The Journal of Systems and Software 207 (2024) 111857

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automated functional and robustness testing of microservice architectures ☆

Luca Giamattei *, Antonio Guerriero, Roberto Pietrantuono, Stefano Russo

Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Napoli, Italy

ARTICLE INFO

MSC:
0000
1111

Keywords:
Microservices testing
Functional testing
Robustness testing
Causal inference

ABSTRACT

Microservice Architectures (MSA) are nowadays largely adopted by companies in several domains to provide on-demand services. The reliability of microservices is fundamental to avoid failures compromising the business functionalities. MSA automated testing is possible thanks to well-defined service interfaces specified in open formats like OpenAPI/Swagger.

To support automated MSA functional and non-functional testing, we define a framework that: (i) generates test cases with valid and invalid inputs, and executes and monitors tests; (ii) provides coverage and failure information not only on edge, but also on internal microservices; (iii) has the novel feature of identifying *causal relations* in observed chains of microservices failures.

We abstract the testing process of MSA, present the MacroHive framework and its *causal inference* engine, compare it experimentally to state-of-the-art tools, and discuss its benefits in the MSA testing process.

MacroHive exhibits performance comparable to advanced existing tools in terms of edge-level coverage. However, MacroHive has a better failure rate and provides the unique advantages of giving insights about internal coverage and failures, and of inferring causality in failure chains, evidencing microservices to be improved to increase the whole MSA reliability.

1. Introduction

The Microservice Architecture (MSA) architectural style is nowadays largely adopted by companies like Netflix, Amazon, eBay, PayPal, and Twitter due to the possibility to independently develop and deploy loosely coupled services running in their own processes and interacting via lightweight mechanisms (Lewis and Fowler, 2014). These characteristics favor the adoption of lean or agile development practices like DevOps, enabling rapid and frequent software releases (even many per day). Such dynamic contexts demand for high automation of the testing process to assess and improve software quality (Patel and Tyagi, 2022).

As MSA code is polyglot and distributed across various repositories, black-box testing is usually deemed as a very viable option for testing, requiring only access to the system with a well-defined interface (Viglianisi et al., 2020).

Specification-based (black-box) testing foresees two main phases (Pezzè and Young, 2008): test specification and test cases generation. A test specification describes how a test case is required to be. Since MSA specifications are documented with open formats like OpenAPI/Swagger (Ma et al., 2018),[1] the Application Programming Interfaces (API) of microservices can be automatically retrieved and parsed to generate a test specification. Then, automatic techniques for RESTful web services can be borrowed for MSA testing, as long as they are able generate test cases from the specification (Corradini et al., 2021a; Atlidakis et al., 2019; Arcuri, 2019). State-of-the-art (SOTA) tools leverage various techniques (depending on the testing purpose) for generating inputs compliant with the API specification. For instance, EvoMaster (Arcuri, 2021) generates *valid* inputs (for *functional testing*); RestTestGen (Corradini et al., 2022), RESTler (Atlidakis et al., 2019), and bBOXRT (Laranjeiro et al., 2021) generate *invalid* inputs intentionally violating the specification (for *robustness testing*).

Since MSA have characteristics that allow the automation of several tasks within the testing process, we first abstract the main phases of a typical testing process: after the preliminary identification of the testing objective (functional or non-functional) we have 5 phases: *APIs specification collection*; *Tests specification definition*; *Test suite generation*; *Test execution*; *Test reporting*. We then present MacroHive, a framework built as a collection of microservices deployable along the MSA in a testing environment. We highlight how MacroHive covers the phases of the outlined process, comparing it against four state-of-the-art tools on three well-known benchmark systems (*TrainTicket*, *Sock Shop*, and *FTGO*).

MacroHive was first presented in own previous work (Giamattei et al., 2022), focusing on test generation. Here it is extended by a

☆ Editor: Prof W. Eric Wong.

* Corresponding author.
E-mail address: luca.giamattei@unina.it (L. Giamattei).

[1] OpenAPI Initiative, https://www.openapis.org.

https://doi.org/10.1016/j.jss.2023.111857
Received 18 February 2023; Received in revised form 8 September 2023; Accepted 25 September 2023
Available online 5 October 2023

causal reasoning-based inferential engine that detects the microservice culprit for a chain of failures, and reports the chain along with other metrics to support failure analysis and debugging. MacroHive differs from state-of-the-art tools in that: *(i)* it supports both functional and robustness testing, *(ii)* it adopts a combinatorial testing strategy, *(iii)* it observes internal interactions and report about internal coverage, which is crucial in MSA testing (Ghani et al., 2019), and *(iv)* it is the first tool incorporates causal reasoning to infer the likely root cause of a failure.

We evaluate MacroHive comparing it to SOTA tools. We first discuss the extent to which they support the MSA testing process (Research Question RQ1). Then, we report the results of experiments to quantitatively assess MacroHive performance with respect to comparable tools for *functional* (RQ2) and for *robustness* testing (RQ3).

MacroHive shows results comparable to SOTA tools, achieving high coverage of various response codes in functional testing, and achieving high failure rate in robustness testing. Unlike other tools, MacroHive provides the additional advantage of identifying the microservices more likely to be the culprit in chains of observed failures inside the MSA; this comes from the integration of a gray-box monitoring infrastructure – tracking the internal coverage and detecting internal failures (not visible in a black-box perspective) – and of the causal reasoning-based inferential engine.

In the following: Section 2 gives an overview of the related work and of state-of-the-art tools used for MSA testing. Section 3 illustrates the high-level testing process for MSA. Section 3 describes how MacroHive supports the testing process. Sections 5 present the experiments and results, respectively. Section 7 concludes the paper.

2. Related work

Several studies present testing techniques for MSA; a systematic mapping study is presented by Waseem et al. (2020).

MSA *resilience testing* is investigated by Heorhiadi et al. (2016) and Long et al. (2020): the former study proposes the *Gremlin* framework to systematically test the failure-handling capabilities of microservices, by injecting faults into inter-service messages; the latter presents a fitness-guided technique to find as many bugs in the fault handling logic as possible in a set amount of time.

MSA *reliability testing* is investigated in own work (Pietrantuono et al., 2018), presenting an *in-vivo* testing technique for on-demand MSA reliability estimation at any time in the operational phase.

Microservices *performance testing* is the goal of the FPTS framework proposed by de Camargo et al. (2016) to evaluate the performance delivered by single microservices, through a workload created thanks to annotations to be inserted within their source code. Performance testing of containerized MSA is addressed by Lei et al. (2019), who propose a method based on Kubemark, a tool for running Kubernetes experiments on simulated clusters.[2]

As for *functional testing*, because of the prominent role of RESTful API in MSA (Arcuri, 2021), MSA testing mainly leverages existing tools for black-box testing of RESTful web services. These tools usually have 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 several tools (Corradini et al., 2021a): they aim to maximize the coverage of methods specified in the API via data and operations dependencies; the comparison is in terms of "robustness", meant as the ability to manage real-world systems, and of the coverage criteria defined by Martin-Lopez et al. (2019). The resulting three main tools are: RestTestGen, RESTler and bBOXRT.

RestTestGen (Corradini et al., 2022) 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 by re-using past observed values.

RESTler (Atlidakis et al., 2019) features stateful input generation via fuzzing, aiming to find security and reliability issues. It infers producer–consumer dependencies among the specified request types, and analyzes feedback from responses in executed tests. Similarly to RestTestGen, input values are selected from a user-configurable dictionary, or from previously observed values.

bBOXRT (Laranjeiro et al., 2021) is for robustness testing of REST services. It is designed around method for injecting faults in requests, attempting to trigger erroneous behaviors. Specification-compliant input values are randomly generated and then mutated to observe the 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 (Arcuri, 2021). 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. (2020) 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 improves results but requires testers to specify dependencies in IDL — this is time-consuming, requires a deep knowledge of the system under test, and reduces automation.

MSA testing can thus 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 (Arcuri, 2021). However, it also poses challenges that we aim to address with this work. First, applying the mentioned tools requires running distinct testing sessions for every edge microservice — a practice that does not scale well with the number of microservices (Zhou et al., 2021). 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 the edge level.

RestTestGen, RESTler and bBOXRT are hereafter considered as tools for robustness testing since they generate inputs violating the specification to cause failures, while EvoMaster is considered a tool for functional testing since it generates inputs compliant with the specification.

3. MSA testing process

The MSA testing process is sketched in Fig. 1. After the preliminary definition of the testing objective (phase ⓪), the process entails five phases.

APIs specification collection. Phase ① consists of collecting the API specifications of microservices. They are often in the OpenAPI format, and include service Uniform Resource Identifier (URI), HTTP method, type and name of every parameter, and HTTP body. OpenAPI allows retrieving the interface of a microservice of interest from its IP address and port number.

Tests specification definition. Phase ② consists of extracting information from the API specification to generate the test cases specification. The API specification of all microservices in the MSA is parsed to extract an input space model (i.e., the set of factors that might affect the behavior or output of the system under test) consisting of: HTTP methods, URIs

[2] Kubernetes community, Kubermark user guide, https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scalability/kubemark-guide.md.

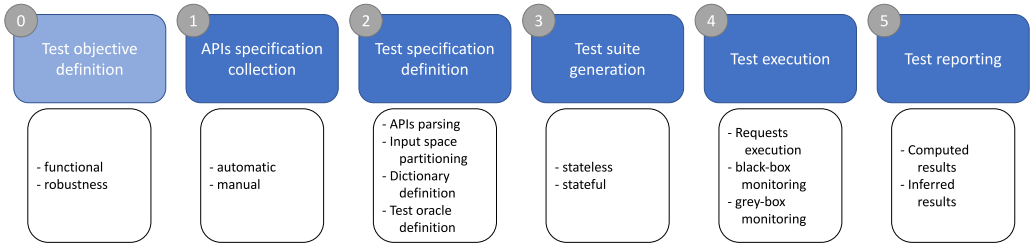


Fig. 1. MSA testing process.

and body templates, HTTP status codes, and parameters' details (type, bounds, default value, etc.).

