented an ONWEER agent for Java which supports Spring Boot applications and the example in Section II, as well as the TeaStore system used in our evaluation. We discuss the fault model used for TeaStore in Section V-A2.

V. EVALUATION

In this section we answer the following research questions about the performance of ONWEER and its underlying concepts:

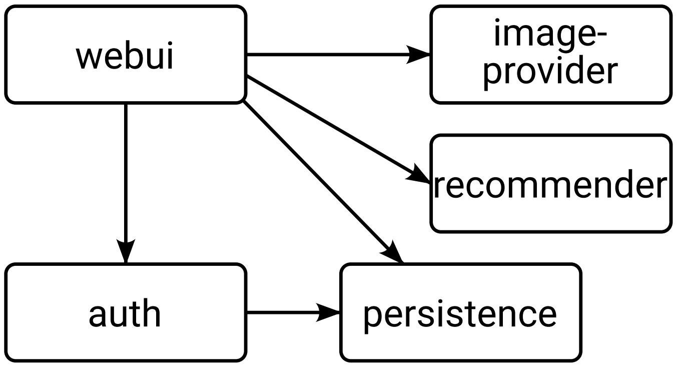


Fig. 2. Architecture of TeaStore. Arrows represent which services send requests to other services. The `registry` service is only used at startup and thus not shown. Requests sent only during startup are also not shown.

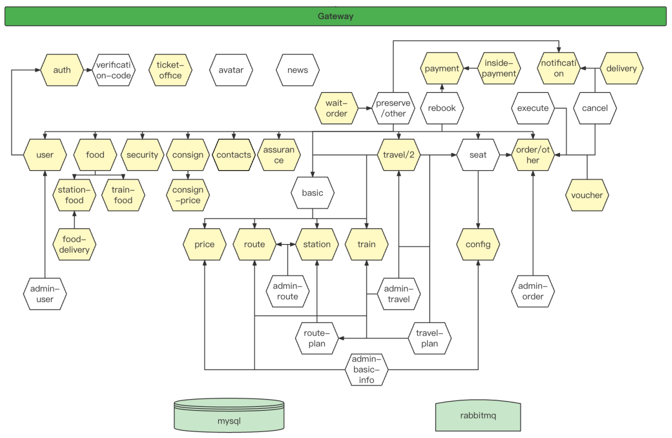


Fig. 3.

RQ1: Can integrating fault injection into a REST fuzzer increase code coverage?

RQ2: Can integrating fault injection into a REST fuzzer cover otherwise unreachable fault handlers?

RQ3: Can integrating fault injection into a REST fuzzer find resilience defects that a regular fuzzer cannot?

RQ1 aims to provide insights into whether fault injection makes the fuzzing process more or less effective, that is, whether it is a tradeoff or a simple advantage. **RQ2** investigates whether ONWEER can increase the coverage of specific pieces of code which are difficult to reach with ordinary fuzzers. **RQ3** aims to determine whether fault injection can reveal bugs which may be missed without fault injection.

We answer these questions by discussing experimental results of running ONWEER on the TeaStore benchmark application [16], comparing results with fault injection disabled and results with fault injection enabled.

A. Experimental Setup

1) Benchmark Systems: As noted by Meiklejohn et al. [6], there are few open source micro-service applications available, due to the fact that micro-services are generally adopted within large organisations and make less sense in the context of open source software. This presents a problem for evaluating ONWEER, as there is no solid corpus of applications to evaluate the approach. Because our approach involves integrating resilience testing and fuzzing, evaluating it requires a micro-service application which is interesting both from a resilience testing and from a fuzzing perspective. Thus, benchmark systems must have several interacting services to allow fault injection as well as a REST API with several operations, each having some parameters which can be fuzzed.

In their paper introducing Filibuster, Meiklejohn et al. [6] presented a corpus of micro-service applications intended for evaluating resilience testers. However, this corpus is highly specialized to resilience testers, and as such its applications generally only include a single operation with very limited parameters. This makes this corpus unsuitable for evaluating our approach, which relies on the integration of fuzzing and fault injection.

