Received: 15 February 2021 | Revised: 22 October 2021 | Accepted: 9 March 2022

DOI: 10.1002/smr.2452

SPECIAL ISSUE - TECHNOLOGY PAPER

Journal of Software: Evolution and Process WILEY

ExVivoMicroTest: ExVivo Testing of Microservices

Luca Gazzola[1] | Maayan Goldstein[2] | Leonardo Mariani[1] | Marco Mobilio[1] | Itai Segall[3] | Alessandro Tundo[1] | Luca Ussi[1]

[1]Department of Informatics Systems and Communication (DISCo), University of Milano-Bicocca, Milan, Italy

[2]Nokia Bell Labs, Tel Aviv, Israel

[3]Nokia Bell Labs, Murray Hill, New Jersey, USA

Correspondence
Alessandro Tundo, Department of Informatics Systems and Communication (DISCo), University of Milano-Bicocca, Milan, Italy.
Email: alessandro.tundo@unimib.it

Funding information
GAUSS National Research Project - PRIN 2015 Program, Grant/Award Number: 2015KWREMX; H2020 Learn Project - ERC Consolidator Grant 2014 Program, Grant/Award Number: 646867; H2020 NGPaaS project, Grant/Award Number: 761557

Abstract

Microservice-based applications consist of multiple services that can evolve independently. When a service must be updated, it is first tested with in-house regression test suites. However, the test suites that are executed are usually designed without the exact knowledge about how the services will be accessed and used in the field; therefore, they may easily miss relevant test scenarios, failing to prevent the deployment of faulty services. To address this problem, we introduce ExVivoMicroTest, an approach that analyzes the execution of deployed services at run-time in the field, in order to generate test cases for future versions of the same services. ExVivoMicroTest implements lightweight monitoring and tracing capabilities, to inexpensively record executions that can be later turned into regression test cases that capture how services are used in the field. To prevent accumulating an excessive number of test cases, ExVivoMicroTest uses a test coverage model that can discriminate the recorded executions between the ones that are worth to be turned into test cases and the ones that should be discarded. The resulting test cases use a mocked environment that fully isolates the service under test from the rest of the system to faithfully reply interactions. We assessed ExVivoMicroTest with the PiggyMetrics and Train Ticket open source microservice applications and studied how different configurations of the monitoring and tracing logic impact on the capability to generate test cases.

KEYWORDS
ex vivo testing, microservices, regression testing, software testing

1 | INTRODUCTION

The microservice architecture is an architectural style introduced to enable the organization of distributed applications as a collection of possibly stateless services, to achieve scalability, separation of concerns, and maintainability.[1] Microservices are widely used by companies such as Netflix, Amazon, and Facebook, to deliver multiple updates per minute.[2] Some of these companies[3] are even switching to the serverless computing paradigm, taking stateless functionality to the extreme. The microservices architecture is often exploited together with container-based technologies in the context of continuous deployment processes to quickly and iteratively update, deploy, and operate services.[4] While this is an extremely effective solution to master software evolution, as witnessed by its increasing adoption, it also introduces nontrivial quality issues, since verification and validation methods must efficiently deal with frequent updates.

The microservices evolution process requires testing every new service release in-house before the updated service can be deployed to the field. However, in-house testing is inevitably partial and often fails to validate scenarios that represent how services are actually used in the field. In fact, test cases (e.g., unit, integration, and system test cases) are designed by developers without exactly knowing neither the characteristics of the operational environment nor the behavior of the users.

For instance, a microservice to manage accounts might never be tested in-house for a sequence of operations that is the creation of an account followed by its immediate removal, assuming this behavior is unlikely to happen in reality. If an incentive to user registration is at some point introduced (e.g., as a consequence of an advertisement campaign), some users might inadvertently register by clicking on advertisements and then immediately removing the undesired account. This would generate field behaviors that were not anticipated and not adequately tested.

