# Automated Grey-box Testing of Microservice Architectures

L. Giamattei*, A. Guerriero, R. Pietrantuono, and S. Russo
DIETI, Universit degli Studi di Napoli Federico II, Napoli, Italy
{luca.giamattei, antonio.guerriero, roberto.pietrantuono, stefano.russo}@unina.it
*corresponding author

***Abstract*—Microservices Architectures (MSA) have found large adoption in companies delivering online services, often in conjunction with agile development practices. Microservices are distributed, independent and polyglot entities – all features favouring black-box testing. However, for real-scale MSA, a pure black-box strategy may not be able to exercise the system to properly cover the interactions involving internal microservices.**

**We propose a grey-box strategy (MACROHIVE) for automated testing and monitoring of (internal) microservices interactions. It uses combinatorial testing to generate *valid* and *invalid* tests from microservices specification. Tests execution and monitoring are automated by a *service mesh* infrastructure. MACROHIVE runs the tests and traces the interactions among microservices, to report about internal coverage and failing behaviour.**

**MACROHIVE is experimented on *TrainTicket*, an open-source MSA benchmark. It performs comparably to state-of-the-art techniques in terms of edge-level coverage, but exposes internal failures undetected by black-box testing, gives detailed internal coverage information, and requires fewer tests.**

***Keywords*—*Microservices; Grey-box Testing; Functional testing***

## I. INTRODUCTION

Microservice architectures (MSA) are a service-oriented software architectural style where services are loosely coupled, run in their own processes, and interact via lightweight mechanisms [**?**]. These characteristics allow for independent development (by different teams, different programming languages) and deployment. They are usually developed according to lean or agile development practices like DevOps, enabling rapid and frequent software releases (even many per day).

Testing is the common solution to assess the quality of an MSA. In agile contexts, test automation and efficiency are of paramount importance to get quick and continuous feedback about quality. As MSA code is polyglot and distributed across various repositories, black-box testing is usually deemed as the most suitable approach [**?**]. Automatic techniques for specification-based black-box testing of RESTful web-services can be applied for MSA testing, as they can generate test cases from documentation of their microservices interface [**?**], [**?**], [**?**].[1] This practice is adopted in black-box testing of service-oriented architectures for fault detection [**?**], [**?**], as well as to test against requirements while achieving some degree of coverage [**?**], [**?**].

However, the characteristics of real-scale MSA can make black-box testing fall short. When many microservices are involved, with complex inter-dependencies, a black box view gives no information about the internal behaviour (both in terms of achieved internal-microservices coverage and of their failing behaviour). Black-box testing exercises functionalities from an external perspective, with requests directed to *edge microservices*. The output of an edge microservice is usually dependent on the interaction with other *internal* microservices, which can be edge for other functionalities, or inaccessible from the outside. The absence of an internal perspective does not allow a tester to distinguish if a failure observed on a request to a microservice is due to the microservice being faulty or to another, interacting, microservice that propagated its failure to the one under test. Also, internal microservices can be invoked by different edge microservices; if one of them is faulty, several different failures can be observed at edge level, in possibly different microservices. Testing without an internal perspective considers these as independent failures.

This paper presents a grey-box specification-based strategy for automatic tests generation and interactions monitoring. The strategy is supported by a tool, MACROHIVE, deployed as a collection of microservices according to a *service mesh* pattern. This provides observability of internal interactions, which is crucial for microservice testing [**?**]. Applied to the *TrainTicket* benchmark [**?**], MACROHIVE turns out to perform comparably to black-box state-of-the-art techniques in edge-level coverage; it however: *i)* exposes a number of internal failures undetected by black-box testing (distinguishing propagated from masked failures), thus easing the identification of faulty microservices and of failure propagation chains; *ii)* gives details about internal dependencies, errors, and exceptions – of great importance to practitioners [**?**]; *iii)* and requires a lower number of tests. Moreover, being itself a (set of) microservices deployed with the MSA, it does not need to run separate testing sessions for each microservice to test.

In the following: Section **??** describes related work; Section **??** describes the proposed grey-box strategy; Sections **??** and **??** present experimentation and results, respectively; Section **??** discusses threats to validity; Section **??** concludes the paper.

## II. RELATED WORK

Several studies present testing techniques conceived for MSA [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. Long *et al.* [**?**] present a technique for fitness-guided resilience testing, with the goal of finding as many bugs in the fault handling logic as possible in a set amount of time. Heorhiadi *et al.* [**?**] investigate resilience testing too, proposing the *Gremlin* framework for systematically testing the failure-handling capabilities of microservices,

[1]The most notable open format for specifying web services and MSA Application Programming Interfaces (API) is OpenAPI/Swagger [**?**] (https://www.openapis.org).

by injecting faults into inter-service messages.

De Camargo *et al.* [?] propose the *FPTS* framework for automated performance testing; it helps evaluating performance delivered by individual microservices, through annotations to be used within their source code to generate a specification, needed for workload creation. Lei *et al.* [?] propose a method for performance testing with the use of Kubemark.[2]

None of the above studies deals with functional testing. Nonetheless, because of the prominent role of RESTful API in MSA [?], several tools for black box testing of RESTful web services are suitable for MSA testing too. Such tools usually consider as objectives: maximization of API coverage (number of executed methods), maximization of HTTP response codes coverage, automatic fault detection.

Corradini *et al*. have empirically compared them [?]. All tools aim to maximize the coverage of methods specified in the APIs via data and operations dependencies. The comparison is in terms of "robustness", meant as ability to manage real-world systems, and of coverage criteria as defined by Martin-Lopez *et al*. [?]. The comparison highlights the following three main tools: RestTestGen, RESTler and bBOXRT.

RestTestGen [?] is a stateful test generator, that infers data dependencies with an operation dependency graph. It generates nominal and faulty test cases. Input values are generated from a dictionary, from examples in the specification, randomly, or re-using past observed values.

RESTler [?] is a tool for stateful input generation via fuzzing, aiming to find security issues. The authors focus on inferring producer-consumer dependencies among the specified request types, and on analyzing dynamic feedback from responses given by executed tests. Similarly to RestTestGen input values are selected from a user-configurable dictionary, or from previously observed values.

bBOXRT [?] is for robustness testing of REST services. The authors designed a method for injecting faults in requests, attempting to trigger erroneous behaviors. Specification-compliant input values are randomly generated, and then mutated to observe the system behavior under a faulty workload.