The TeaStore micro-service benchmark system [16] was developed because few viable micro-service benchmark systems exist. It has enjoyed success as an academic benchmark and is used as a benchmark system in various academic papers [17]–[19] This system comprises a webshop application and has a variety of operations, with multiple parameters for each operation and is thus suitable for evaluating the fuzzing side of our approach. Furthermore, the structure of TeaStore, shown in Figure 2, comprises five micro-services. Importantly the links between services are not trivial; the `auth` service is contacted by `webui` and itself contacts `persistence`, which is also contacted by `webui`. Thus, there are sufficient interacting fault injection points available to be a useful benchmark system.

To prepare TeaStore for automated resilience testing, we wrote a REST interface to TeaStore's application code and an OpenAPI schema for this interface, comprising 11 operations. This is needed as TeaStore provides only a web interface and no RESTful API. We made this REST interface follow the same flow and send the same requests as using the web interface, so that the system remains representative. These changes allow ONWEER to work with TeaStore. Finally, the default configuration of TeaStore specifies that requests are not retried, and thus the system is not resilient. In order to actually enable the fault handling logic present in TeaStore, we changed this configuration so that requests are retried once. This modified version of TeaStore is available in our online appendix [9].

2) Experimental Setup: We configured and deployed in their standard configuration, except for the changes noted above, running each service in a separate Docker container. Each service was instrumented by an ONWEER agent for the purposes of collecting coverage information, tracing, and fault injection.

TeaStore uses a custom communication middleware for all requests. Thus, the trace points used for our evaluation are calls into this communication middleware, allowing for very fine-grained tracing. This communication middleware uses Netflix Ribbon [20] which uses Jakarta EE [21] to send REST requests. Faults are injected by throwing a Jakarta

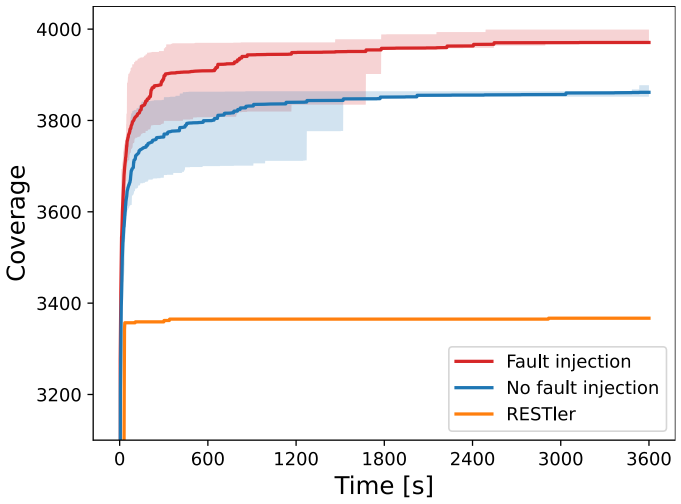


Fig. 4. Comparison of coverage over time between ONWEER with fault injection, ONWEER without fault injection and RESTler . The solid line represents the mean coverage of all fuzzing runs, while bottom and top of the shaded area represent the 10th and 90th percentile of fuzzing runs.

`ProcessingException` in this low-level Jakarta request processing code. This is the same exception thrown when a connection fails, and thus this fault model accurately simulates real-world behavior on unreliable connections.

We configured ONWEER to run for 1 hour on TeaStore and , with an initial population of 3 single-request sequences for each operation in the OpenAPI schema. With these settings, we ran ONWEER 45 times with and 45 times without fault injection enabled on TeaStore, . We restarted the applications between every run, removing any state added by the previous run, and only started sending requests after the application had fully started.

During these runs, we collected detailed information about the fuzzing process, including every sequence in the population and every error returned by even if it did not increase coverage. For each of these population members and errors we also collected the precise requests sent, responses received, which mutators constructed them, and detailed per-class coverage information.

This allows us to answer each research question effectively: we can use coverage achieved over time to answer **RQ1** by comparing how which coverage level is achieved and how quickly it is achieved with and without fault injection. The errors recorded by ONWEER can be examined to answer **RQ2** by looking at the number of unique errors discovered with fault injection which are not found without fault injection. Finally, per-class coverage information lets us compare performance at a more granular level and thus see whether classes responsible for fault handling have increased coverage with fault injection, answering **RQ3**.

B. Results

The results of this experiment are presented in Figure 4, , Table I and Table II. The full dataset generated by this experiment is available in our online appendix [9].

TABLE I
AVERAGES OF COVERAGE, POPULATION SIZE, ITERATIONS AND ERRORS FOUND DURING THE EXPERIMENT.