Two solutions to define the set of possible values for each parameter are:

- Input space partitioning: equivalence classes for all parameters are defined, and they are categorized into valid and invalid. A class is valid if it contains only parameter values that comply with the microservice specification, and invalid if it contains only values that do not.
- Dictionary definition: a list of parameter values is provided, typically including those that are known to work better during testing or that better suit the service. The dictionary may be built manually before the testing session, and/or automatically during the test execution.

The last step of this phase is the test oracle definition. A complete OpenAPI specification reports the expected response codes for each method invocation. These are used to build a test oracle to automatically evaluate the test outcome. Specification-based solutions depend on the quality of the documentation; an incomplete/incorrect specification represents the main limitation. The more detailed the API specification, the better the definition of the test specification and test oracle.

Test suite generation. In phase ③ actual test cases are generated from tests specification. This is accomplished by sampling values from input equivalence classes (valid and/or invalid values, based on the test objective), or by sampling/mutating the input values specified in the dictionary.

Test generation techniques are stateless or stateful. Stateless techniques generate test cases statically from the API specification so that they do not require feedback from the previously generated tests. Stateful techniques generate test cases dynamically, using feedback from executed tests. They model operation and parameter dependencies, re-using parameter values from previous requests/responses and generating sequences of requests trying, for example, to create resources before accessing and/or modifying them.

Test execution. In phase ④ tests are executed as HTTP requests sent to edge microservices, that in turn can invoke internal microservices to execute complex business functionalities. A key component of test execution is monitoring, which involves gathering the request–response pairs from all microservices. This knowledge is required to comprehend how the entire MSA behaves when tests are run. We distinguish:

- *Black-box* monitoring: which collects only request–response couples exchanged with the edge microservices.
- *Gray-box* monitoring: which collects also the request–response couples exchanged with and among internal microservices.

Test reporting. In phase ⑤ each request–response pair is evaluated by the test oracle to detect failures. A set of metrics is computed depending on the testing objective. Response code coverage and response code class coverage are suitable for functional testing evaluation (Martin-Lopez et al., 2019), while number of failures and failure rate are indicative for robustness testing. When looking at an MSA as a whole thanks to gray-box monitoring, these metrics apply also at internal MSA levels.

Gray-box monitoring data are used also to infer relationships among operations or microservices (e.g., building an operation/service dependency graph). These relationships can be used to train models (e.g., Machine Learning models or causal models) able to identify microservices causing failure patterns in invocation chains for automatic root cause analysis (Ji et al., 2020; Wu et al., 2020).

4. MacroHive

MacroHive is conceived to support a tester along the MSA testing process, for both functional and robustness testing. Fig. 2 shows the four main components in its architecture: `uTest`, `uSauron`, `uProxy` (uP), and `uKnows`.

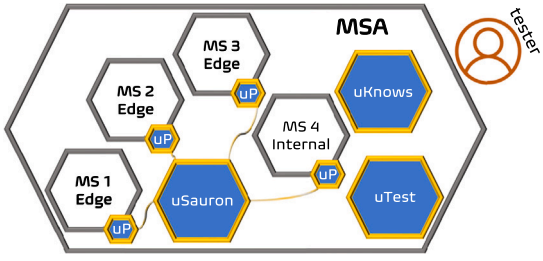


Fig. 2. The four MacroHive components, deployed along with the MSA under test.

`uTest` is responsible for the first four phases of the testing process. It automatically retrieves the OpenAPI specification for the MSA under test in phase ①, defines the test specification ②, generates test cases ③, and execute them performing also black-box monitoring ④.

An extract of an OpenAPI specification is shown in Listing 1. It describes a service with two parameters — one *in path* (`userId`, required) and one *in body* (`username`, required). It returns 201 or 400 HTTP status codes.

Listing 1: A sample microservice OpenAPI specification

```
host:   exampleHost:8080
paths:  '/user/{userId}/username':
  post:
  parameters:
      - name: userId
        in: path
        required: true
        type: integer
      - name: body
        in: body
        required: true
        schema: '$ref': './definitions/usernameSchema'
  responses:
      '201': description: 'Created'
      '400': description: 'Bad Request'
definitions:    usernameSchema:
  type: object
  properties:
      username:
          type: string
          value: LCG001 #Example value
  required:
      - username
```

Table 1
Input space partitioning for the microservice of Listing 1.

Parameter	Name	Type	Equivalence classes	Category
p_1(*required, in path*)	*userId*	*integer*	$c_{1,1}$: positive value in range	*valid*
			$c_{1,2}$: negative value in range	*valid*
			$c_{1,3}$: alphanumeric string	*invalid*
			$c_{1,4}$: no value	*invalid*
p_2(*required, in body*)	*username*	*string*	$c_{2,1}$: string in range	*valid*
			$c_{2,2}$: specified example value(s)	*valid*
			$c_{2,3}$: empty string	*invalid*
			$c_{2,4}$: no string	*invalid*

`uTest` parses the retrieved API to extract the *input space model*. The equivalence classes are defined for each parameter based on the type and then categorized into valid and invalid (e.g., integer: positive/negative value in range as valid and alphanumeric/empty string as invalid) (Bertolino et al., 2020). It also automatically extracts, for each method, the expected output from the API specification, hence building a test oracle. An example of input space model for the specification in Listing 1 is shown in Table 1.

Test generation adopts a combinatorial pairwise strategy, particularly suited to detect multi-factor faults — a high percentage of software faults (Hu et al., 2020).

For functional testing, `uTest` generates a nominal test suite, composed of only test cases with *valid* inputs, namely a test in which the inputs of all parameters are chosen from valid classes (when available, examples and default values are preferred). For instance, a test case with valid inputs generated from the specification in Table 1 shall have for p_1 (*userId*) a value chosen from class $c_{1,1}$ (a positive value in range); for p_2 (*username*) a value from class $c_{2,2}$ (the specified example value in Listing 1).

For robustness testing, `uTest` generates a test suite with *valid* and *invalid* inputs. A test case with invalid inputs is one in which for at least one parameter the input values are taken from an invalid input class. For instance, a test case with invalid inputs shall have for p_1 (*userId*) a value chosen from class $c_{1,3}$ (an alphanumeric string) or ($c_{1,4}$), and/or for p_2 (*username*), a value from class $c_{2,1}$ (a string in range) or $c_{2,1}$ (no string).

Actual test cases are generated by randomly picking values from selected equivalence classes.

`uSauron` and `uProxy` complement `uTest` in phase ④ by performing gray-box monitoring. They constitute a service mesh infrastructure aiming to trace dependencies between microservices and to log (both edge and internal) request–response couples generated by the executed tests. Although many monitoring tools are available (e.g., Prometheus,[3] Jaeger[4]), we preferred to build our infrastructure favoring automation and avoiding code instrumentation. `uProxy` (*uP*) is deployed alongside each microservice to test/monitor, complying with the sidecar pattern (Burns and Oppenheimer, 2016; Jamshidi et al., 2018). 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 pair.

The information packet is composed of: request/response URL, request/response body, HTTP response code, response time, and 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 the executed tests.

The last phase ⑤ is supported by `uKnows`, which collects data by `uSauron` and `uProxy`, and exploits *causal reasoning* to determine which microservices are responsible for failures and, in general, for erroneous behaviors.

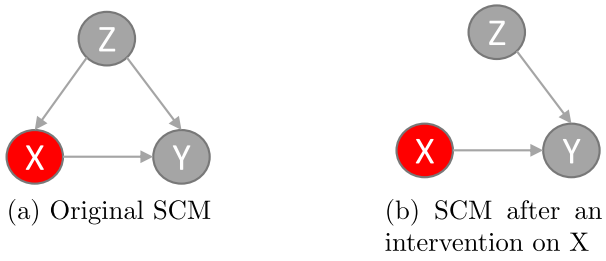


Fig. 3. Example of Structural Causal Model and intervention.

Causality is the influence by which an event contributes to the production of other events (Nogueira et al., 2022). There are two main activities in causal reasoning: Causal Structure Discovery (CSD), aimed at extracting a causal model (a mathematical representation of causal relationships between random variables) from observational data; and Causal Inference (CI), aimed at quantifying the effect of changing one or more random variables on others, starting from a causal model. The random variables describe quantity of interest (e.g., a categorical variable representing the response code of a microservice), and can be of any type (e.g., both numerical and categorical); the value they take depends on the probability distribution associated with them.

A widely adopted solution to model causality is to use *Graphical Causal Models* (GCMs). A GCM consists of a causal Direct Acyclic Graph (DAG) where nodes are random variables and edges define cause–effect (tail-arrow) relationships between couples of variables. The latter determines the impact of a change of a certain variable, called cause, over an outcome of interest, called effect. The most prevailing case is a Structural Causal Model (SCM), a GCM that uses a Functional Causal Model (FCM), where the value of each variable X_i is assumed to be a deterministic function of its parents $Pa(X_i)$ and of the unmeasured disturbance U_i ($X_i = f(Pa(X_i), U_i)$). An SCM is formally defined as follows.

Definition 1 (*Structural Causal Model (SCM)*)**.** An SCM is a Directed Acyclic Graph $\mathcal{G} = (\boldsymbol{X}, \mathcal{E})$, where nodes $\in \boldsymbol{X}$ are random variables and edges $\in \mathcal{E}$ are the causal relationships between them. Causal relationships are described as a collection of structural assignments $X_i := f_i(Pa(X_i), U_i)$ that define the random variables X_i as a function of their (endogenous) parents $Pa(X_i)$ and of (exogenous) independent random noise variables U_i.