A state-of-the-art tool for automated testing of RESTful Web Services is EvoMaster, proposed by A. Arcuri. Initially conceived for white-box testing, EvoMaster has then been extended to support black-box testing [?]. It performs random testing, adding heuristics to maximize the HTTP response code coverage. EvoMaster did not take part in the comparison, as Corradini *et al.* stated that it was not available yet.

Martin-Lopez *et al.* [?] propose RESTest, a black-box tool for automatic fault detection. They use an Inter-parameter Dependency Language (IDL). The results obtained depend on the available information about dependencies; more information improve results, but require testers to specify dependencies in IDL - this is time-consuming, requires a deep knowledge of the system under test, and reduces automation.

[2]Kubemark is a performance testing tool for running Kubernetes experiments on simulated clusters (https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scalability/kubemark-guide.md).

MSA testing can indeed borrow tools conceived for black box testing of RESTful Web Services. This alleviates the burden of manual API testing in service-based systems, which is a common practice in industry [?]. However, it also poses challenges that we aim to address with this work. First, applying the mentioned tools requires to run distinct testing sessions for every edge microservice – a practice that does not scale well with the number of microservices [?]. Second, microservices' interactions can result in complex invocation chains involving internal services in a real scale application; when these are insufficiently covered by a test suite, failures may remain undetected. This may well happen with all described black-box techniques, as they consider coverage metrics only at edge level.

With respect to the above, our contribution is twofold:

- We propose a grey-box testing strategy specific for MSA, which adopts a combinatorial testing generation technique, supported by an automated tool (MacroHive), deployed itself as a collection of microservices and not requiring to run separate testing sessions for each microservice; differently from existing tools, the proposed strategy allows to compute coverage of internal microservices, and it provides insights into the failing behaviour.
- We highlight the benefit of, and need for, a grey-box strategy rather than a black-box one, by experimentally comparing MacroHive with four of the above-mentioned black-box tools. The results highlight the shortcomings of black-box testing due to the impossibility of collecting metrics for MSA internal services, which motivated the proposed MacroHive tool.

## III. Grey-box testing strategy

### A. Overview

The grey-box strategy for testing an MSA, aims to expose and characterize failures[3] and to provide internal coverage information. It focuses on observability, which is important when debugging a distributed system such as an MSA [?]. MSA are usually characterized by:

- *edge* microservices, exposing APIs to external users to access the functionality offered by the systems;
- *internal* microservices, exposing APIs to other microservices to implement complex business functions.

A microservice can be edge for some functions and internal for others. Black-box testing may not be able to allow testers to evaluate the test suite's ability to cover internal interactions. Moreover, they cannot spot when a microservice fails due its own fault or due to the failure of an internal microservice.

MacroHive generates tests starting from the microservices' API, and for every executed test observes the chain of requests among internal microservices. It supports the proposed grey-box testing strategy via automated test suite generation, then execution and monitoring thanks to an infrastructure - designed

[3]In the MSA literature, a failure is considered as a request yielding a *5xx* HTTP response code, indicating an error condition, an unhandled exception, or in general the inability to serve the request [?], [?], [?].

according to the *service mesh pattern* [**?**] - deployed with the MSA under test.

At the end of a session, the following results concerning edge and internal microservices are provided to the tester:

- the set of executed tests with the corresponding outcome;
- the path of requests of each test through the internal microservices;
- a set of metrics at both edge and internal microservices levels (e.g., number of failures, average response time);
- a set of metrics for each level of dependency, namely the depth of a microservice in the requests chain.

With this information, the tester can discriminate different kinds of failures involving internal microservices, such as *masked failures* (corresponding to correct responses from edge microservices, despite failures of internal microservices), and *propagated failures* (incorrect responses of the edge microservices due to failures of internal microservices).

## *B.* MacroHive

MacroHive is conceived to automatically expose both edge and internal failures, so that a tester does not need to manually inspect request paths. This functionality allows catching internal failures, undetectable by black-box strategies. It also allows identifying the true cause of edge-level failures, namely if due to the edge itself or to internal microservices. Since the testing process targets microservices of the same MSA, it is possible to detect common cause failures (e.g., a single faulty microservice that causes failures of other microservices).

Figure **??** shows the MacroHive infrastructure. It has three main components: *uTest*, *uSauron* and *uProxy* (uP). The first is responsible for test cases generation and execution. The other components form a support inter-service communication infrastructure [**?**] to be deployed with the SUT. An MSA is composed of many microservices with independent deployments, often controlled by multi-container management tools such as Docker Compose [**?**], [**?**]. MacroHive automatically manipulates a docker-compose YAML file to add a sidecar proxy to each microservice to test/monitor.

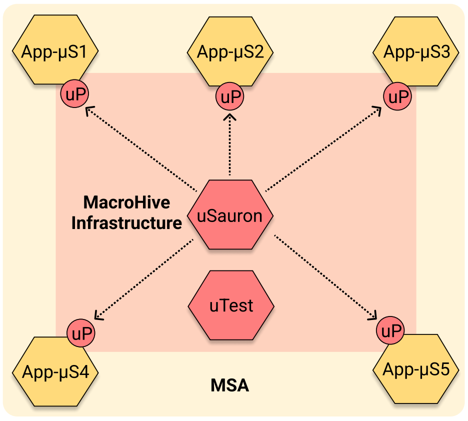


Figure 1: The MacroHive infrastructure

TABLE I: Example of input space partitioning

| Parameter | Type | Input Classes | Category |
|---|---|---|---|
| $p_1$ *(required, in path)* | *string* | $c_{1,1}$: in range | *valid* |
| | | $c_{1,2}$: specified example value(s) | *valid* |
| | | $c_{1,3}$: empty string | *invalid* |
| | | $c_{1,4}$: no string | *invalid* |
| $p_2$ *(required, in body)* | *integer* | $c_{2,1}$: positive value in range | *valid* |
| | | $c_{2,2}$: negative value in range | *valid* |
| | | $c_{2,3}$: alphanumeric string | *invalid* |
| | | $c_{2,4}$: no value | *invalid* |
| $p_3$ *(optional, in body)* | *boolean* | $c_{3,1}$: {true,false} | *valid* |
| | | $c_{3,2}$: no value | *valid* |
| | | $c_{3,3}$: empty string | *invalid* |
| | | $c_{3,4}$: alphanumeric string | *invalid* |