		With fault injection				Without fault injection				Difference
Application	Statistic	Mean	Stdev	Min	Max	Mean	Stdev	Min	Max	Mean
TeaStore	Coverage	3970.73	25.86	3818	4001	3861.62	24.86	3712	3894	109.11
	Population size	53.49	2.60	48	60	52.04	2.71	46	58	1.44
	Errors found	12404.33	1873.97	8804	17909	531.7	327.40	73	1623	11873.0
	Iterations	126344.13	12819.76	97954	150830	121871.86	9968.62	95520	141121	4472.27
	Coverage									
	Population size									
	Errors found									
	Iterations									

TABLE II
PER-SERVICE, PER-CLASS DIFFERENCE IN COVERAGE BETWEEN ONWEER RUNS WITH AND WITHOUT FAULT INJECTION ENABLED.

		Mean coverage		
Service	Class name	With faults	Without faults	Difference
webui	LoadBalancerCommand	48	31	17
auth	LoadBalancerCommand	48	31	17
webui	DefaultLoadBalancerRetryHandler	9	5	4
auth	DefaultLoadBalancerRetryHandler	9	5	4
webui	ServiceLoadBalancer	42	39	3
auth	ServiceLoadBalancer	42	39	3
auth	TrackingFilter	8	5	3
webui	ProcessingException	2	1	1
webui	LoadBalancerContext	27	26	1
auth	LoadBalancerContext	27	26	1
webui	RestHelpers	19	18	1

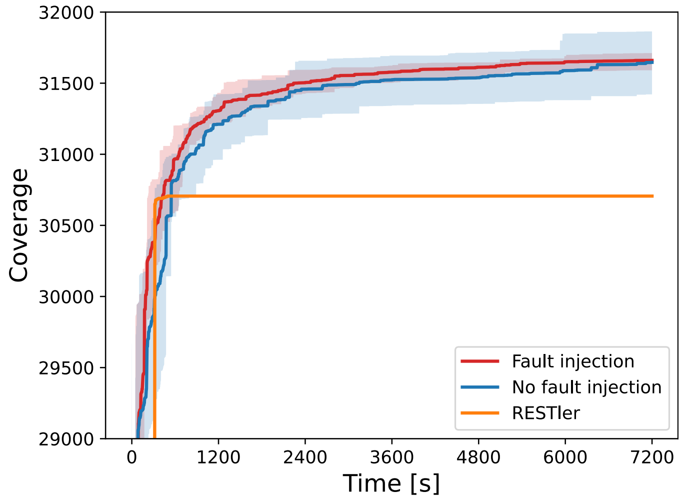


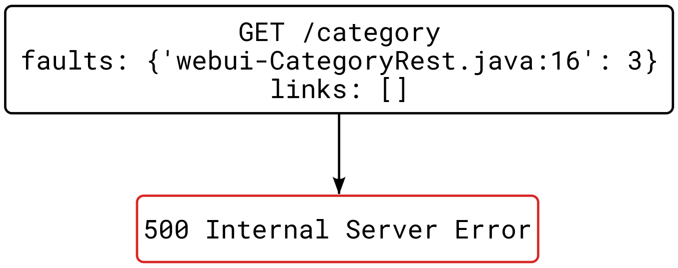
Fig. 5.

Looking at Table I we see the mean coverage over all runs is 109 lines higher when using fault injection, and the maximum coverage achieved in all runs without fault injection is less than the mean coverage achieved over all runs with fault injection.

Furthermore, Figure 4 shows that after half an hour of fuzzing, most runs are close to saturation. At this point, the majority of runs with fault injection achieve more coverage than the majority of runs without fault injection.

Table II shows the per-class difference between the mean coverage achieved with and without fault injection in TeaStore. As coverage is collected on a per-service basis, communication middleware classes which are shared between services such as `ServiceLoadBalancer` will have their coverage measured separately for every service. Thus, we have a separate coverage score for each class on every service. The table is pruned to only show coverage differences with a magnitude greater than 1, as it is expected that there will be some noise between any two sets of runs.