To timely consider the actual behavior of the services in operation also during in-house testing stages, it is possible to exploit ex vivo testing strategies.[5] *Ex vivo testing* uses information extracted from the field to augment existing test cases or produce new test cases that can be used to validate software upgrades consistently with field usages.[6,7] Indeed, ex vivo testing may limit the number of faults that escape in-house testing due to the lack of knowledge of the behavior of the software in operation, reducing the number of failures that are discovered late in the field, once services have been updated. Relevant applications of ex vivo testing include MapReduce applications,[6] autonomous vehicles,[8] and self-adaptive systems.[9]

This paper studies ex vivo testing applied to microservice architectures. The main objective of the presented technique is to collect representative executions from the field and to convert them into in-house test cases that can be executed to validate changes according to the actual usages of the services. The resulting approach is ExVivoMicroTest, illustrated in Figure 2. Our technique observes the input-output behavior of a microservice, records these interactions, and then synthesizes automatic test cases that replay the observed interactions and check the results produced by the service under test.

The design of ExVivoMicroTest includes several key capabilities: (1) the capability to sample observed interactions based on the likelihood of collecting interesting interactions not already covered in the existing test suite, (2) the capability to transform the data collected in the field into test cases that replay interactions and check the results produced by the service under test, (3) the capability to synthesize mocks that capture and check interactions between the service under test and the other services of the system, and (4) the capability to select test cases and influence the probability of collecting additional tests based on a representation of the behavioral space of the service under test. Note that the generation of mocks guarantees the full isolation of the testing procedure, which does not require running any other service beyond the service under test. ExVivoMicroTest targets microservice architectures as it assumes that services are stateless. Such assumption is realistic for the microservices applications as typically the services themselves are stateless and any state information required by such an application is stored externally.[1]

To avoid the generation of false alarms, since the behavior of some operations might be intentionally changed by the developers from one version to another, the test cases that stimulate certain operations can be filtered out before running ExVivoMicroTest. In fact, ExVivoMicroTest exploits the stateless nature of microservices to efficiently select and replay portions of the executions observed in the field.

We studied the trade-offs between multiple monitoring policies and different representations of the behavioral space, in terms of the capability to cover new interesting behaviors with test cases obtained from recorded interactions and the cost of obtaining these test cases. Results provide evidence of the scenarios that can be well addressed by ExVivoMicroTest based on their intrinsic level of variability: Regular and highly dynamic behaviors can be well addressed by ExVivoMicroTest, while scenarios with rare events might be more challenging to be effectively addressed.

This paper extends the former version of this work[10] in multiple ways:

- It defines a monitoring strategy that continuously captures requests to the service under test and exploits a tracing profile to decide when to activate and deactivate tracing, that is, when to start and stop saving every incoming and outgoing request for the service under test.
- It defines a behavioral space model that can be used to represent the set of diverse behaviors that might be covered with the tests extracted from the field.
- It defines an abstraction strategy to map concrete operations observed in the field into the behavioral space model.
- It defines a tracing profile update strategy to regularly update the probability of activation and deactivation of the tracing based on the already collected ex vivo tests.
- It experiments ExVivoMicroTest with PiggyMetrics[11] and Train Ticket[12] systems, discussing trade-offs between configurations.

The rest of the paper is organized as follows. Section 2 provides background information about the microservices architecture and introduces the PiggyMetrics system, which is used in the rest of the paper to illustrate the approach. Section 3 introduces ExVivoMicroTest. Sections 4 and 5 describe the steps performed in the field and in-house, respectively. Section 6 provides the implementation details. Section 7 presents empirical results. Section 8 discusses related work. Finally, Section 9 provides final remarks.

2 | BACKGROUND INFORMATION AND RUNNING EXAMPLE

This section presents the microservice and container technologies and introduces PiggyMetrics, a microservice-based application that we use to both illustrate and assess ExVivoMicroTest.

2.1 | Microservices and containers

Microservices are self-contained software units, typically implementing single business functions.[1] Applications composed of microservices are easy to maintain and evolve because their services can be changed independently, for instance, to accommodate new requirements and bug fixes.

Microservices are often coupled with containers,[13] which offer a standard way to create lightweight executable packages that include everything necessary to run services. In fact, each microservice is normally deployed into a different container, and applications composed of multiple services run over a set of interacting containers.