TABLE II: A sample test case specification

| | |
|---|---|
| URI template | http://exampleHost:8080/examplePath/{$c_{1,2}$} |
| HTTP method | POST |
| body template | {$p_2$:{$c_{2,2}$},$p_3$:{$c_{3,1}$}} |
| HTTP status code | 201, 400 |

***uTest***

This service generates and executes a test suite. It adopts a pairwise generation strategy that could help testers to detect multi-factor faults, which are a high percentage in software systems [**?**]. Compared to other state of the art techniques, we expect a combinatorial design to substantially reduce testing cost, while providing good coverage and fault detection ability [**?**]. *uTest* automatically retrieves the specification (in the OpenAPI/Swagger format) of the edge microservices of the MSA under test. The API are parsed to extract an Input Space Model consisting of HTTP methods, URIs and body templates, HTTP status codes and parameters details (type, bounds, default value, etc.); equivalence classes [**?**] are defined for each parameter and then categorized into valid and invalid.[4]

Table **??** shows an example of input space partitioning for a request with three parameters. By selecting two equivalence classes per parameter, *test case specifications* are produced with a pairwise combinatorial strategy: a *2-way* test suite is generated, covering all pairs of parameter classes. Table **??** shows a sample test case specification: a test case generated from this specification shall have for $p_1$ a value chosen from class $c_{1,2}$ (the *example* value); for $p_2$ a value from class $c_{2,2}$ (negative value in range), and for $p_3$ the value *true* or *false*.

We call *valid* test cases those containing parameter values all belonging to valid input classes; *invalid* test cases those containing at least a parameter value belonging to an invalid class. To generate a nominal test suite (composed of only *valid* test cases), only valid classes per parameter are selected (when available, examples valid and default values are preferred), otherwise *valid* and *invalid* classes per parameter are chosen to generate a mixed test suite (e.g., for robustness testing).

The generated tests are executed by sending HTTP requests. MacroHive allows generating requests also in case of authen-

[4]A class is valid if it contains only input parameter values which do comply to the microservice specification, and invalid if it contains only values that do not.

tication, by specifying credentials or tokens in the configuration file. The test outcome is automatically determined by evaluating the received HTTP status code.

***uSauron and uProxy***

These two components constitute a service mesh infrastructure to trace service dependencies and log request-response couples during a testing session. Although many monitoring tools are available in the literature (e.g., Prometheus[5], Jaeger[6], etc.), we preferred to build our infrastructure in favor of automation and flexibility with minimum instrumentation.

*uProxy* (*uP*) is deployed alongside each microservice to test/monitor, complying with the sidecar pattern [?], [?]. Each proxy performs two tasks:

- acting as a reverse proxy for the coupled microservice;
- sending to *uSauron* an information packet whenever it collects a request-response couple.

Different threads run these tasks to minimize communication delay. The information packet is composed of: request/response URL, request/response body, HTTP response code, response time, sender/receiver address.

*uSauron* is a microservice responsible for the collection of information provided by proxies. In particular, it aims to log proxies packets and compute fine-grained metrics (e.g., coverage, dependencies) for each test. For this purpose, *uSauron* runs a distributed algorithm during a testing session to link collected information to executed tests.

***Test execution algorithm***

The tests execution algorithm run by MacroHive (Figure **??**) is realized by *uTest* (the test executor), *uSauron* (the collector), and *uProxies* (the probes). The example in Figure **??** shows a test involving microservices *uS4* (edge) and *uS2*, *uS3* (internal); it entails the following messages: a *start recording* message (number 1) is sent by *uTest* to *uSauron*; it notifies the intent to run test $t$ and that every subsequent message received by uSauron needs to be linked to $t$. Then, *uTest* actually starts the test $t$, sending an HTTP request to the *uP* proxy coupled with the edge microservice (message number 2). The involved proxies intercept the request-response couples with the edge microservice (2,7) and the internal interactions (3,6 and 4,5). For every intercepted request/response, the proxies send information packets to *uSauron* (messages 7.1, 6.1, and 5.1), which links them to test $t$. When *uTest* receives the response for $t$ (message number 7), it sends a *stop record* message to *uSauron* (message number 8). On receipt, *uSauron* stops the packets recording and saves the collected records.

This algorithm is executed for every test in a testing session. The way it is designed, the monitoring infrastructure can capture any concurrent calls of internal microservices made within the same test execution. At the end of a session, *uSauron* outputs a set of statistics.

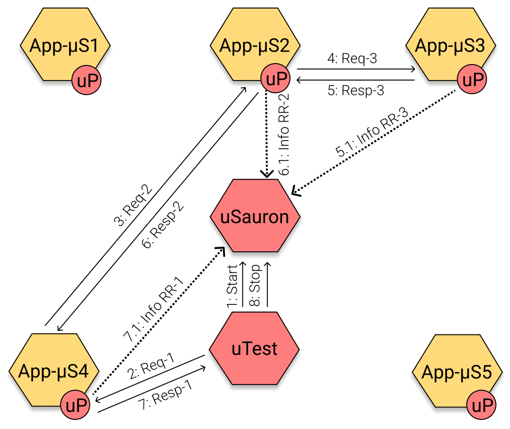


Figure 2: Example test execution sequence

## IV. Experimentation

This section reports the experiments run to assess the MacroHive's strategy in terms of coverage, fault detection, and cost. We check MacroHive's performance against four state-of-the-art black-box testing tools – namely, *EvoMaster* [?], *ResTestGen* [?], *bBOXRT* [?], *RESTler* [?] - and we investigate the pros and cons of the "grey-box" strategy featured by MacroHive. The other mentioned tools (RESTest [?], QuickRest [?], and the Eclipse plugin [?]) are either not available or cannot be run.

For repeatability and reproducibility, we provide MacroHive code for running the experiments[7].