We can see that most instances of increased coverage are clearly related to fault handling. For example, on both the `webui` and `auth` services we see a coverage increase in `LoadBalancerCommand`, `ServiceLoadBalancer` and `LoadBalancerContext`. These are all classes involved in retrying failed connections by retrying the request on the same or other service, as well as paths for when these retries fail. We see a coverage increase in these classes on the `webui` and `auth` services, as these are the services that send requests to other services and thus can have faults injected. The classes which are not involved in this logic, `GenericExceptionMapper` and `ProcessingException` have higher coverage since the errors discovered by ONWEER cause more exceptions to be thrown. `ProcessingException` specifically is thrown

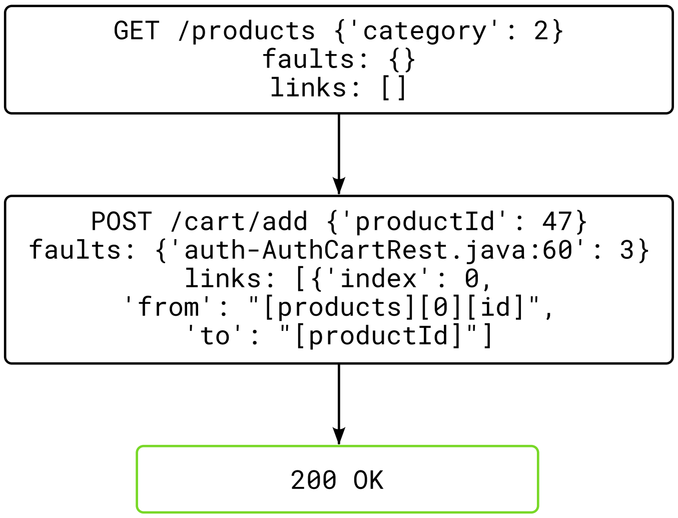


GET /category
faults: {'webui-CategoryRest.java:16': 3}
links: []
500 Internal Server Error

Fig. 6. A short sequence with three faults injected at the same fault injection point, resulting in the application crashing and returning a `500 Internal Server Error` status code. Nodes denote requests, with method, endpoint, faults, and links, while edges denote the sequence order. The final node represents the response of the last request in the sequence.

TABLE III

Unique errors	TeaStore	TrainTicket
Without faults	42	36
With faults	34	42
Only with faults	25	7
Manually filtered	16	3



GET /products {'category': 2}
faults: {}
links: []
POST /cart/add {'productId': 47}
faults: {'auth-AuthCartRest.java:60': 3}
links: [{'index': 0,
'from': "[products][0][id]",
'to': "[productId]"]
200 OK

Fig. 7. A sequence with three faults injected in the `auth` service, showing that this service is resilient to faults. The semantics of this figure is the same as Figure 6.

directly by the ONWEER instrumentation, so its increased coverage is expected. This lets us confirm that fault handlers which are not covered by regular fuzzers can be covered with fault injection, answering **RQ2** affirmatively.

To answer **RQ3**, we can look at the errors recorded during execution. Table I shows that many more errors are found with fault injection than without fault injection, but this is the total number of requests resulting in an error, not the number of unique bugs found.

First, some errors are found without fault injection.

Second, we look at the errors found when using fault injection. All these errors have the same basic pattern, shown in Figure 6. As seen in Figure 6, the error involves three faults being injected at the same fault injection point, in this case three faults in service `webui`, in file `CategoryRest.java`, on line 16. These three consecutive errors when trying to send a request cause TeaStore's retry mechanism to fail and throw an exception, which is not handled by the application code and thus causes a `500 Internal Server Error` response.

Taking all of this in account, we can conclude that ONWEER can find resilience defects which cannot be found using a fuzzer without fault injection, and affirmatively answer **RQ3**.

C. Discussion

One aspect of Table I which has not yet been discussed is the row recording population size. Because fault injection increases coverage of this system, and the per-class results in Table II indicate coverage did not decrease for any class under fault injection, we might also expect the population size to be somewhat larger to account for new interesting inputs using fault injection. However, the results are not clear. While the average population size of all runs with fault injection is 1.44 sequences larger than the average population size of all runs without fault injection, the difference is relatively small compared to the population size. Furthermore, the difference is also rather small compared to the standard deviation, and the minimum/maximum ranges of the population sizes overlap significantly. Thus, we would likely need a larger sample size and more detailed statistical tests to determine whether there is a significant difference in population size.