Containers can be used in cloud environments to obtain distributed applications that flexibly exploit the available computational resources, dynamically scaling up and down their components.[14] Scaling policies typically aim at maintaining quality of service without wasting resources. To facilitate autoscaling and other advanced features, microservices are often implemented as stateless services whose state information is stored externally. For instance, a microservice that manages accounts can be decoupled from its database and any other state information, to obtain a stateless service that can be easily replicated according to the workload. Stateless replica share state information through the (stateful) services and resources that manage it.

In this highly dynamic and distributed setting, the way the individual services are used in the field can be largely different with respect to the way the same services have been tested in house. The objective of ExVivoMicroTest is capturing these usages and automatically mapping them to a set of representative test cases that can be executed in house.

2.2 | Running example

In this paper, we use PiggyMetrics,[11] an open source system to deal with personal finances, to both describe and assess ExVivoMicroTest. The architecture of the functional services of PiggyMetrics is shown in Figure 1. All the microservices in PiggyMetrics are independently deployable. The Account microservice implements functionalities to manage accounts, savings, and expenses. The Statistics microservice implements functionalities to compute major statistics and track cash flow dynamics of the accounts. The Notification microservice stores contacts information and notification services. Each microservice is a stateless microservice with its own database running in a separated container. The requests to these three services are collected and handled by a gateway component. The communication between the services is based on REST application

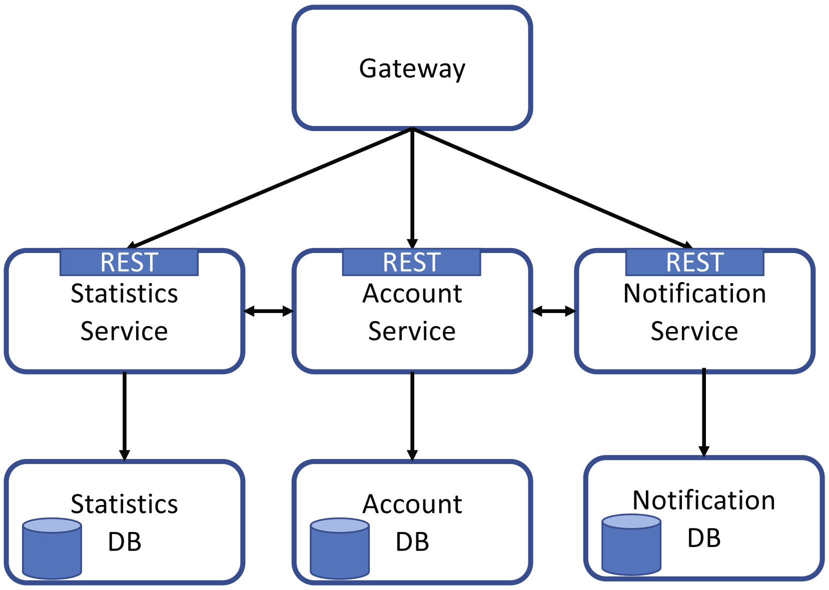


FIGURE 1 Microservice architecture of the PiggyMetrics[11] functional services

program interfaces (APIs), while the communication between services and databases (DBs) uses the MongoWire protocol. In addition to these services, PiggyMetrics includes a few auxiliary technical services, such as services for log analysis, dashboard, and authentication, not shown in Figure 1.

3 | ExVivoMicroTest: OVERVIEW AND MODELS

Figure 2 shows the high-level structure of the ExVivoMicroTest process. It consists of multiple activities that span from the field to the development environment, to finally generate an *in-house regression test suite* that captures how services are *used in the field*.

In the field, ExVivoMicroTest *monitors* the service under test, to determine when to enable and disable trace recording, *traces* interactions, to save incoming and outgoing requests and responses so that they can be replayed in-house, and *forwards* the recorded traces for in-house analysis. Every time tracing is enabled and then disabled, a trace is recorded. On the long run, ExVivoMicroTest opportunistically saves several traces based on the decisions taken by the monitor.