### *A. Experimental subject*

The experimental subject is *TrainTicket*, a well-known open-source MSA benchmark, composed of 41 microservices [?]. This MSA has been extensively used in previous research and is considered representative of a real-world MSA [?], [?], [?], [?], [?], [?], [?], [?], [?], [?]. It is worth noting that other usual benchmarks in the related literature, such as *Sock Shop*[8] (6 microservices), *Pet Clinic*[9] (3 microservices), *FTGO*[10] (7 microservices), *PiggyMetrics*[11] (3 microservices), used in [?], [?], [?], [?], [?], are inadequate for testing MSAs: they are (small) collections of microservices that do not interact to each other – in this sense, they are not realistic MSA. For this reason, these subjects are not suitable for the grey-box experiments. However, in the online appendix[??] we report the results of MacroHive on a subset of these subjects (namely, *FTGO* and *SockShop*), confirming its performance as a pure black-box technique.

### *B. Tests generation strategies*

We define two tests generation strategies for MacroHive:

[5]https://prometheus.io/
[6]https://www.jaegertracing.io/
[7]https://github.com/uDEVOPS2020/MacroHive
[8]https://microservices-demo.github.io/
[9]https://github.com/spring-projects/spring-petclinic
[10]https://github.com/microservices-patterns/ftgo-application
[11]https://github.com/sqshq/piggymetrics

TABLE III: Coverage metrics

| Coverage metric | Description |
|---|---|
| *Status code class* | 100% status code class coverage when it is able to trigger both correct and erroneous status codes. Conversely, if it only triggers status codes belonging to the same class (either correct or erroneous), the reached coverage is 50%. 2XX class represents a correct execution and 4XX and 5XX classes represent an erroneous execution. |
| *Status code* | Ratio of the number of obtained status codes to the total number of status codes documented in the OpenAPI specification, for each operation. 100% status code coverage if, for each operation, it is able to test all the status codes. |

- MacroHive_pV (pV): generates a *2-way* test suite with *valid* input classes;
- MacroHive_p (p): generates a *2-way* test suite with *valid* and *invalid* input classes.

Mixed test suites are expected to provide better coverage results since they run both valid and invalid tests. We aim to compare the effectiveness of both strategies on edge and internal microservices. The comparison with RESTler, bBOXRT, and RestTestGen is on tests with valid and invalid input; EvoMaster generates tests only with valid input.

All tools are run 10 times each on any of the 34 externally accessible services, out of the 41 microservices in *TrainTicket*. Because EvoMaster runs tests with and without authentication (a token is provided), each tool is run both ways for fairness of comparison. Compared tools have been configured with theirs default settings or, when available, with the configuration that was shown to yield the best performance in the respective papers. For instance, RESTler has been configured with the BFS-cheap algorithm, which was the one that achieved best results with low time budgets [**?**]. When available, the maximum time budget is set to 150 seconds, namely 10 times the average time required by MacroHive to perform a testing session (15 seconds). We use *Burp Suite* to collect black-box tests input and output [**?**]. Then, we export the logs and feed them in Restats [**?**], a tool to compute coverage metrics.

## C. Research Questions

Three research questions are defined, to assess coverage, fault detection ability and cost of the proposed strategy.

### Coverage

An objective commonly pursued in specification-based testing of service-based software is to maximize *coverage*. Corradini *et al.* list eight coverage metrics [**?**]; we consider two of them, which apply to coverage of internal microservices too, for which we do not require availability of the API specification. They are *status code* coverage (SC) and *status code class* coverage (SCC) (Table **??**). While these coverage metrics are useful when testing individual services, they provide insufficient information for inter-dependent (micro)services as in an MSA. Indeed, coverage values observed with black-box testing at edge level give no insight about internal MSA dynamics and the internal failing microservice(s) within the MSA.

MacroHive grey-box strategy allows to compute the coverage of paths *internal* to the MSA. Namely, it computes coverage values at the various levels of dependency. For instance, 50% status code class coverage at a certain dependency level in an MSA with two internal microservices A and B may be achieved via coverage of A = 0% and B = 100% as well as by A = B = 50%. The latter is preferable as interactions with both microservices are covered.

To account for this, MacroHive measures the status code class and dependencies coverage of the internal microservices. For dependencies coverage, the number of all possible dependencies of a microservice is inferred from execution traces by running all the generated pairwise tests (4,600 tests) repeated 10 times, for a total of 46,000 test (to account for randomness introduced by combinatorial testing) – an approach commonly preferred for microservices, for which a static dependencies inference strategy is not exhaustive [**?**], [**?**], [**?**].

Spotting scarcely-covered internal services highlight those that need to be tested more from a unit testing perspective. Also, it allows discriminating the balanced from unbalanced internal coverage values under the same edge-level coverage.

We expect a combinatorial approach to increase internal coverage since the different combinations of values of input parameters should trigger different internal patterns.

**RQ1**: *What is the coverage of* MacroHive *compared to black-box testing?*

### Fault detection

An internal perspective of the MSA is expected to provide useful insights into failing behaviours, by supporting fault localization. A failure[12] observed at edge-level can be determined by a fault activated in any of the internal services along the request propagation chain (we call it a *propagated failure*) or by the edge service itself. Different failures can be caused by a single faulty microservice (*common-cause failure*). A pure black-box strategy ignores this distinction, making root cause analysis - and debugging - harder. An even more subtle situation occurs when a fault in a microservice is propagated within the MSA and does not achieve the edge service, namely the microservice failure is masked by the MSA (e.g., it could be tolerated by some other service): in this case, there may be a silent erroneous state within the MSA that escapes black box testing (we call it a *masked failure*).

**RQ2**: *What is the fault detection ability of* MacroHive *compared to black-box testing?*

### Cost

The cost of MacroHive is due to the number of generated

[12]Without loss of generality, failures considered hereafter are as defined in Section **??**, namely 5xx status codes.

test cases to run (like the black-box strategies) plus the cost of monitoring due to the grey-box-level testing. The combinatorial technique adopted by MacroHive is expected to significantly decrease the number of generated test cases. The monitoring infrastructure, on the other hand, adds additional cost compared to other techniques.

**RQ3:** *What is the cost incurred by* MacroHive *compared to state-of-the-art black-box testing techniques?*

## V. Results

### A. RQ1: Coverage

The four state-of-the-art techniques, MacroHive_p, and MacroHive_pV are run to compute the SCC and SC coverage reached by the respective test suites.