CI aims at estimating the effect of setting one variable X_k to a specific value "x" (i.e., doing an intervention on X_k) on one or more variables X_i of interest. Pearl and Mackenzie (2018) introduced the *do-operator* (written as $P(X_i|do(X_k = x))$), a mathematical representation

[3] Prometheus open source monitoring system, https://prometheus.io/.
[4] Jaeger open source distributed tracing, https://www.jaegertracing.io/.

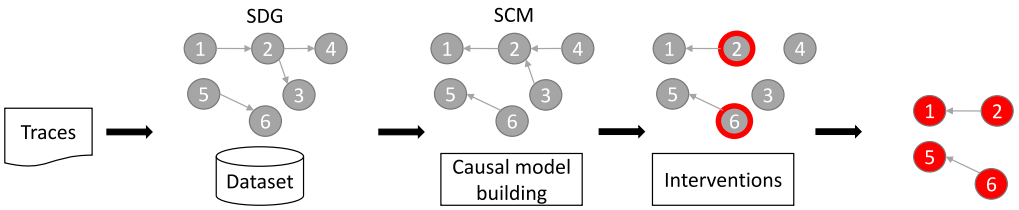


Fig. 4. Workflow of MacroHive causal inference component (uKnows).

of physical intervention, that changes the SCM graph by removing causal relations of X_k with its predecessors and replacing the definition $X_k := f_k(Pa(X_k), U_k)$ in the SCM with $X_k := x$. An example is in Fig. 3(a), where X, Y, and Z represent the behavior (properly codified — e.g., the HTTP status codes of responses in our case) of 3 microservices (M_x, M_y, and M_z) calling each other. A variable Z is a function of its parents and of a variable U_z capturing the random noise — the noise variables are usually not represented in the graph for simplicity, but there is a hidden node, U_x, U_y, U_z, associated with each variable with an arrow pointing to X, Y and Z respectively. When there is no parent, say for Z, the equation becomes $Z = U_z$. In the example, U_z is the random variable capturing the behavior Z and the value it takes depends solely on the U_z distribution. The behavior Z affects the other two (e.g., an erroneous/correct behavior causes others erroneous/correct behavior), and X affects Y; thus: $X = f(Z, U_x)$ and $Y = f(X, Z, U_y)$. Suppose we want to estimate the impact of failure of M_x on M_y ($X \to Y$ in the graph). Z is said to be a *confounder* since it causally affects both X and Y, generating a spurious association. This means, for instance, that the failure of M_y can be falsely attributed to M_x, even if M_z was the ultimate cause. Performing an intervention leads to the SCM reported in Fig. 3(b), which allows estimating the real effect of the behavior X on Y, because X is no longer influenced by Z.

Causal Inference requires a causal model, which can be built in two main ways: by intervening on variables and observing the post-intervention distributions (through controlled experiments), or by using CSD algorithms that aim to seize causal structure from observational data. Causal discovery algorithms can be divided into constraint-based (e.g., PC, FCI, RFCI), score-based (e.g., GES, FGES, GFCI), and FCM-based (e.g., LinGAM) (Glymour et al., 2019). An extensive discussion of CSD algorithms can be found in Nogueira et al. (2022).

MacroHive component `uKnows` leverages the power of causal reasoning to automatically infer causal relations between microservices. Fig. 4 shows its workflow. It extracts the Service Dependency Graph (SDG) from the output of `uSauron`, collected during the testing sessions, and derives a Direct Acyclic Graph (DAG). Multiple tests in a session have different inputs, thus dependencies based on different inputs will be captured by the graph.

Then, `uKnows` parses and transforms traces in a dataset containing an entry for every HTTP request, with the microservices involved in the interaction. The DAG and the dataset are the input for the CSD algorithm. In particular, the SDG is used as prior knowledge; it specifies required edges in the causal graph: a causal relation in the causal graph must exist between two microservices connected in the SDG. Then, the CSD algorithm infers the final SCM by fitting the causal graph with the dataset. This model characterizes cause–effect relationships between microservices.

For CSD, we use `py-causal`,[5] a python library that wraps the Java tool `Tetrad` (Ramsey et al., 2018). We use the FCI algorithm (Spirtes et al., 2001), one of the simplest solutions, with the default settings.[6] The FCI algorithm starts with a complete undirected graph connecting all the nodes and applies conditional independence tests to remove edges (with only edges that indicate potential causal relationships). The method tests possible d-separations $X \perp\!\!\!\perp Y | Z$ in the skeleton. If there is at least a variable in Z that d-separates[7] the edge, then it is removed (Nogueira et al., 2022). Finally, FCI applies several rules to direct the edges (Spirtes et al., 2001). It gives asymptotically correct results even in the presence of *confounders* (unobserved direct common cause of two measured variables) (Glymour et al., 2019). As shown in Fig. 4, the SCM built from the SDG will have inverted arrows. For instance, a failure in microservice M_1 is likely to find causes in downstream nodes (M_2, M_3, and M_4) and not the other way around.

In the *Interventions* step, `uKnows` does interventions to predict what would be the effect of a failure (in robustness testing) or of an erroneous behavior (in functional testing) of a microservice on the others causally related to it. We consider erroneous behavior HTTP status codes of classes 4xx and 5xx (Martin-Lopez et al., 2019); we consider failures only response codes of the class 5xx.

Assume we want to assess the effect of a failure of M_2 on M_1 and M_6 on M_5. We perform interventions on M_2 and M_6, setting them to fail (Fig. 4) and thus removing all the edges coming from M_2 and M_6 parents. Then, using a new data sample derived by sampling from the post-intervention distributions, we estimate the outcome by looking at the sample statistics. If a failure is predicted, a causal relation is detected and reported in the output (Fig. 4, last step). Note that the interventions are not physical, but queries to the model — a real intervention would be to inject a failure in M_2, or to inject a fault to cause its failure. This allows saving the cost of executing such tests, by exploring the effect of numerous hypothetical failures of microservices without actually injecting faults or failures. Clearly, this is traded off by the accuracy of the estimate, that when using the causal surrogate model is a *prediction* of the effect, ultimately depending on the model accuracy.

To show how interventions are performed, let us consider a chain of invocations as a sequence of microservices $M_1, \ldots, M_n$, where M_1 calls M_2 that in turn calls M_3, and so on till M_n. The chain in the SCM will have all the edges directed in the opposite direction, namely from M_i to M_{i-1}. The interventions are done as follows: for each node M_k in the chain with at least one out edge (we do not intervene on M_1 as it does not have effects on others) the engine intervenes on the model querying what would happen to the M_k's successors (i.e., *do they fail or not?*) if M_k exhibited an erroneous behavior or failed. In other words, we evaluate $P(M_i | do(M_k = fail))$ for each ith successor of M_k in the chain. The query on the model returns a value for each microservice in the chain corresponding to its expected behavior (HTTP status code) if M_k fails. From the output of the interventions, `uKnows` builds an output graph by drawing edges from the nodes which it intervened on to each node for which a failure is predicted. The result of these interventions is a graph highlighting the microservices causally involved in erroneous behaviors (functional testing) or failures (robustness testing). These are

[5] py-causal v1.2.1, https://zenodo.org/record/3592985.

[6] FCI settings: testId = fisher-z-test, depth = −1, maxPathLength = −1, completeRuleSetUsed = False.

[7] Let X, Y, and Z be disjoint subsets of all vertexes in the DAG. Z *d-separates* X and Y just in case every path from a variable in X to a variable in Y contains at least one vertex X_i such that either X_i is a *collider* (i.e. the arrows converge on X_i in the path) and no descendant of X_i (including X_i) is in Z, or X_i is not a collider, and X_i is in Z.

the microservices that developers should focus on, since they cause erroneous behavior in other microservices in the system. For example, in Fig. 4 they are nodes M_2 and M_6. The interventions are done via `do-why` (Sharma et al., 2019), a Microsoft's library to perform inference on causal models.

5. Experimentation

We investigate how the features of MacroHive – black-box combinatorial test generation (`uTest`), gray-box monitoring (`uSauron`), causal inference (`uKnows`) – support the testing process for MSA. Experiments have been conducted assessing the performance in terms of coverage, fault detection, and cost. MacroHive is compared to the state-of-the-art black-box testing tools EvoMaster (Arcuri, 2021), ResTestGen (Corradini et al., 2022), RESTler (Atlidakis et al., 2019), bBOXRT (Laranjeiro et al., 2021).

The other tools mentioned in Section 2 – RESTest (Martin-Lopez et al., 2020), QuickRest (Karlsson et al., 2020), and the Eclipse plugin (Ed-douibi et al., 2018) – are either not available or cannot be run (due to internal errors or incompatibility with the case study).

For repeatability and reproducibility, we make available MacroHive code for running experiments.[8]

5.1. Experimental subjects

The main experimental subject is *TrainTicket*, a well-known open-source MSA benchmark, composed of 41 microservices (Zhou et al., 2018b). This MSA has been extensively used in previous research and is considered representative of a real-world MSA (Zhou et al., 2021; Ji et al., 2020; Zhou et al., 2019; Li et al., 2021; Cortellessa et al., 2022; Hou et al., 2019; Zhou et al., 2018a; Walker et al., 2021; Wu et al., 2021; Liu et al., 2020). It is worth noting that other usual benchmarks in the related literature, such as *Sock Shop*[9] (6 microservices), *Pet Clinic*[10] (3 microservices), *FTGO*[11] (7 microservices), *Piggy Metrics*[12] (3 microservices), used in Corradini et al. (2021a), Rahman and Lama (2019), Joseph and Chandrasekaran (2020) and Chen et al. (2016, 2014), are inadequate for testing MSA: they are (small) collections of microservices that do not interact with each other — in this sense, they are not realistic MSA. For this reason, these subjects are not suitable for the experiments. However, we report the results also for *SockShop* and *FTGO*.