In house, ExVivoMicroTest first *splits* the recorded traces depending on the services involved in the recorded interactions. An *I/O trace* includes the requests *received by* the target service in a recorded trace and the corresponding responses. *Interaction traces* include the requests *produced by* the target service to the other services available in the systems and the corresponding responses produced by these services, in the scope of a recorded trace. Each pair of interacting services produces a different interaction trace. The I/O trace and the interaction traces obtained from the same recorded trace are a part of the same execution slice.

Next, ExVivoMicroTest *filters* the execution slices that must be used to synthesize test cases based on the behaviors that are still uncovered. Trace selection implies updating the set of covered behaviors in the *behavioral space of the target service*.

ExVivoMicroTest then *synthesizes* the test cases and the mocks necessary for replaying the selected execution slices. The resulting test cases enrich the in-house test suite with new automated test cases that faithfully reproduce the usages observed in the field. Note that since the services tested by ExVivoMicroTest are stateless, as recommended in a microservice architecture, the recorded executions can be replayed starting from any point. The impact of state information stored in external services is implicitly captured by the recorded interactions. For instance, the mock object corresponding to an external database can respond to the received requests according to the observed interactions, which depend on the actual state of the database in the field at the time the interaction was recorded.

ExVivoMicroTest further *updates* the tracing profile, which determines the probability of activating tracing in the field for each event that can be observed.

Finally, when the service is updated, the synthesized test cases can be used to *validate changes* in the service under test, considering how the service is used in the field.

In the rest of this section, we first discuss how we represent the behavioral space and the tracing profile of a service and then we detail each step of the approach in Section 4, dedicated to field activity, and Section 5, dedicated to in-house activity. A list of symbols and notations defined and used throughout the remaining sections is given in Table 1.

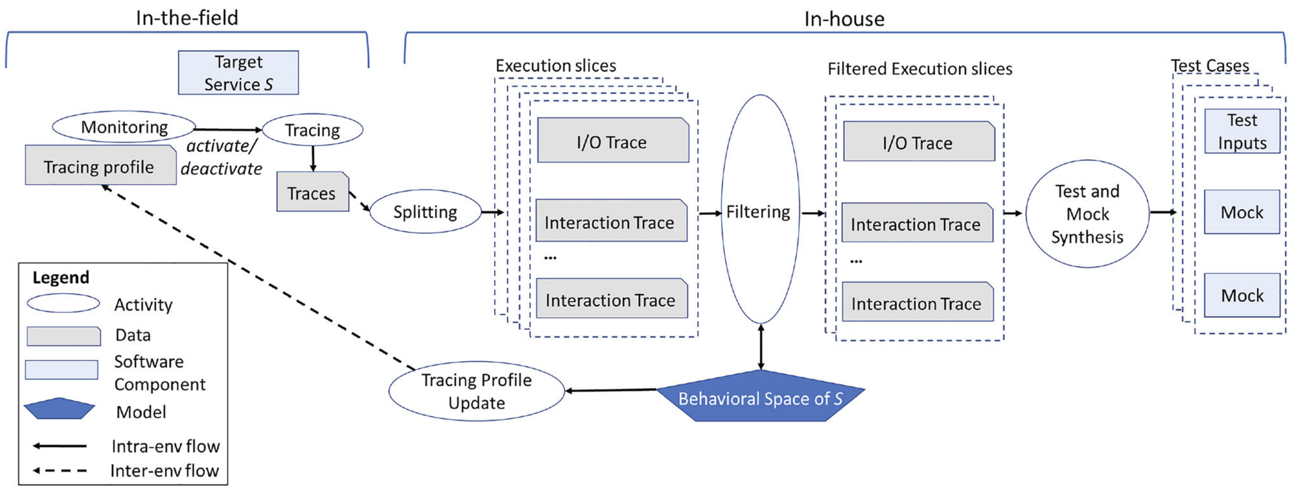


FIGURE 2 The ExVivoMicroTest approach. The target microservice is monitored while it is running in a production environment. Traces collected during the execution are then parsed and further processed to synthesize mockup services simulating interactions with the target microservice. These traces are filtered according to a probabilistic model that is computed based on the behavior of the service under test. This model reflects the diversity of executions and abstracts behaviors to then provide coverage to a wide variety of executions. In parallel, new test cases are generated based on these interactions. These test cases are then used to test an updated version of the microservice