Therefore, we conducted a Friedman test, which is robust to non-normality and heteroscedasticity, with the Iman and Davenport extension on SCC and SC coverage values of each microservice and each repetition with a level of significance $\alpha = 0.05$ [?]. The test detects if at least one factors level significantly differs from another. With *p-value* $< 2.2$E-16 for both SCC and SC values it rejects the null hypothesis that average coverage values do not significantly differ.

Figures **??** and **??** show the *post hoc* pairwise comparison results by the ranking resulting by the Nemenyis test critical difference plot [?]. Techniques with no significant difference are grouped together using a bold horizontal line the greater the distance between two algorithms, the smaller the p-value for the null hypothesis of equal performance (the distance being the average ranking). It shows that the techniques are almost equivalent as for SCC coverage, except for RESTler, which is ranked first. The second plot shows that RESTler, MacroHive_p, bBOXRT, and MacroHive_pV are equally good in terms of SC coverage since they are able to find a higher number of different HTTP status codes compared to EvoMaster and RestTestGen.

Figure **??** plots the Status Code Class coverage for each edge microservice, while Figure **??** shows the average SCC coverage among all edge microservices. The Figures show that the performance of the various approaches is comparable. Since SCC coverage considers only two classes, values lower than $50\%$ mean that the test is unable to cover a *documented status code* (i.e., a status code described in the API) for one or more methods of the microservice under test.

SCC values of MacroHive_p and RESTler are always greater than $50\%$, meaning that it is able to obtain at least one documented status code for each method, on average for each microservice. Among the MacroHive variants MacroHive_p shows the best performance with a slight difference.

Figure **??** and Figure **??** show the Status Code coverage per microservice and the average SC per technique. As in the case of SCC, the results are similar on the average. Values greater than $25\%$ (SC) mean that the techniques are able to obtain at least a quarter of the documented codes, since the Status Code Class coverage can be $50\%$ when $100$ different status codes are specified and just one of them is detected.

Although MacroHive does not consider any heuristic to improve the coverage obtained in the testing session, the results show that it is comparable to the state-of-the-art approaches in terms of reached coverage.

The real advantage of MacroHive is the ability to measure internal coverage. The following investigation aims to evaluate to what extent MacroHive is able to exercise internal microservices through edge requests.

To investigate this aspect, let us define the dependency level $L_r$ of a request $r$ made to an edge microservice $M_0$ as the length of the path of requests $p_r =< M_0, M_1, \ldots, M_{L_r} >$ made from $M_0$ to the other microservices in the MSA. For instance, a level-2 dependency means that $M_0$ invoked a service $M_1$, which in turn invoked $M_2$. In addition, we group the edge-level microservices in different classes, based on the dependency level. To this aim we ran $T = 46,000$ tests executed on all the edge-level microservices for inferring the potential dependencies (Section **??**). In detail, we ran $T_i$ tests for the $i$-th microservice, with $T = \sum_{i=1}^{N} T_i$. The execution traces of the $T_i$ tests directed to the $i$-th microservice can be seen as a set of paths $P_i = (p_1, p_2, \ldots, p_{T_i})$. From these, we have drawn the maximum dependency level ($L_{Max} = max_{1 \leq r \leq T_i}(L_r)$) observed for that microservice, and assigned the microservice to the class based on it, $C = L_{Max}$.

For *TrainTicket*, the biggest dependency level of all edge microservices is 5, resulting in 6 different classes, from 0 to 5. Specifically, it has 13 microservices belonging to class 0, meaning that they have no dependencies with internal microservices; 11 class-1 microservices; 5 class-2 microservices; 3 class-3 microservices; 1 class-4 and 1 class-5 microservices.

Figure **??** shows the SCC coverage achieved by the two

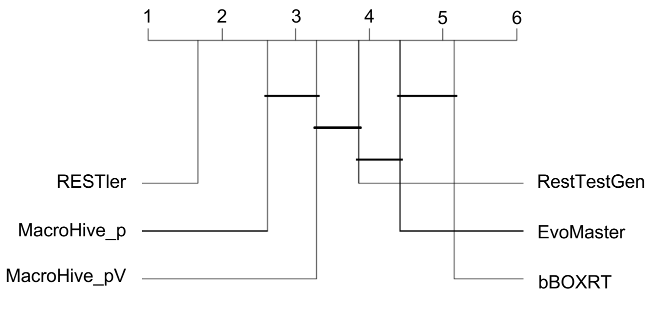


Figure 3: RQ1: Critical Differences (CD) of SCC coverage

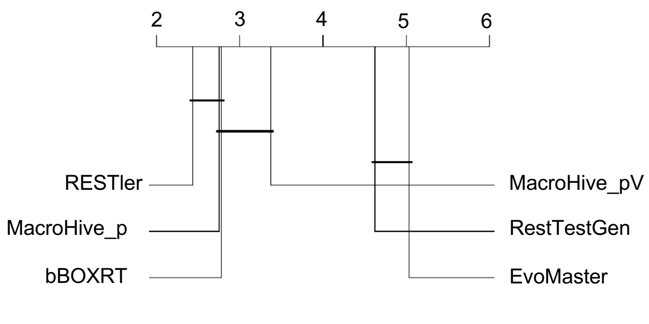
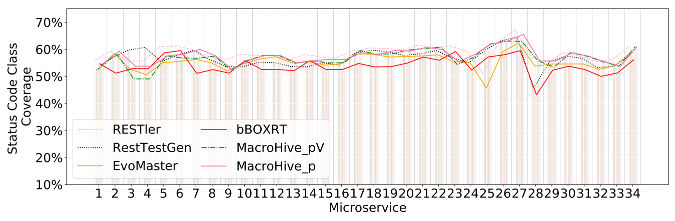
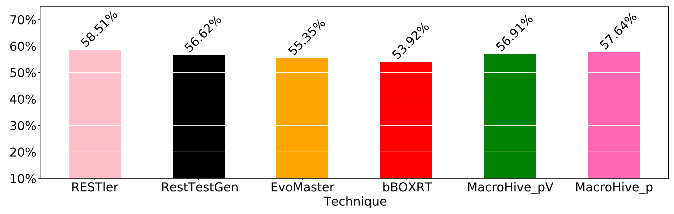