5.2. Tests generation

In experiments, MacroHive generates *2-way* test suites, as follows:

- *For functional testing*: test suite with *valid* inputs;
- *For robustness testing*: test suite with *valid* and *invalid* inputs.

The comparison with RESTler, RestTestGen and bBOXRT is on tests with both types of inputs; EvoMaster generates tests only with valid inputs.

The tools are run 10 times each on any of the 34 externally accessible microservices, out of the 41 in *TrainTicket*.[13] 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 their default settings or, when available, with the configuration that was shown to yield the best performance. For instance, RESTler has been configured with the BFS-cheap algorithm, which achieves best results with low time budgets (Atlidakis et al., 2019). When available, the maximum time budget is set to 150 s, namely 10 times the average time of a MacroHive testing session. We use *Burp Suite* to collect tests input and output (Portswigger, 2021). Then, we export the logs and feed them in Restats (Corradini et al., 2021b), a tool to compute coverage metrics.

Table 2
Coverage metrics.

Coverage metric	Description
Status Code Class	Status Code Class (SCC) coverage is 100% when the test suite triggers both correct and erroneous status codes. SCC coverage is 50% if test suite triggers only status codes belonging to the same class (correct or erroneous). Codes 2xx represent correct executions; codes 4xx and 5xx represent erroneous executions.
Status Code	Status Code (SC) is the ratio of the number of obtained status codes to the total number of status codes documented in the specification, for all operations. SC coverage is 100% when, for each operation, the test suite covers all the status codes.

5.3. Research questions

RQ1 How do the different tools support the MSA testing process?
This research question aims to qualitatively investigate the extent to which the compared tools automate or support the MSA testing activities described in Section 3.

RQ2 How does MacroHive perform in supporting functional testing, compared to EvoMaster?
Assuming that the test objective is to find unexpected behavior of the MSA under test when tests with valid inputs are generated, we consider EvoMaster as a term of comparison. For fair comparison, we consider the metrics status code coverage and status code class coverage (Table 2). First, we compare the results of both approaches considering the system as black-box, then we show the advantages using MacroHive due to the internal microservices monitoring and inference.

RQ3 How does MacroHive perform in supporting robustness testing, compared to RestTestGen, bBOXRT, and RESTler?
Assuming that the test objective is to expose failures of the MSA under test, we consider, for each tool, the number of failures and the failure rate. As before, we compare the results considering the system as a black-box, and then highlight the advantages of the MacroHive's internal microservices monitoring and inference.

5.4. RQ1: support for the MSA testing process

To answer RQ1 we accurately examined all the compared tools. Our findings are summarized in Table 3. All tools are able to automatically generate tests from the specification. A relevant difference concerns phase ①: MacroHive and EvoMaster automatically collect the API specification, given the addresses of microservices that expose them,[14] while the other tools require them as input (as JSON files). In other words, except for MacroHive and EvoMaster, testers have to manually retrieve the microservices specification and feed them to the tools. Furthermore, MacroHive can handle many microservices at the same time, generating a test suite for the entire MSA, while all the other tools can generate tests for just one API at a time (testing session). Actually, rather than testing the MSA as a whole, they regard the testing sessions of different microservices as independent from each other.

[8] MacroHive is available at: https://github.com/uDEVOPS2020/MacroHive.
[9] Sock Shop, https://microservices-demo.github.io/.
[10] PetClinic, https://github.com/spring-projects/spring-petclinic.
[11] FTGO, https://github.com/microservices-patterns/ftgo-application.
[12] Piggy Metrics, https://github.com/sqshq/piggymetrics.
[13] We found 7 internal microservices in *TrainTicket* (*user-*, *authentication-*, *verification-code-*, *ticket-office-*, *avatar-*, *wait-order-*, and *news-service*).
[14] OpenAPI allows to retrieve a specification via the Swagger user interface, reachable online with an HTTP request directed to the microservice exposing it.

Table 3
RQ1: Tools support for the phases of the MSA testing process in Fig. 1.

Phases		MacroHive	EvoMaster	RestTestGen	RESTler	bBOXRT
Test objective	*functional testing*	✓	✓			
	robustness testing	✓		✓	✓	✓
API specification collection	*automatic*	✓	✓			
	manual	✓		✓	✓	✓
Test specification definition	*API parsing*	✓	✓	✓	✓	✓
	input space partitioning	✓				✓
	dictionary definition			✓	✓	
	test oracle definition	✓	✓	✓	✓	✓
Test suite generation	*stateless*	✓				
	stateful		✓	✓	✓	✓
Test execution	*requests execution*	✓	✓	✓	✓	✓
	black-box monitoring	✓	✓	✓	✓	✓
	gray-box monitoring	✓				
Test reporting	*test outcome*	✓	✓	✓	✓	✓
	computed statistics	✓	✓	✓	✓	
	inferred results	✓		✓		

6. Results

As for test specification definition (phase ②), the tools have a similar definition of the test oracle: all tests returning a response code belonging to a 5xx class are considered as failed. In MacroHive, EvoMaster, and RestTestGen the test oracle considers as failed also those tests with responses that do not comply to HTTP response codes listed in the API specification. RestTestGen and EvoMaster consider failed also tests with responses that do not match the schema (i.e., expected response body) in the API specification.

As for test generation (phase ③), three out of five tools (EvoMaster, RestTestGen, and RESTler) are stateful. EvoMaster performs random testing, adding heuristics to maximize the HTTP response code coverage. RestTestGen and RESTler leverage data and operation dependencies inferred from previously executed tests to choose, respectively, the parameters' values and requests' sequences. bBOXRT and MacroHive extract input classes from the API specification in a stateless manner, and then generate inputs for the classes: bBOXRT executes valid requests with random values compliant to the OpenAPI specification, and then mutates them to trigger erroneous behaviors; MacroHive uses a combinatorial strategy, generating random values within the boundaries of the input classes (valid and/or invalid).

All the tools are able to collect request–response couples exchanged with the edge microservices to execute tests (phase ④). MacroHive is able to collect also the request–responses couples of internal microservices thanks to the sidecar monitoring infrastructure (`uSauron`).

As for reporting (phase ⑤), RestTestGen, bBOXRT, and EvoMaster show results for individual requests, thus needing other tools to compute metrics. RestTestGen reports also the operation dependency graph inferred from the execution, while EvoMaster reports the percentage of consumed budget and the fitness (number of HTTP codes covered). Besides requests' results, RESTler provides basic statistics like number of valid/invalid sequences of operations, total object creations, and the set of reproducible bugs. MacroHive reports, in addition to the HTTP SCC coverage, basic statistics both for generation (number of different paths and method, and number of tests generated) and execution (number of executed tests, successes, failures, unique failures, and details on response times). It also reports gray-box coverage metrics of internal microservices, failure chains, and dependencies. Finally, with its causal inference engine, it provides automatic reporting and identification of the critical parts of the system under test, considering the microservices interaction.

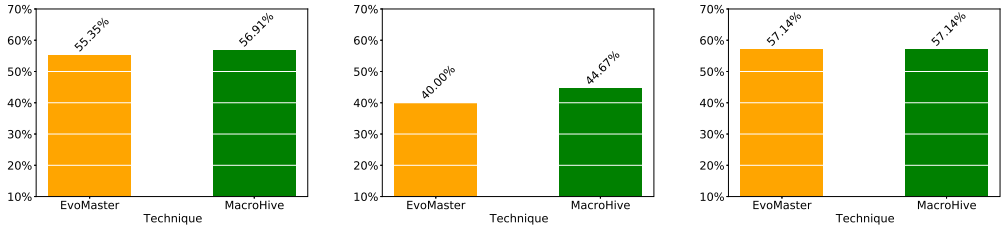
6.1. RQ2: Functional testing

Coverage comparison. EvoMaster and MacroHive compute the Status Code Class (SCC) and Status Code (SC) coverage reached by the respective test suites. On the output of the 10 repetitions, we perform a Wilcoxon rank sum test to statistically compare the results. The null hypothesis of the two coverage datasets coming from the same population is rejected with *p-value* = 8.08E−7 for SCC and *p-value* = 1.37E−11 for SC for *TrainTicket*, while it is not rejected for *Sock Shop* and *FTGO* since the *p*-value is always greater than 0.05. Rejecting the null hypothesis ensures that the difference between the two approaches is statistically significant, allowing us to state which is the best one. On the contrary, the impossibility to reject the null hypothesis implies that the techniques are equivalent.

Fig. 5 shows the average Status Code Class coverage of edge microservices per subject, while Fig. 6 the SCC coverage per *TrainTicket* edge microservice. 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. Overall, MacroHive slightly outperforms EvoMaster.

For *TrainTicket*, SCC values of MacroHive are always greater than 50%, meaning that for each microservice it obtains at least one documented status code for each method, on average.

Fig. 7 compares the average Status Code coverage of EvoMaster and MacroHive per subject, while Fig. 8 compares their SC coverage per



(a) *TrainTicket* (b) *Sock Shop* (c) *FTGO*

Fig. 5. RQ2: Status Code Class coverage per subject.

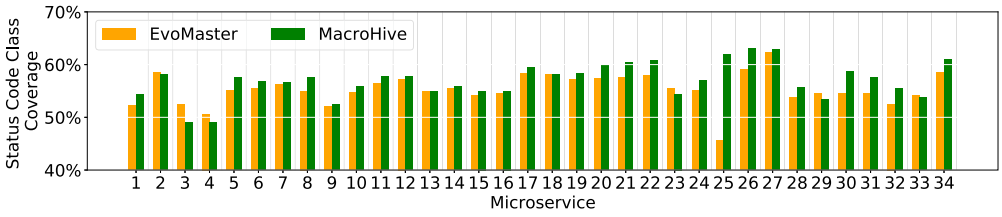


Fig. 6. RQ2: Status Code Class coverage per edge microservice (*TrainTicket*).