Another interesting result concerns the faults injected in the `auth` service. As seen in Table II, the coverage of the TeaStore middleware increased under fault injection just as it did, and thus we expect to find sequences in the population which inject faults in the `auth` service to achieve that increase. These sequences are indeed present, and an example can be seen in Figure 7. Notably, this sequence does *not* result in a 500 status code and is thus not considered an error by ONWEER. When faults are injected in the communication middleware of the `auth` service, the retry mechanism fails and an exception is thrown, in the same way as in the `webui` service. The `auth` service catches and correctly handles this exception, returning a `504 Gateway Timeout` status code to the `webui` service, which this service handles, producing a normal `200 OK` result.

However, despite returning a `200 OK` status code, the specified product is not added to the cart. Thus, this is arguably a resilience defect, as the requested operation is not completed, yet no indication of this is given to the user. This is a limitation of ONWEER, as it uses the response status code as an oracle. While a 500 status code reliably indicates a bug in the system under test, response with a 200 status code can still be incorrect.

VI. FUTURE WORK & LIMITATIONS

A. Nondeterminism & State

Our approach assumes that a sequence of REST requests has a deterministic execution path and relies only on state the sequence itself creates. However, in the context of micro-service systems these assumptions do not always hold. First, many interesting sequences of REST requests do change the state of the system by adding or deleting resources, and these changes persist to future request sequences. This cannot be avoided in general because restarting the system between every sequence would be very slow, as many micro-service systems take several minutes to start. We mitigate this issue with links between requests filling in parameters with earlier responses. This allows a request sequence to create a resource and then operate on it in a deterministic way, regardless of the name or ID that resource is nondeterministically assigned. However, we do not enforce this and thus request sequences may still rely on state not created earlier in the sequence.

Second, even in the absence of explicit user-visible state, micro-service systems may still be nondeterministic. For example, message queues or caching mechanisms may be used to increase the performance or reliability of the system but can cause the same request sequence to take different paths.

This nondeterminism causes issues for our prototype implementation. If a sequence that relies on previously existing state is added to the population, it may fail if that state is mutated and thus not yield further interesting sequences. Furthermore, spurious increases in coverage due to service replication may result in uninteresting inputs being added to the population. This nondeterminism also presents a challenge for our fault injection strategy, as it relies on the fault injection points executed by a sequence always remaining the same.

Mitigating the effects of nondeterminism is an important avenue for future research to increase the applicability of fuzzing-based resilience testers. Potential approaches could be taken from the existing literature on flaky tests, which offer some methods of mitigating nondeterminism in testing. Another approach could be to render the system deterministic by integrating sources of nondeterminism in the fuzzing framework, which is a common technique in fuzzing and concolic testing.

B. Service Replication

Micro-service applications often deploy multiple copies of many services, meaning that requests can be handled by any of these services. This is another source of nondeterminism, but requires special consideration as it is very common and exposes limitations of our coverage feedback approach.

Considering the coverage of each service separately is insufficient to handle this case, as it is clear that rerunning the same request could increase its coverage simply by it being routed to another service. However, it is also not clear if it would be correct to consider a line of code fully covered if any service has executed it. Improper coordination between replicated services can result in resilience defects, and thus requires testing, which requires some feedback as to how thoroughly these services have been tested.

Thus, the development of new coverage or feedback metrics tuned to the needs of micro-service systems is an important avenue of future research. Such a metric could not only improve resilience testing, but also manually written tests and chaos engineering.

C. Additional Fault Types

In our evaluation, we only injected connection error faults. However, many other kinds of faults can occur in micro-service systems, such as services failing or being delayed. In order to better test more complicated micro-service systems, it is necessary to integrate these faults into resilience testing.

Future work should explore which faults are most interesting to inject and how they can be injected. As discussed in Section III-B and Section IV-C, the architecture of our approach as well as the architecture of ONWEER enable such experimentation. By design, faults are opaque to the fuzzer; a new fault type can be added simply by writing instrumentation that traces its fault injection points and injects it.

VII. Related Work

A. Resilience Testing

Since the advent of chaos engineering, there has been interest in automating resilience testing approaches in testing environments, yielding a variety of different resilience testing approaches in the literature.