Figure 4: RQ1: Critical Differences (CD) of SC coverage

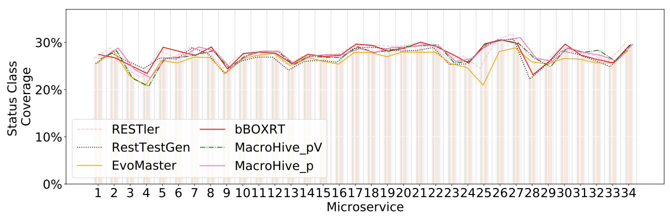


(a) Per edge microservice

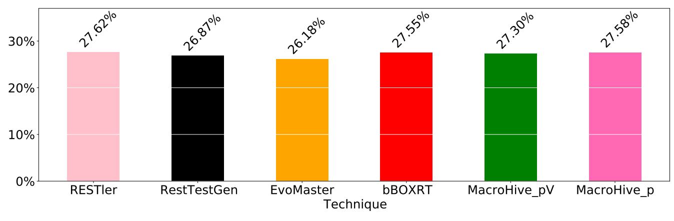


(b) Average per Technique

Figure 5: RQ1: Status Code Class coverage



(a) Per edge microservice



(b) Average per Technique

Figure 6: RQ1: Status Code coverage

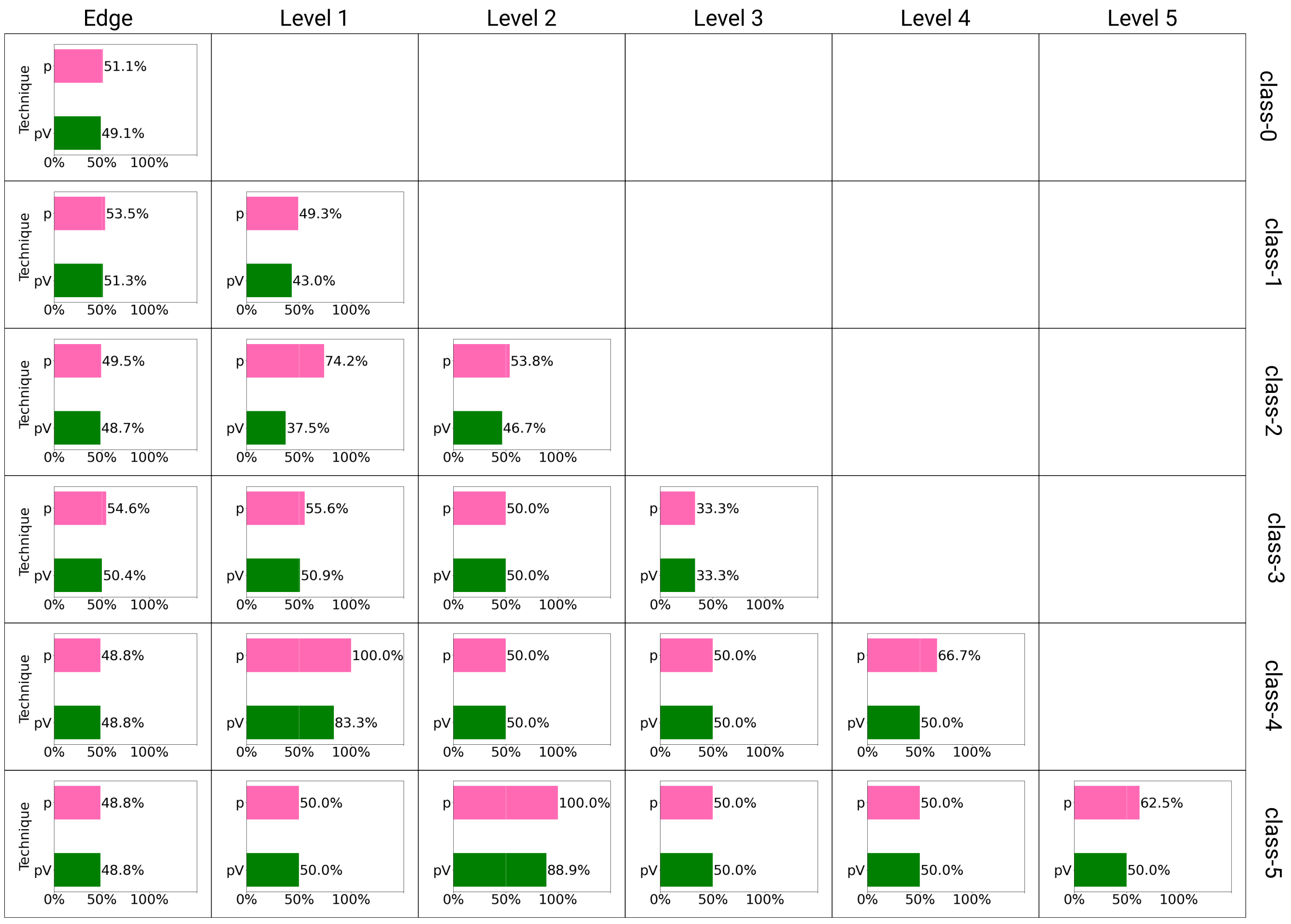


Figure 7: RQ1: Status Class coverage per level and class

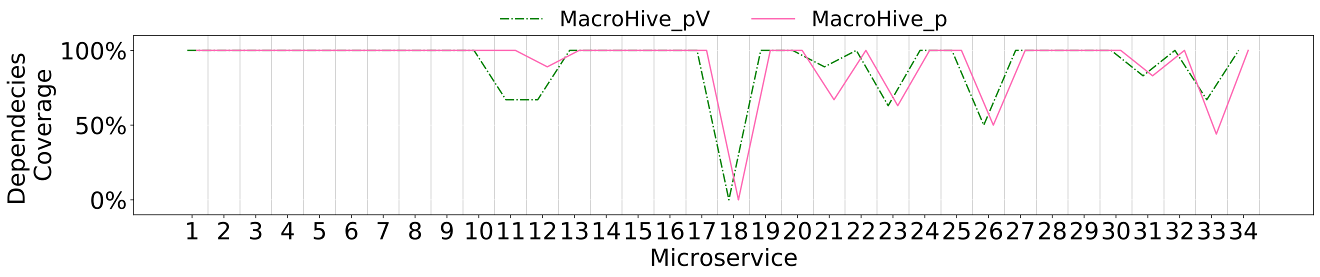


Figure 8: RQ1: Dependencies coverage

variants of MACROHIVE. The internal microservices API is not always known when testing the edge microservice and observing the internal chain of calls. For this reason, SCC coverage for internal microservices is computed assuming that different methods of the edge microservice invoke different methods of the internal ones[13]. We see that MACROHIVE_P reaches the best values for the deepest dependency levels of microservices of classes 1, 2, 4 and 5. Because of the lower failure rate compared to MACROHIVE_PV, this can suggest that 4xx codes rise up the coverage values. Indeed, invalid inputs are more prone to cause such codes (e.g. 400 – bad request HTTP code). This indicates that pairing invalid classes is the best approach for maximizing status class coverage at deepest MSA levels.