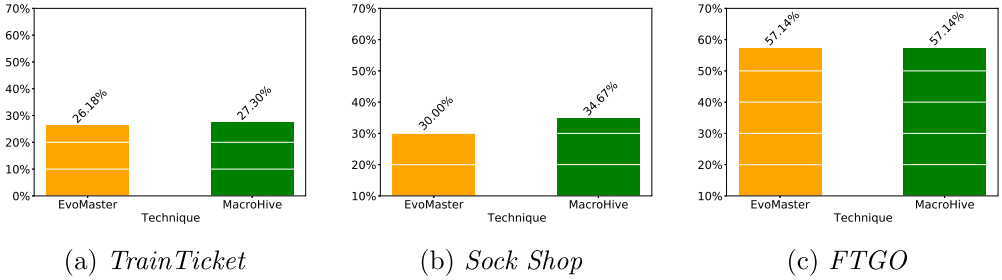


Fig. 7. RQ2: Status Code coverage per subject.

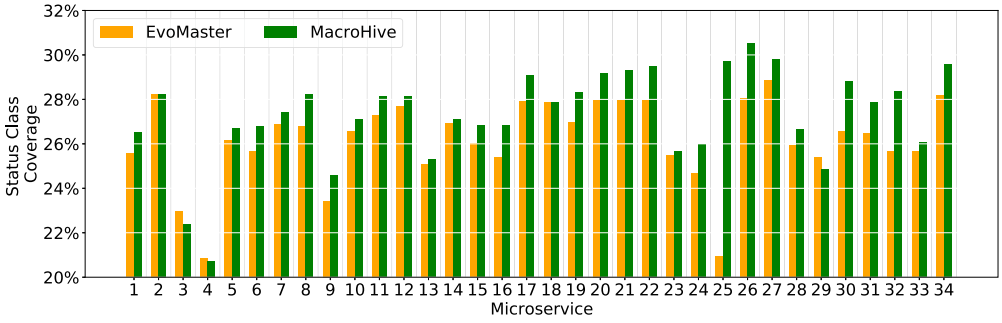


Fig. 8. RQ2: Status Code coverage per edge microservice (*TrainTicket*).

TrainTicket edge microservice. SC values greater than 25% mean that the techniques obtain at least a quarter of the documented codes, while the SC coverage is 50% when all the detected codes belong just to one class. For instance, assume a method with 100 different status codes specified and just one of them is detected. The SC would be 1%, the SCC 50%.

Again, MacroHive coverage slightly outperforms EvoMaster, despite the latter uses heuristics to improve the coverage obtained in a testing session.

Internal coverage. With its gray-box sidecar monitoring, MacroHive allows evaluating the coverage of internal microservices when invoked by other microservices. The following investigation aims to evaluate to what extent MacroHive exercises internal microservices through edge requests. For this analysis we consider only TrainTicket, as the other MSA subjects, as reported in Section 5.1, lack internal interacting microservices.

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 $p_r = \langle M_0, M_1, \ldots, M_{L_r} \rangle$ 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. For *TrainTicket*, the biggest dependency level of all edge microservices is 5.

Fig. 9 shows the internal SCC coverage achieved by MacroHive for each *TrainTicket* edge microservice. Each plot shows the coverage achieved through the invocation of the ith microservice (with $i \in (1, 34)$, namely the edge microservices of *TrainTicket*) at a certain level. 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 microservices.[15]

We see that for microservice 13 (Fig. 9a) and 9 (Fig. 9b), MacroHive achieves an internal microservices coverage greater than 80% (for microservices invoked at levels 1 and 2 of the invocation chain), although the SCC at the edge microservice is lower than 26% and 24%, respectively. These two microservices have the longest invocation chains (microservice 13 reaches level 4 and microservice 9 reaches level 5). The Figures also show that two edge services with similar edge-level coverage can conceal very different internal coverage patterns — information that can help better focus testing efforts.

Inference. MacroHive provides automatic identification of causal relationships between erroneous behaviors (i.e., 4xx and 5xx HTTP status

[15] SCC is preferred to SC for working in the absence of the API specification of internal microservices. It is computed assuming that each method has response codes 2xx, 4xx, and 5xx (see Table 2).

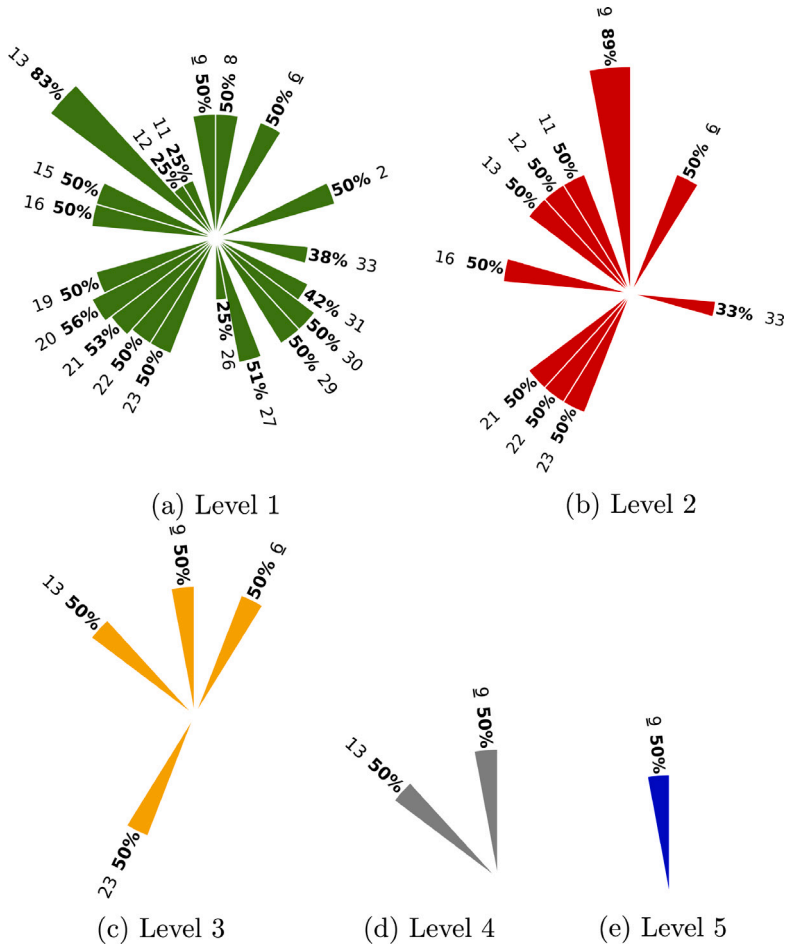


Fig. 9. RQ2: Status Code Class coverage per microservices level.

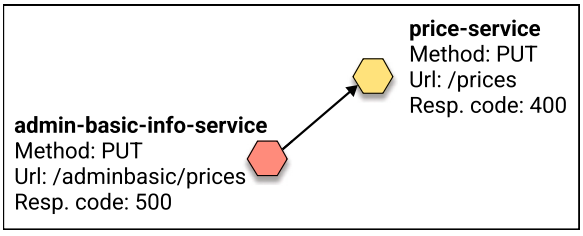


Fig. 10. RQ2: Example of erroneous behavior.

codes) between different microservices. An example manually extracted from traces is shown in Fig. 10: the service under test `admin-basic-service` sends a request to `price-service`, which responds with a code 400 (bad request), and this causes a code 500 in the first one.

MacroHive extracts a causal model from observational data, and then queries the model with interventions, such as: *"What happens to other microservices M_i $(i = 1, \dots, k-1)$ if one microservice M_k in a chain behaves erroneously?"* (formally: $P(M_i|do(M_k = error))$). Answering this kind of question allows to evaluate the impact of erroneous behavior of microservices on others and to extract a graph (Fig. 11), that depicts cause–effect relations among microservices. This output gives testers rapid and automatic feedback on the most critical parts of the MSA. Note that, among other relations, the graph also identified the situation shown in Fig. 10.

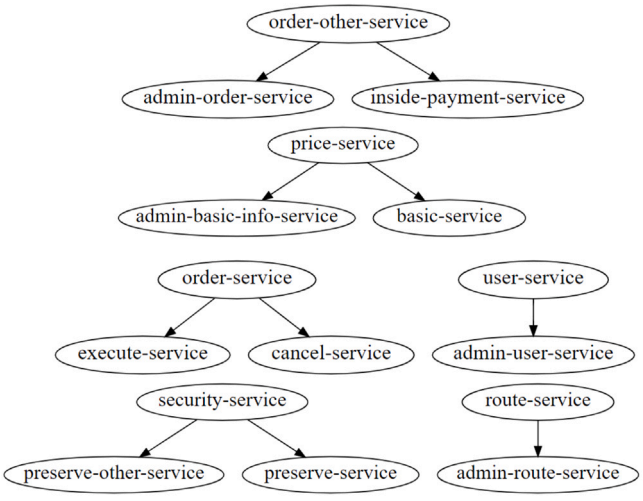
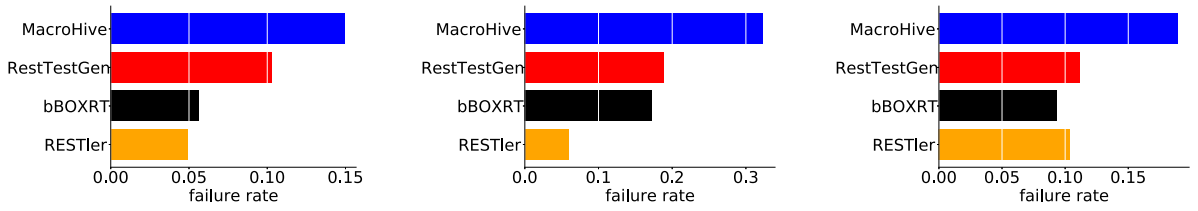


Fig. 11. RQ2: Erroneous behaviors cause–effect relations.