TABLE 1 Symbols and notations

Symbol	Meaning
$S, S_{Ireq}, S_{Oresp}, S_A$	Service, its incoming and outgoing requests and its alphabet
E	Execution slice for a request to S
$Rep(S)_N$	Set of potentially reproducible sequences of length N for S
C_S	Set of sequences already covered by tests for S
$\overline{C_S}$	Set of sequences yet to be covered
$likelihood(e)$	Probability of turning on tracing when event e is observed
p_{min}, p_{max}	Minimum and maximum likelihood, $0 < p_{min} \leq p_{max} \leq 1$
$conf(e)$	Confidence factor—reflects the capability of an event to be the predecessor of uncovered sequences
$tp_S(e)$	Tracing profile function for event e

3.1 | Behavioral space of a service

To identify the scenarios that must be recorded into the the ex vivo regression test suite of a service S, ExVivoMicroTest considers the sequences of requests and the corresponding responses produced by S. The ultimate objective is to cover the possible execution scenarios (i.e., sequences of requests and responses) with test cases extracted from field executions, so that these scenarios can be replayed when needed (e.g., when the service is updated).

More rigorously, each service S is associated with the following:

- a set of incoming requests S_{Ireq}, which represents the requests to S originated by other services,
- a set of outgoing responses S_{Oresp}, which represents the responses that S can produce as a consequence of an incoming request. Given a response $e \in S_{Oresp}$, we indicate with $I(e)$ the set of incoming requests that can result in e.

The alphabet of a service S is indicated with S_A and is defined as follows:

Definition 1 Alphabet of a service. For a set of requests S_{Ireq} and a set of responses S_{Oresp} associated with a service S, the alphabet of S is given by $S_A = S_{Ireq} \cup S_{Oresp}$.

In this paper, we focus on services that can be exercised through an API (e.g., RESTful services), which can be easily associated with their set of requests and responses. However, the approach is not limited to them. For instance, the alphabet of a Web application could be defined in terms of the pages that can be requested and the possible parameters of the requests. Similarly, the alphabet of a database service can be abstracted in terms of the type of operation (e.g., add, delete, or update) that is executed.

Since requests and responses have parameters (e.g., a user registration request is associated with the many fields that describe a user), considering every possible combination of values as a different element of the alphabet to be covered in the tests would generate large, potentially infinite, alphabets for services. Covering sequences from these alphabets would indeed generate huge test suites of little practical utility. It is thus necessary to abstract from parameter values.

On the other hand, not taking parameter values into account may result in small alphabets that lack the ability to distinguish requests and responses based on their content. For instance, invoking an API to buy 0 items or 1 item would be represented with the same symbol of the alphabet. If these two cases are treated in the same way, the resulting ex vivo test suites are likely to miss many relevant scenarios. To find a good trade-off between test accuracy and size of the test suites, we rely on results obtained in the domain of protocol inference,[15,16] where functions that map concrete data values to an abstract and small categorical domain have been successfully used. Among the many abstraction strategies that are available,[15–18] we decided to rely on the abstraction functions defined by Dallmeir et al.,[15] since they match well our domain, and we could complete them by introducing specific extensions to deal with strings, lists/maps, and HTTP status codes. The final result is shown in Table 2. *Variable* indicates the type of the element that must be abstracted. *Abstraction* indicates the values in the categorical domain that we use to abstract the possible concrete values. *Mapping* indicates the symbol used to present every possible category.

In the case parameters are already associated with validation routines (e.g., a validation routine implemented in the API to check for the correctness of the parameters), we exploit this domain knowledge to define the categories accordingly, instead of relying on the default abstractions that are listed in Table 2. Some examples of cases extracted from PiggyMetrics are reported in Table 3.

In the case of RESTful API, ExVivoMicroTest concatenates the elements of the categorical domains (obtained by abstracting the params and the payload of the requests and the responses) with the name of the operation (a prefix that represents if it is a request or a response followed by the name of the endpoint) to finally obtain an abstract symbolic representation of a request or a response (i.e., an element in S_A), as shown in Table 4.