Figure **??** shows the dependencies coverage achieved. The variants exhibit the same coverage; they are able, however, to find different microservice dependencies.

## B. RQ2: Fault detection

To answer RQ2, we run the four techniques and MACROHIVE to execute the testing sessions as described in Section **??**.

Figure **??** reports the Average Failure Rate (AFR), namely the number of failures detected averaged over edge microservices and repetitions. The Friedman test is run with a level of significance $\alpha = 0.05$. The test returns a *p-value* $< 2.2$E-16, rejecting the null hypothesis that average failure rates values do not significantly differ.

[13]SCC is preferred to SC because it does not need the API specification of internal microservices.

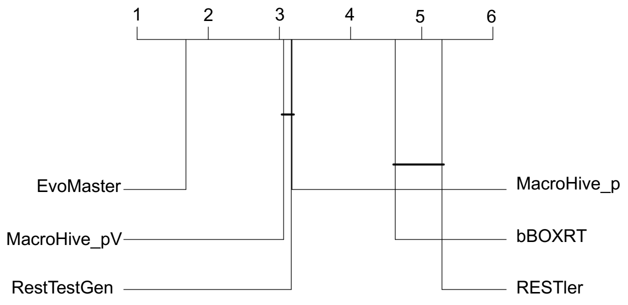


Figure 10: RQ2: Critical Differences of algorithms failure rate

Figure **??** plots the Nemenyis test critical difference. While *bBOXRT* and *RESTler* have (statistically) significantly lower AFR, MACROHIVE variants show values similar to *ResTestGen*. EvoMaster exhibits the best AFR, almost 0.175. The AFR value greater than 0.15 (EvoMaster and MACROHIVE_PV) indicates that more than 15% of the generated tests expose a failure. Both EvoMaster and MACROHIVE_PV generate only inputs compliant to the API, anyway achieving the highest failure rate. This may indicate a poor specification or a better ability to exercise the code (with poor exception handling) compared to invalid requests which can be handled by early input validation (e.g., a malformed request).

Besides the failures exposed at edge level, MACROHIVE spots internal failures, highlighting the internal failure propagation chains, as well as possible masking effects.

Figure **??** shows the average number of internal failures (again over all the microservices and repetitions) detected by

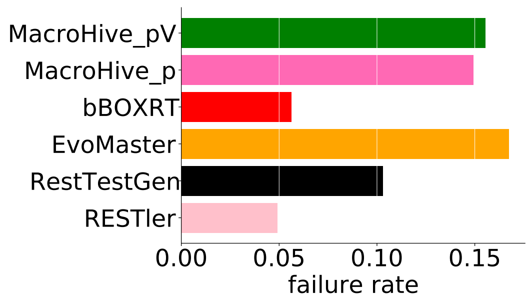


Figure 9: RQ2: Average failure rate

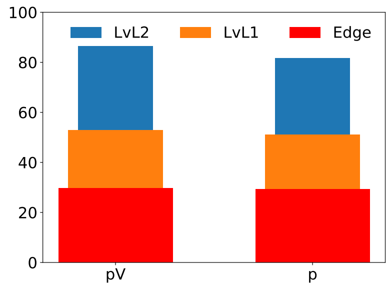
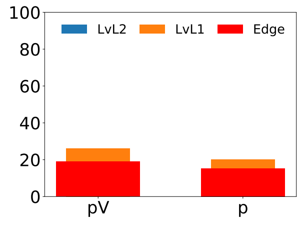
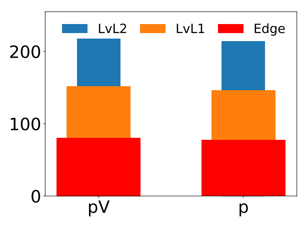


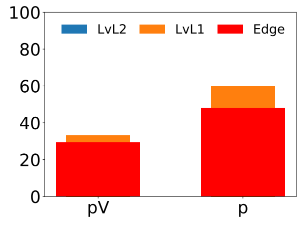
Figure 11: RQ2: Edge and internal failures



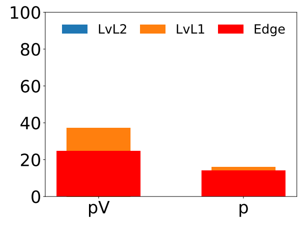
(a) class-1



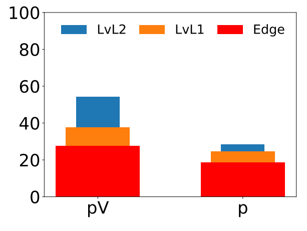
(b) class-2



(c) class-3

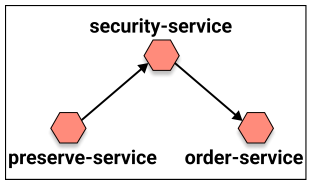


(d) class-4

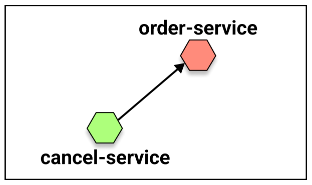


(e) class-5

Figure 12: RQ2: Number of failures detected for edge and internal microservices up to level 3



(a) Propagated failure



(b) Masked failure

Figure 13: Examples of failing internal microservices

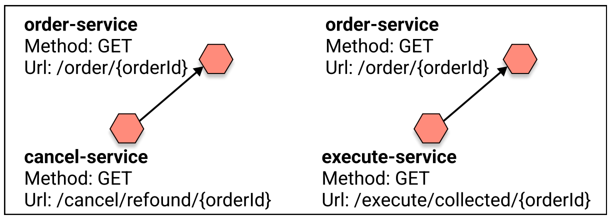


Figure 14: Example of common cause failure

the variants of MacroHive, broken down by level. As shown, MacroHive_pV reaches slightly better results. We did not find failures deeper than level-2.