FATE and DESTINI [22] are notable for being one of the earliest implementations of the resilience testing concept. FATE identifies failure injection points, which are any I/O points in the target application, and systematically explores every possible fault combination in order to test the resilience of the system. Information from this resilience testing is then used in DESTINI to specify correct recovery behaviors for the system. FATE systematically explores the fault space but does not explore execution space, and is specialized towards "infrastructure" distributed applications such as Zookeeper rather than micro-service applications.

Gremlin [5] is another early resilience testing tool which allows developers to write *recipes* which specify a pattern of faults to apply to a system. For example, a developer could write a recipe which simulates a service being overloaded, and the system can then be tested with this fault recipe to determine its behavior under overloaded services. However, this approach offers neither automated fault space exploration nor execution space exploration.

Chaokka [7] is a resilience tester specialized to Akka actor systems. It operates by injecting all possible faults in a system, and then using delta debugging to determine the minimal set of faults that will still cause a test failure. While Chaokka explores the fault space using delta debugging, it assumes a fixed test case under which delta debugging is performed.

Filibuster [6] makes the requirement for execution scenarios implicit by being implemented as a library which can enrich end-to-end tests with resilience information, letting developers specify the desired behavior when certain faults occur. It efficiently searches the fault space by assuming that micro-services encapsulate services they depend on, meaning that downstream faults are only visible in the form of a specific status code returned by the upstream service. However, it relies on existing end-to-end tests written by developers of the system and does not automatically explore the fault space.

Fault Injection Analytics [23], unlike most other resilience testing tools, injects faults into a system under some synthetic load rather than executing under specific end-to-end tests. The system is traced during the fault injection run, and the traces are compared to traces without fault injection using a machine learning model to find *anomalies* in the fault-injected traces, which are identified as resilience defects. This approach of looking for anomalies during a load test of the system differs substantially from our approach, which attempts to find specific combinations of requests and faults that reveal a resilience defect.

B. REST Fuzzing

As REST APIs are commonly used in industry, and thus there is also a large amount of interest in automated testing of these interfaces, leading to a wide variety of REST API fuzzers [24]. While none of these approaches incorporate fault injection, it is interesting to consider the differences in approach between ONWEER and common REST fuzzers.

RESTler [11] stateful REST API fuzzer. It relies on a Swagger/OpenAPI specification of the API, and analyzes this specification to determine the dependencies and outputs of each operation. Using this information, it then constructs sequences so that each request's dependencies can be fulfilled by earlier requests in the sequence to efficiently test the API. RESTLer systematically extends sequences with new requests until it cannot find any new sequences, attempting to exhaustively explore the sequence space, while our approach instead uses mutators to construct sequences during the fuzzing process. Furthermore, our approach does not use information from the OpenAPI schema to determine which responses can be used as future parameters. Such technical optimisation can be considered in future instantiations of the approach.

EvoMaster [25] is REST testing tool based on the approach of the EvoSuite search-based testing tool. Because it is based on EvoSuite's approach, EvoMaster instruments the system under test to gain feedback and speed up the testing process. This instrumentation is more heavy-weight than our approach, as it uses detailed metrics to enable search-based testing. It builds sequences out of templates of common REST API patterns rather than analyzing dependencies between operations, whereas our approach constructs sequences during the fuzzing process. EvoMaster has also been extended to RPC frameworks and used in industry [26].

RestTestGen [27] is a modular framework for implementing REST fuzzers. It aims to making it simpler to try new REST testing approaches by implementing many components which are commonly used in REST testing tools, such as an OpenAPI specification parser, mutation operators, etc. Like RESTler, RestTestGen attempts to deduce relations between requests from the provided schema, which our approach does not do. Our focus has been on increasing coverage through fault injection.

Schemathesis [12] is a property-based tester for REST APIs, which we used to implement ONWEER. It uses an OpenAPI specification to determine the arguments and expected responses for an API, and then generates a large number of example requests for each operation in order to test if the responses listed in the API are correct and exhaustive. While it is effective at generating REST requests [24], it does not support stateful fuzzing, and thus ONWEER implements its own stateful fuzzing layer.

VIII. CONCLUSION

In this paper, we presented and described a novel resilience testing approach which integrates greybox fuzzing and fault injection. This combination of techniques allows more efficient and more thorough resilience testing of micro-service systems, as faults are injected during fuzzing instead of during developer-specified test cases. We presented our prototype implementation of this approach, called ONWEER, and evaluated this tool on the TeaStore benchmark system. We determined that our tool can increase coverage on benchmark systems, cover otherwise unreachable fault handlers, and identify resilience defects in micro-service systems.