6.2. RQ3: Fault detection

Failure rate and unique failures. For a quantitative analysis of failures, we ran the tools (10 repetitions), and collected the request–response couples with *burp*, then evaluated as failing (5xx codes only) or not by a test oracle.



(a) *TrainTicket* (b) *Sock Shop* (c) *FTGO*

Fig. 12. RQ3: Average failure rate per subject.

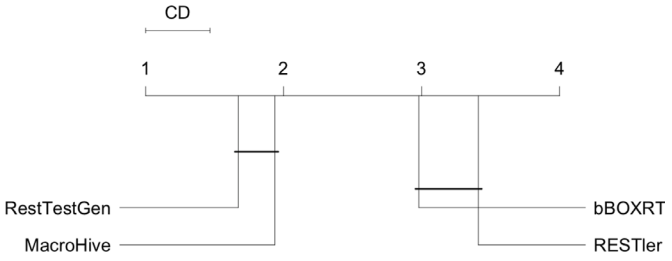


Fig. 13. RQ3: Critical differences of failure rate of compared tools.

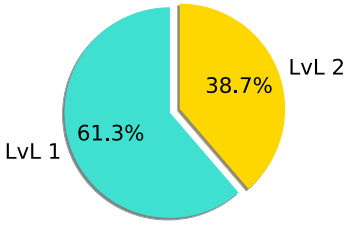


Fig. 14. RQ3: Distribution of internal failures across microservices levels.

Table 4
RQ3: Comparison of MacroHive to robustness testing tools.

Tool	Per microservice			
	Failure rate	# of tests	# of failures	# of unique failures
MacroHive	14.9%	266.1	42.5	7.6
ResTestGen	10.3%	5933.1	536.2	13.1
bBOXRT	5.6%	4695.6	271.0	10.7
RESTLer	4.9%	4548.6	213.7	7.9

Fig. 12 reports the Average Failure Rate (AFR), namely the ratio between the number of failures detected and the number of executed tests, averaged over edge microservices and repetitions. The Friedman test is run with a level of significance $\alpha = 0.05$. The test returns *p-value* < 2.2E−16 for *TrainTicket* and *p-value* < 3.7E−2 for *Sock Shop*, rejecting the null hypothesis that failure rate values do not significantly differ. The differences for FTGO are not significant since the *p-value* is greater than 0.05. We perform Dunn's test to detect which pairs exhibit a significant difference in the failure rates obtained on the *TrainTicket* and *Sock Shop* applications. While for *Sock Shop* there is only one significant difference between MacroHive and RESTler (*p-value* = 0.029), for *TrainTicket* the techniques show different behaviors, as shown by the plot of the critical difference in Fig. 13. While bBOXRT and RESTler have statistically significantly lower AFR, values for MacroHive are similar to ResTestGen. However, MacroHive exhibits the best average AFR, with almost 15% of generated tests exposing failures.

Table 4 reports number of tests, of failures and of unique failures (average over microservices). MacroHive stateless combinatorial technique features a much lower number of tests than the other tools. These generate at least 17 times the number of tests of MacroHive, trying to cover parameters/operations dependencies through information collected during test execution.

For a finer analysis of detected failures and to avoid bias due to the number of executed tests, we further evaluated the number of failed methods (*unique failures*), so as multiple failures of the same method to count as one. This is reported for each tool in the last column of Table 4. RestTestGen and bBOXRT show the best results on average. They found respectively 13.1 and 10.7 different failures per microservice. RESTler and MacroHive behave worse, respectively with 7.9 and 7.6 failures. A deeper manual inspection of results pointed out that RestTestGen and bBOXRT were particularly effective in finding failures in microservices with few (or none) dependencies. For instance, they found for the `order-service` and `travel-service` (the microservices with the highest number of failures) respectively {22, 17.6} and {17, 14.6} against {14, 10} of both MacroHive and RESTler. The first microservice has zero dependencies, making it easier for stateful tools to generate valid and invalid input to better exercise it, and although the second has 3 dependencies, the failures RestTestGen and bBOXRT were able to find (and MacroHive and RESTler not) were mostly related to not-dependent requests. In summary, the better results are due to: (*i*) better ability to find intra-microservice parameter dependencies (in not-dependent microservices); (*ii*) better ability to activate faults with input mutations.

Internal failures. A unique feature of MacroHive is the detection of internal failures. Table 5 reports the results. The subject here is *TrainTicket*. We notice that, on average on repetitions, 33.7 additional failures are detected with respect to the edge-level failures (almost 1 failure per microservice). Such internal failures are distributed among level-1 and level-2 dependencies (Fig. 14). We did not find failures at levels deeper than 2.

Internal failures are the trickiest failures to find, and remained undetected by other tools. In addition, through the identification of these failures, MacroHive was able to highlight internal failure propagation chains, as well as possible masking effects. Fig. 15 shows two situations spotted by MacroHive in *TrainTicket*. Fig. 15a shows a propagated failure: two failing internal services, `security-service` and `order-service`, cause the edge microservice `preserve-service` to fail. Without MacroHive, the propagated failures would have been associated with edge microservices, while an internal failure is the real cause. Fig. 15b shows a masked failure: a test passed, despite a failure occurring in the internal microservice `order-service`. Masked failures are not detected by the other compared tools, as they never reach the edge microservice. This kind of failure can silently corrupt the state of the MSA and manifest unexpectedly in operation (Wang et al., 2020; Jagadeesan and Mendiratta, 2020; Mathur, 2020). Undoubtedly, some of these failures may have been tolerated by the designed fault tolerance mechanisms, while others may have simply been prevented from spreading by the program control flow; engineers want to understand the causes of the microservice failure in both scenarios.

A further MacroHive feature is the identification of failures occurred in different microservices, yet caused by a common faulty method in a service. Fig. 16 shows one such case in *TrainTicket*,

Table 5
RQ3: Detailed metrics about failures detected by MacroHive.

MacroHive	AFR	# of internal failures	# of unique Failures	Edge failures	Internal failures
Total	/	713.6	292.0	258.3	33.7
Per microservice	38.2%	20.9	8.6	7.6	0.9

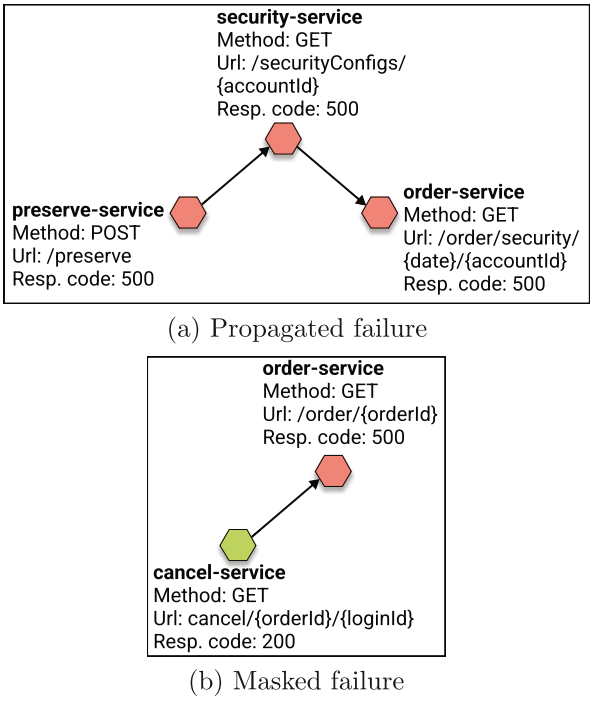


Fig. 15. RQ3: Examples of failing internal microservices.

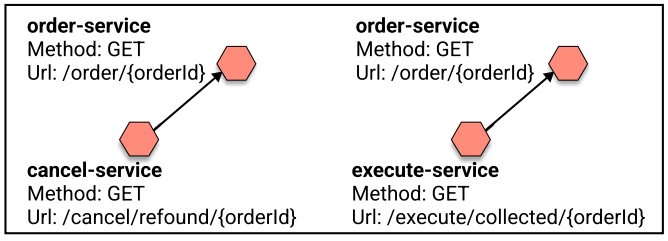


Fig. 16. RQ3: A case of common cause failure in the *TrainTicket* subject.

whose services `cancel` and `execute` failed similarly. This information would help a tester to spot that the common cause is the method *GET /order/{orderId}* of `order-service`.

Table 6 reports the average number of propagated and masked failures among all microservices detected by MacroHive, together with the average number of executed tests and failures observed at the edge. MacroHive exposes 328.7 propagated failures and 1 masked on average. In particular, it identifies 18 unique failure propagation chains and exactly 1 masked failure coming from the microservice `cancel-service` (see Fig. 15b).

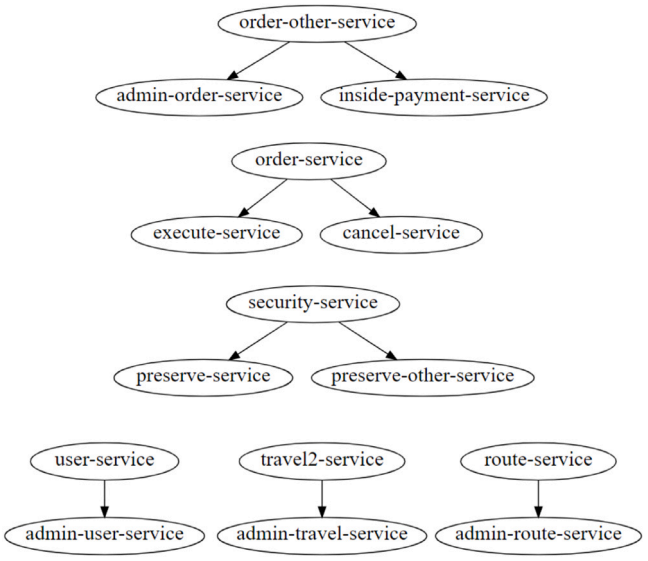


Fig. 17. RQ3: Failures cause–effect relations discovered for *TrainTicket* microservices.