Figure **??** shows the average number of internal failures for classes 1 to 5 (class 0 is omitted, as there cannot be internal failures). Except for the 3 microservices of class-3, MacroHive_pV is the variant performing best at detecting internal failures. We see also that level 2 failures are mostly in class-2 and class-5 microservices.

A deeper analysis allows to identify the microservice originating the failure propagation chain. Figure **??** shows two situations spotted in *TrainTicket* thanks to MacroHive. Figure **??** shows a propagated failure: two failing internal services, `security-service` and `order-service`, cause the edge microservice `preserve-service` to fail. Figure **??** shows a masked failure: a test passed, despite a failure occurred in the internal microservice `order-service`.

Failures in different microservices may be caused by a common faulty service. An example is in Figure **??**. Two services (`cancel` and `execute`) fail by exhibiting similar behaviour. Through the information provided by MacroHive, a tester can infer that the primary cause is the method *GET /order/orderId* of `order-service`. The grey-box approach unveils such *common-cause failure* occurrences.

Table **??** reports the average number of propagated and masked failures among all microservices detected by the two variants of MacroHive, together with the average number of executed tests and failures observed at the edge. MacroHive_p exposes 328.7 propagated failures and 1 masked on average, MacroHive_pV 352 and 4. All exposed masked failures come from the same microservice (`cancel-service`). These are internal failures that would have not been detected by a black-box strategy, as they do not reach the edge microservice. These can silently corrupt the state of the MSA and manifest unexpectedly in operation [**?**], [**?**], [**?**]. Clearly, some of these failures might have been tolerated by the designed fault tolerance mechanisms, others might be simply stopped from propagating by the program control flow; in both cases, engineers are interested in figuring out the reasons for the microservice failure. Furthermore, propagated failures would have been associated with edge microservices, while the true cause would be an internal error.

TABLE IV: RQ2: Number of propagated and masked failures

| Variant | Executed tests | Edge Failures | Propagated failures | Masked failures |
|---|---|---|---|---|
| MacroHive_pV | 4,460 | 756.7 | 352 | 4 |
| MacroHive_p | 4,497.6 | 744 | 328.7 | 1 |

## C. *RQ3: Cost*

RQ3 is about the cost of the proposed techniques. Figure **??** shows the average number of tests executed by each technique for each microservice in the previous research questions. The number of tests generated by MacroHive in each testing session is at least one order of magnitude lower than the other techniques. Furthermore, the very low variance of number of tests generated by MacroHive depends on the stateless generation methodology. In fact, MacroHive generates always the same number of tests (for a certain microservice) with the combinatorial strategy. Conversely, the other tools generate

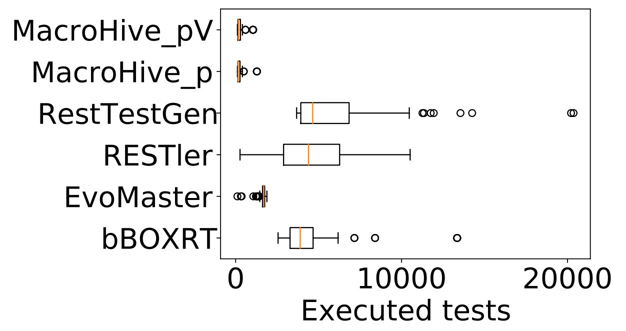


Figure 15: RQ3: Number of executed tests per technique

a considerable amount of tests trying to explore parameter dependencies, which are very hard to discover in complex distributed systems such as MSA.

MacroHive relies on monitoring; its costs concern deployment and run-time overhead. As each microservice is supplied with a sidecar, two containers are deployed per microservice. This impacts the deployment process, as more containers must be independently deployed, and linearly affects scalability.

The monitoring overhead is related to the delay introduced by proxies redirecting requests (responses) to (from) a microservice. This is a known issue, one of the main challenges in building a service mesh [**?**]. The delay due to proxies is the additional time incurred to forward a request or the corresponding response. We measured this delay as $1 \pm 0.5$ ms (median and semi inter-quartile range over all microservices), with the microservices response time equal to $7 \pm 2.5$ ms.

## VI. Threats to Validity

The main threats to validity and possible mitigation strategies are as follows.

*a) Construct validity:* dependencies coverage and internal status code class coverage are computed on an estimation of the ground truth. Indeed, it is built only with MacroHive, then eventual dependencies that we are not able to explore are not considered. Furthermore, we consider a failure propagated when an edge failure presents at least an internal failure. This may not always be the case, as we can have mixed chains of propagated and masked failures. We are working on the identification of these tricky cases too.

*b) Internal validity:* despite our efforts (including code inspection by senior co-authors) to ensure that the MacroHive prototype is free of defects, their presence cannot be excluded and could partly corrupt the experimentation. Furthermore, the sidecar proxies introduce a delay in the internal requests, which could have determined some observed failures. Our inspection of results did not identify any such case.

*c) External validity:* the use of the only *TrainTicket* subject hinders generalization. The online appendix to this study includes results on two further subjects; while these additional results confirm MacroHive performance in a edge-level perspective, by their nature (limited involvement of internal microservices) they could not be used for experiments with an internal perspective. Finding realistic MSAs is a recognized problem [**?**]; we will contribute to this by collaboration with industry in the frame of ongoing projects.

## VII. Conclusions

Grey-box testing of Microservice Architectures allows testers to get information about internal microservices behaviour, in terms of coverage and failures. We presented a grey-box testing strategy and the MacroHive infrastructure, which automatically generate and execute test suites from the API documentation of edge microservices, monitoring and then analyzing interactions among internal microservices.

The case study shows that the proposed approach is very effective in detecting different kinds of failures (edge failures, internal failures, propagated failures, and masked failures), by exploring internal dependencies, and providing useful information about faulty microservices.

The cost of the technique is paid mostly in terms of overhead, since it requires to deploy a proxy for each microservice to monitor. This cost is traded off by a better understanding of faulty behaviours of the internal microservices. As this kind of tests can be executed in a staging environment, the overhead does not impact the MSA in production.

## Acknowledgement

This project has received funding from the European Unions Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 871342. It is also supported by the DIETI COSMIC project.