REFERENCES

[1] V. Velepucha and P. Flores, "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges," *IEEE Access*, vol. 11, pp. 88 339–88 358, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10220070

[2] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

[3] M. A. Chang, B. Tschaen, T. Benson, and L. Vanbever, "Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, S. Uhlig, O. Maennel, B. Karp, and J. Padhye, Eds. ACM, 2015, pp. 371–372.

[4] H. Tucker, L. Hochstein, N. Jones, A. Basiri, and C. Rosenthal, "The Business Case for Chaos Engineering," *IEEE Cloud Computing*, vol. 5, no. 3, pp. 45–54, May 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8383672

[5] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic Resilience Testing of Microservices," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2016, pp. 57–66. [Online]. Available: https://ieeexplore.ieee.org/document/7536505

[6] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, and R. Padhye, "Service-Level Fault Injection Testing," in *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, C. Curino, G. Koutrika, and R. Netravali, Eds. ACM, 2021, pp. 388–402.

[7] J. De Bleser, D. Di Nucci, and C. De Roover, "A Delta-Debugging Approach to Assessing the Resilience of Actor Programs through Run-time Test Perturbations," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, ser. AST '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 21–30. [Online]. Available: https://dl.acm.org/doi/10.1145/3387903.3389303

[8] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.

[9] Anonymous, "Experiment data for Onweer, including source code of Onweer, source code of modified TeaStore, and experiment output JSON files," Mar. 2025. [Online]. Available: https://zenodo.org/records/15425114

[10] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, and U. Sarid, "OpenAPI Specification," Feb. 2021. [Online]. Available: https://spec.openapis.org/oas/v3.1.0.html

[11] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 748–758.

[12] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web API schemas," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 345–346. [Online]. Available: https://doi.org/10.1145/3510454.3528637

[13] "Hypothesis," 2024. [Online]. Available: https://hypothesis.works/

[14] D. MacIver, "How Hypothesis Works," Dec. 2016. [Online]. Available: https://hypothesis.works/articles/how-hypothesis-works/

[15] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 329–340. [Online]. Available: https://dl.acm.org/doi/10.1145/3293882.3330576

[16] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sep. 2018, pp. 223–236. [Online]. Available: https://ieeexplore.ieee.org/document/8526888

[17] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn, "Microservices: A Performance Tester's Dream or Nightmare?" in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 138–149. [Online]. Available: https://dl.acm.org/doi/10.1145/3358960.3379124

[18] L. Liao, J. Chen, H. Li, Y. Zeng, W. Shang, C. Sporea, A. Toma, and S. Sajedi, "Locating Performance Regression Root Causes in the Field Operations of Web-Based Systems: An Experience Report," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4986–5006, Dec. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9629300

[19] J. Keim, S. Schulz, D. Fuchß, C. Kocher, J. Speit, and A. Koziolek, "Trace Link Recovery for Software Architecture Documentation," in *Software Architecture*, S. Biffl, E. Navarro, W. Löwe, M. Sirjani, R. Mirandola, and D. Weyns, Eds. Cham: Springer International Publishing, 2021, pp. 101–116.

[20] "Netflix/ribbon," Netflix, Inc., Mar. 2025. [Online]. Available: https://github.com/Netflix/ribbon

[21] "Jakarta EE." [Online]. Available: https://jakarta.ee/

[22] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "{FATE} and {DESTINI}: A Framework for Cloud Recovery Testing," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing

[23] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Fault Injection Analytics: A Novel Approach to Discover Failure Modes in Cloud-Computing Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1476–1491, May 2022.

[24] M. Zhang and A. Arcuri, "Open Problems in Fuzzing RESTful APIs: A Comparison of Tools," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 144:1–144:45, Sep. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3597205

[25] A. Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, pp. 3:1–3:37, Jan. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3293455

[26] M. Zhang, A. Arcuri, Y. Li, Y. Liu, and K. Xue, "White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, pp. 122:1–122:38, Jul. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3585009

[27] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "RestTestGen: An Extensible Framework for Automated Black-box Testing of RESTful APIs," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct. 2022, pp. 504–508. [Online]. Available: https://ieeexplore.ieee.org/document/9978261