Inference. MacroHive `uKnows` service establishes whether the failures of any two services are causally related or not. To establish a *systematic* causal relation between, it is not sufficient to observe one failing trace (which could be due, for instance, to a rarely occurring condition); there could also well be relations among a failing microservice and microservices non-adjacent in the SDG, that do not manifest from the SDG (e.g., because intermediate microservices could mask the root microservice failure).

The causal inference engine identifies the microservices whose failures have a systematic effect on other microservices, thus needing particular attention, Testers benefit from the inference results in two ways, as they can discover: (*i*) the most failure-prone microservices in interactions with others (nodes that are less robust to failures of invoked microservices); (*ii*) those microservices whose failure systematically induce the failure of others.

Fig. 17 shows the cause–effect failure relationships between microservices found by MacroHive. The arrows in the graphs represent causal relations between microservices failure (they start from the service causing the failure). MacroHive found that 10 out of the 18 failure chains (Table 6) were propagated failures (a systematic causal relation is detected between internal and edge failures). MacroHive is able to decompose propagation chains and to identify causality on a subset of them. For instance, it detected a level-3 failure chain (`order-service` to `security-service` and `security-service` to `preserve-service`) but highlighted cause–effect relations between two of them (`security-service` to `preserve-service`).

Table 6
RQ3: Number of propagated and masked failures.

Tool	Executed tests	Edge failures	Propagated failures	Unique propagated failures	Masked failures
MacroHive	4462.6	992	328.7	18	1

6.3. Threats to validity

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

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 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.

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

External validity. The use of only *TrainTicket* for the gray-box analysis hinders generalization. We reported the results with two further subjects; while they confirm MacroHive performance at edge level, by their nature (limited internal microservices) they could not be used for experiments in a gray-box perspective. Finding real-world MSA subjects is a known problem (Zhou et al., 2018b); we are tackling it by collaborating with industry in ongoing projects.

7. Conclusions

MacroHive is a framework for generation, execution and gray-box investigation of the results of functional and robustness tests for microservice architectures, with a high degree of automation. It features combinatorial test case generation, an infrastructure for execution and monitoring of interactions, and a causal inferential engine, giving testers insights about internal coverage, internal failing services, and causal relations between failures.

Apart from the combinatorial test case generation, MacroHive peculiarities with respect to existing tools are in that it provides coverage and failure information not only for edge, but also for MSA internal microservices, and it identifies causal relationships in observed chains of microservices failures. The availability of a causal model allows engineers to query the model for additional tasks, such as evaluating alternative configurations/deployments, comparing architectural solutions, and placing fault tolerance mechanisms where they are actually more required.

The experiments show that MacroHive is effective in achieving high coverage, as well as in detecting a variety of kinds of failures (both edge and internal). As for functional testing, it generates tests with edge-level coverage comparable (slightly better) to the state-of-the-art tool EvoMaster (black-box version), but enriched with useful insights about the coverage of internal microservices. This increases the tester's confidence in the appropriateness of the test suite. For robustness testing too, the experiments show that the combinatorial test generator exhibits coverage performance comparable to other tools at the edge level, but achieves a superior failure rate.

The cost of the technique is paid mostly in terms of overhead since it requires deploying a proxy for each microservice to monitor. 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 (Li et al., 2019). The delay due to proxies is the additional time incurred to forward a request or the corresponding response. We measured this delay as 1 ± 0.5 ms (median and semi inter-quartile range over all microservices), with the microservices response time equal to 7 ± 2.5 ms. As this kind of test can be executed in a staging environment, the overhead does not impact the MSA in production.

MacroHive is a flexible framework, whose components are themselves microservices, allowing easy plug in of other tests generators, and of other monitoring or inferential services as well. Indeed, as future step, we plan to integrate other test generators (e.g., with search-based testing strategies), and to exploit the causal model for causality-driven tests generation.

CRediT authorship contribution statement

Luca Giamattei: Conceptualization, Methodology, Investigation, Software, Formal analysis, Validation, Writing, Visualization. **Antonio Guerriero:** Conceptualization, Methodology, Investigation, Formal analysis, Validation, Writing, Visualization. **Roberto Pietrantuono:** Methodology, Supervision, Funding acquisition, Project administration, Writing, Visualization. **Stefano Russo:** Methodology, Supervision, Funding acquisition, Resources, Writing, Visualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

We have shared the link to our code in the paper.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 871342 "uDEVOPS".

References

Arcuri, A., 2019. RESTful API automated test case generation with EvoMaster. ACM Trans. Softw. Eng. Methodol. 28 (1), http://dx.doi.org/10.1145/3293455.

Arcuri, A., 2021. Automated black- and white-box testing of restful apis with evomaster. IEEE Softw. 38 (3), 72–78. http://dx.doi.org/10.1109/MS.2020.3013820.

Atlidakis, V., Godefroid, P., Polishchuk, M., 2019. Restler: Stateful REST API fuzzing. In: IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 748–758. http://dx.doi.org/10.1109/ICSE.2019.00083.

Bertolino, A., De Angelis, G., Guerriero, A., Miranda, B., Pietrantuono, R., Russo, S., 2020. DevOpRET: Continuous reliability testing in DevOps. J. Softw.: Evol. Process 35 (3), http://dx.doi.org/10.1002/smr.2298, e2298 smr.2298.

Burns, B., Oppenheimer, D., 2016. Design patterns for container-based distributed systems. In: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16). USENIX Association, Denver, CO, pp. 1–6, available at: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns.

Chen, T., Shang, W., Hassan, A.E., Nasser, M., Flora, P., 2016. CacheOptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications. In: Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). ACM, pp. 666–677. http://dx.doi.org/10.1145/2950290.2950303.

Chen, T., Shang, W., Jiang, Z.M., Hassan, A.E., Nasser, M., Flora, P., 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In: 36th International Conference on Software Engineering (ICSE). ACM, pp. 1001–1012. http://dx.doi.org/10.1145/2568225.2568259.

Corradini, D., Zampieri, A., Pasqua, M., Ceccato, M., 2021a. Empirical comparison of black-box test case generation tools for restful apis. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp. 226–236. http://dx.doi.org/10.1109/SCAM52516.2021.00035.

Corradini, D., Zampieri, A., Pasqua, M., Ceccato, M., 2021b. Restats: A test coverage tool for RESTful APIs. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 594–598. http://dx.doi.org/10.1109/ICSME52107.2021.00063.

Corradini, D., Zampieri, A., Pasqua, M., Viglianisi, E., Dallago, M., Ceccato, M., 2022. Automated black-box testing of nominal and error scenarios in restful apis. Softw. Test. Verif. Reliab. 32 (5), 1–33. http://dx.doi.org/10.1002/stvr.1808.

Cortellessa, V., Di Pompeo, D., Eramo, R., Tucci, M., 2022. A model-driven approach for continuous performance engineering in microservice-based systems. J. Syst. Softw. 183, 111084. http://dx.doi.org/10.1016/j.jss.2021.111084.

de Camargo, A., Salvadori, I., Mello, R., Siqueira, F., 2016. An architecture to automate performance tests on microservices. In: Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services. ACM, pp. 422–429. http://dx.doi.org/10.1145/3011141.3011179.

Ed-douibi, H., Cánovas Izquierdo, J.L., Cabot, J., 2018. Automatic generation of test cases for REST APIs: A specification-based approach. In: 22nd International Enterprise Distributed Object Computing Conference (EDOC). IEEE, pp. 181–190. http://dx.doi.org/10.1109/EDOC.2018.00031.

Ghani, I., Wan-Kadir, W., Mustafa, A., Imran Babir, M., 2019. Microservice testing approaches: A systematic literature review. Int. J. Integr. Eng. 11 (8), 65–80. http://dx.doi.org/10.30880/ijie.2019.11.08.008.

Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S., 2022. Automated grey-box testing of microservice architectures. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 640–650. http://dx.doi.org/10.1109/QRS57517.2022.00070.

Glymour, C., Zhang, K., Spirtes, P., 2019. Review of causal discovery methods based on graphical models. Front. Genet. 10, http://dx.doi.org/10.3389/fgene.2019.00524.

Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M.K., Sekar, V., 2016. Gremlin: Systematic resilience testing of microservices. In: IEEE 36th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp. 57–66. http://dx.doi.org/10.1109/ICDCS.2016.11.

Hou, X., Liu, J., Li, C., Guo, M., 2019. Unleashing the scalability potential of power-constrained data center in the microservice era. In: Proceedings of the 48th International Conference on Parallel Processing (ICPP). ACM, http://dx.doi.org/10.1145/3337821.3337857.

Hu, L., Wong, W., Kuhn, D., Kacker, R., 2020. How does combinatorial testing perform in the real world: an empirical study. Empir. Softw. Eng. 25, http://dx.doi.org/10.1007/s10664-019-09799-2.

Jagadeesan, L., Mendiratta, V., 2020. When failure is (not) an option: Reliability models for microservices architectures. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, pp. 19–24. http://dx.doi.org/10.1109/ISSREW51248.2020.00031.

Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S., 2018. Microservices: The journey so far and challenges ahead. IEEE Softw. 35 (3), 24–35. http://dx.doi.org/10.1109/MS.2018.2141039.