TABLE 2 Abstractions and mappings

Variable	Abstraction	Mapping
Numerical value	NULL | lt0 | gt0 | eq0	0 | 1 | 2 | 3
String value	NULL | empty | notEmpty	0 | 1 | 2
Object ref	NULL | notNULL (instanceof Class C)	0 | 1
Enum	NULL | all e in E	0…e
Boolean	NULL | true | false	0 | 1 | 2
List/Map	NULL | empty | notEmpty	0 | 1 | 2
Status code	1xx | 2xx | 3xx | 4xx | 5xx	0 | 1 | 2 | 3 | 4

TABLE 3 Domain-specific abstractions and mappings examples

Attribute	Abstraction	Mapping
Account.note	NULL | ≤ 20000 | >20000	0 | 1 | 2
User.username	NULL | ≥ 3 and ≤ 20 | other	0 | 1 | 2
User.password	NULL | ≥ 6 and ≤ 40 | other	0 | 1 | 2

TABLE 4 HTTP request–response abstractions

Request / response	Abstraction
Incoming request	{ request prefix }{ endpoint }{ query params }{ payload }
Outgoing response	{ response prefix }{ endpoint }{ query params }{ status code }{ payload }

Example The API method *createAccount* of the PiggyMetrics Account Service has an input payload that is an instance of the *User* class and returns an instance of the *Account* class (query params are empty in both cases). The User class has the attributes *User.username* and *User.password* that are abstracted with three possible symbols each, as shown in Table 3. This generates nine possible symbols of the requests of the form *ReqCreateAccout00*, where *Req* represents requests, *CreateAccount* represents the name of the operation, and *00* represents the fields username and password both assigned with *NULL* values. For the sake of simplicity, let us assume that the response Account class includes only two fields, a saving field and a note field (the real Account class in PiggyMetrics has also a string field, two list fields and an object reference field). The saving field is abstracted with the default rule for object references (see Table 2), while the note field is abstracted with its domain rule (see Table 3), resulting in six possible symbols for the responses. In total, the example *createAccount* API method is abstracted with 15 symbols of the behavioral space.

To extensively validate the behavior of a service S, ExVivoMicroTest exploits combinatorial testing principles[19]; that is, it aims at synthesizing a test suite that covers every *potentially reproducible* sequence of N events in S_A. A sequence $e_1 \ldots e_n$ of operations $e_i \in S_A$ is potentially reproducible in-house if the following conditions hold:

- C_1 - The sequence starts with a request. In fact, if a recorded sequence starts with a response, it is impossible to reproduce, since the incoming request that caused the response would not be available in the scope of the saved trace.
- C_2 - The sequence cannot contain more responses than requests. In fact, to obtain a reproducible execution, the responses must be caused by the requests in the scope of the recorded sequence. Since a request cannot generate more than one response, the number of responses cannot be higher than the number of requests.
- C_3 - The responses must be caused by the requests that occur in the recorded sequence; otherwise, the response could not be reproduced since the request that originated the response would be outside the scope of the saved trace.

More rigorously,

Definition 2 Potentially reproducible sequences of events. Given $EV = e_1 \ldots e_n$, such that $e_i \in S_A, EV$ is potentially reproducible if:

- $C_1 : e_1 \in S_{Ireq}$
- $C_2 : |e_i \in S_{Ireq}| \geq |e_j \in S_{Oresp}|$
- $C_3 : \forall e_i \in S_{Oresp}, \exists e_j, s.t., 1 \leq j < i, e_j \in I(e_i)$

We indicate with $Rep(S)_N$ the set of all potentially reproducible sequences of length N of the service S.