Ji, S., Wu, W., Pu, Y., 2020. Multi-indicators prediction in microservice using granger causality test and attention LSTM. In: 2020 IEEE World Congress on Services (SERVICES). IEEE, pp. 77–82. http://dx.doi.org/10.1109/SERVICES48979.2020.00030.

Joseph, C., Chandrasekaran, K., 2020. IntMA: Dynamic interaction-aware resource allocation for containerized microservices in cloud environments. J. Syst. Archit. 111, 101785. http://dx.doi.org/10.1016/j.sysarc.2020.101785.

Karlsson, S., Čaušević, A., Sundmark, D., 2020. QuickREST: Property-based test generation of OpenAPI-Described RESTful APIs. In: IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, pp. 131–141. http://dx.doi.org/10.1109/ICST46399.2020.00023.

Laranjeiro, N., Agnelo, J., Bernardino, J., 2021. A black box tool for robustness testing of REST services. IEEE Access 9, http://dx.doi.org/10.1109/ACCESS.2021.3056505.

Lei, Q., Liao, W., Jiang, Y., Yang, M., Li, H., 2019. Performance and scalability testing strategy based on kubemark. In: 2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA). IEEE, pp. 511–516. http://dx.doi.org/10.1109/ICCCBDA.2019.8725658.

Lewis, J., Fowler, M., 2014. Microservices - a definition of this new architectural term. available at: http://martinfowler.com/articles/microservices.html.

Li, Z., Chen, J., Jiao, R., Zhao, N., Wang, Z., Zhang, S., Wu, Y., Jiang, L., Yan, L., Wang, Z., Chen, Z., Zhang, W., Nie, X., Sui, K., Pei, D., 2021. Practical root cause localization for microservice systems via trace analysis. In: 2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS). IEEE, pp. 1–10. http://dx.doi.org/10.1109/IWQOS52092.2021.9521340.

Li, W., Lemieux, Y., Gao, J., Zhao, Z., Han, Y., 2019. Service mesh: Challenges, state of the art, and future research opportunities. In: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE). IEEE, pp. 122–127. http://dx.doi.org/10.1109/SOSE.2019.00026.

Liu, P., Xu, H., Ouyang, Q., Jiao, R., Chen, Z., Zhang, S., Yang, J., Mo, L., Zeng, J., Xue, W., Pei, D., 2020. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In: 31st International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 48–58. http://dx.doi.org/10.1109/ISSRE5003.2020.00014.

Long, Z., Wu, G., Chen, X., Cui, C., Chen, W., Wei, J., 2020. Fitness-guided resilience testing of microservice-based applications. In: IEEE International Conference on Web Services (ICWS). IEEE, pp. 151–158. http://dx.doi.org/10.1109/ICWS49710.2020.00027.

Ma, S., Fan, C., Chuang, Y., Lee, W., Lee, S., Hsueh, N., 2018. Using service dependency graph to analyze and test microservices. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 02. IEEE, pp. 81–86. http://dx.doi.org/10.1109/COMPSAC.2018.10207.

Martin-Lopez, A., Segura, S., Ruiz-Cortés, A., 2019. Test coverage criteria for RESTful web APIs. In: Proc. of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (a-TEST). ACM, pp. 15–21. http://dx.doi.org/10.1145/3340433.3342822.

Martin-Lopez, A., Segura, S., Ruiz-Cortés, A., 2020. RESTest: Black-box constraint-based testing of RESTful web APIs. In: Kafeza, E., et al. (Eds.), Service-Oriented Computing. In: Lecture Notes in Computer Science, vol. 12571, Springer, pp. 459–475. http://dx.doi.org/10.1007/978-3-030-65310-1_33.

Mathur, M., 2020. Leveraging Distributed Tracing and Container Cloning for Replay Debugging of Microservices (Ph.D. thesis). University of California, Los Angeles, https://escholarship.org/uc/item/7dp9q7j5.

Nogueira, A.R., Pugnana, A., Ruggieri, S., Pedreschi, D., Gama, J., 2022. Methods and tools for causal discovery and causal inference. WIREs Data Min. Knowl. Discov. 12 (2), e1449. http://dx.doi.org/10.1002/widm.1449.

Patel, A.R., Tyagi, S., 2022. The state of test automation in DevOps: A systematic literature review. In: Proceedings of the 2022 Fourteenth International Conference on Contemporary Computing, IC3-2022. ACM, pp. 689–695. http://dx.doi.org/10.1145/3549206.3549321.

Pearl, J., Mackenzie, D., 2018. The Book of Why: The New Science of Cause and Effect, first ed. Basic Books, Inc., USA.

Pezzè, M., Young, M., 2008. Software Testing and Analysis: Process, Principles, and Techniques. John Wiley & Sons.

Pietrantuono, R., Russo, S., Guerriero, A., 2018. Run-time reliability estimation of microservice architectures. In: 29th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 25–35. http://dx.doi.org/10.1109/ISSRE.2018.00014.

2021. Portswigger: Burp suite. https://portswigger.net/burp.

Rahman, J., Lama, P., 2019. Predicting the end-to-end tail latency of containerized microservices in the cloud. In: 2019 IEEE International Conference on Cloud Engineering (IC2E). IEEE, pp. 200–210. http://dx.doi.org/10.1109/IC2E.2019.00034.

Ramsey, J., Zhan, K., Glymour, M., Sanchez Romero, R., Huang, B., Ebert-Uphoff, I., Samarasinghe, S.M., Barnes, E.A., Glymour, C., 2018. TETRAD - A toolbox for causal discovery. In: 8th International Workshop on Climate Informatics.

Sharma, A., Kiciman, E., et al., 2019. DoWhy: A python package for causal inference. https://github.com/microsoft/dowhy.

Spirtes, P., Glymour, C., Scheines, R., 2001. Causation, Prediction, and Search, second ed. In: Adaptive Computation and Machine Learning, MIT Press, Cambridge, MA, USA.

Viglianisi, E., Dallago, M., Ceccato, M., 2020. RESTTESTGEN: Automated black-box testing of restful APIs. In: 13th International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 142–152. http://dx.doi.org/10.1109/ICST46399.2020.00024.

Walker, A., Laird, I., Cerny, T., 2021. On automatic software architecture reconstruction of microservice applications. In: Kim, H.e.a. (Ed.), Information Science and Applications. In: Lecture Notes in Electrical Engineering, vol. 739, Springer, pp. 223–234. http://dx.doi.org/10.1007/978-981-33-6385-4_21.

Wang, T., Zhang, W., Xu, J., Gu, Z., 2020. Workflow-aware automatic fault diagnosis for microservice-based applications with statistics. IEEE Trans. Netw. Serv. Manag. 17 (4), http://dx.doi.org/10.1109/TNSM.2020.3022028.

Waseem, M., Liang, P., Márquez, G., Di Salle, A., 2020. Testing microservices architecture-based applications: A systematic mapping study. In: 27th Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 119–128. http://dx.doi.org/10.1109/APSEC51365.2020.00020.

Wu, L., Tordsson, J., Elmroth, E., Kao, O., 2020. Microrca: Root cause localization of performance issues in microservices. In: NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium. IEEE, pp. 1–9. http://dx.doi.org/10.1109/NOMS47738.2020.9110353.

Wu, L., Tordsson, J., Elmroth, E., Kao, O., 2021. Causal inference techniques for microservice performance diagnosis: Evaluation and guiding recommendations. In: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). IEEE, pp. 21–30. http://dx.doi.org/10.1109/ACSOS52086.2021.00029.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., Ding, D., 2021. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. Softw. Eng. 47 (2), 243–260. http://dx.doi.org/10.1109/TSE.2018.2887384.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Liu, D., Xiang, Q., He, C., 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM, pp. 683–694. http://dx.doi.org/10.1145/3338906.3338961.

Zhou, X., Peng, X., Xie, T., Sun, J., Li, W., Ji, C., Ding, D., 2018a. Delta debugging microservice systems. In: 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 802–807. http://dx.doi.org/10.1145/3238147.3240730.

Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., Zhao, W., 2018b. Benchmarking microservice systems for software engineering research. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE-Companion), ICSE '18. ACM, pp. 323–324. http://dx.doi.org/10.1145/3183440.3194991.

Luca Giamattei (Ph.D. student) received the M.S. degree in computer engineering in 2021 from the Federico II University of Naples, Italy. He is currently a Ph.D. student in Information Technology and Electrical Engineering from the same university. His main research interests are in testing of DNN-enabled systems and distributed software systems. In this context, he collaborated in international projects. He published in international conferences in the field of software engineering and software testing.

Antonio Guerriero (Ph.D.) is Assistant Professor at Federico II University of Naples, Italy. He received a Ph.D. degree in Information Technology and Electrical Engineering from the same university in 2022. His main research interests are in testing of ML-based systems and distributed software systems. In this context, he collaborated on both national and international projects. He published in international conferences and journals in the field of software reliability, software engineering, and software testing.

Roberto Pietrantuono (Ph.D.) is Associate Professor at University of Naples Federico II. He is in the Dependable Systems and Software Engineering Research Team since 2007. His research interests focus on software testing and on dependability of software systems. He co-founded Critiware (www.critiware.com) a company working in critical systems engineering since 2011. He is involved in several projects and currently coordinates an MSCA RISE European project (uDevOps). He is senior member of IEEE and member of ACM.

Stefano Russo (Ph.D.) is Professor of Computer Engineering at Federico II University of Naples, Italy, where he teaches Software Engineering and Distributed Systems, and leads the DESSERT research group (www.dessert.unina.it). He (co-)authored over 190 papers in the areas of software testing, software aging, middleware technologies, mobile computing. He is Associate Editor of IEEE Trans. on Services Computing, and Senior Member of the IEEE.