Note that there might still be some sequences that are not reproducible in $Rep(S)_N$. For instance, although requests and responses are apparently paired, a response might by chance refer to a request outside the scope of the saved execution, or a particular combination of requests could be impossible to generate in a specific system. Still $Rep(S)_N$ represents a quite accurate representation of the set of feasible interactions that ExVivoMicroTest may aim to cover. Note that covering $Rep(S)_N$ with in-house tests is not an option for two reasons: (1) the objective of ExVivoMicroTest is to obtain a regression test suite that captures how software is used in the field, and thus, in-house tests cannot replace ex vivo tests, and (2) the size of $Rep(S)_N$ cannot be practically addressed in-house, since, already for sequences of length 3 ($N=3$), it might be necessary to implement test suites that cover millions of combinations of requests and responses, as reported in our evaluation.

ExVivoMicroTest keeps track of the covered sequences. We indicate the sequences already covered for a service S with C_S. The set C_S is initially empty, and it is incrementally enriched with the covered sequences. Normally, C_S does not converge to $Rep(S)_N$, since the full set of combinations could be so large to be impractical to be covered and some of the sequences might simply be infeasible.

For convenience of notation, we use $\overline{C_S}$ to indicate the potentially reproducible sequences of evens in S_A of length N not yet covered, that is, $\overline{C_S} = Rep(S)_N \setminus C_S$.

3.2 | Tracing profile

Tracing cannot be continuously active; otherwise, a huge and unmanageable amount of data would be collected from the field. Since services normally have a mostly deterministic behavior, also depending on the received inputs, ExVivoMicroTest exploits the history of the execution to estimate the probability of collecting unseen interactions immediately after an event is observed. This information is stored in the tracing profile. In particular, the tracing profile associates every possible event e that can be observed while interacting with a service S with a probability $p_e^{tracing}$ of activating the tracing immediately after observing the event e. More formally:

Definition 3 Tracing profile. Given a service S and an event $e \in S_A$, the tracing profile is a function $tp_S : S_A \rightarrow [0...1]$, such that $tp_S(e) = p_e^{tracing}$.

These probabilities are dynamically adjusted based on the likelihood that an execution that has not been covered yet (i.e., a sequence of events in $\overline{C_S}$) is observed just after an observed event e. A detailed explanation of this process is given in Section 5.5.

4 | ExVivoMicroTest: IN-THE-FIELD STEPS

4.1 | Monitoring

Continuously recording every interaction with a given service may quickly produce extremely large trace files, which are expensive to save, store, ship, and maintain. This cost quickly increases when multiple services are subject to the ex vivo testing process.

To address this issue, ExVivoMicroTest distinguishes between monitoring, which observes executions to predict if future interactions are worth to be recorded, and tracing, which actually records interactions based on the decisions taken by the monitor, to obtain reproducible test cases. In practice, the monitor attached to a service captures both requests and responses and determines tracing activation and deactivation.

The monitor of a service S is instructed with (a) the sets of events that must be monitored, namely, S_A, (b) mapping rules that associate concrete requests and responses to the corresponding elements of the alphabet (i.e., the mapping rules derives from abstraction functions, such as the ones in Tables 2 and 3), and (c) a tracing profile tp_S.

The behavior of the monitor is straightforward: Every time a request to the monitored service or a response from the monitored service is intercepted, the monitor uses the mapping rules to identify the symbol that represents the request or the response, namely, e, and activates tracing with the probability indicated in the tracing profiles, that is, $tp_S(e)$. When tracing is activated, the event e that determined its activation is annotated in the recorded trace. If N is the length of the sequences to be covered, tracing could stay active for any number of events, as long as at least N events are collected. However, collecting just N events might be insufficient to fully record a sequence of interest, while collecting long sequences might record many irrelevant events. We have not studied the impact of this parameter systematically, but based on the empirically evidence that we collected, maintaining tracing active for 2N events is a good compromise between the amount of recorded events and the capability to capture the uncovered sequences of events.

More rigorously, if $Rep(S)_N$ is the set of sequences that must be covered with field tests for a service S, every time tracing is activated, it records everything until $2N$ events in S_A are observed. As reported in the evaluation, monitoring and tracing operations are extremely simple and fast, and their execution does not impact on the monitored service.

4.2 | Tracing

Tracing collects traces that include every information necessary to reproduce the observed executions, namely, all the requests and responses to and from the target service. To conveniently capture requests and responses, tracing works at the application layer. For instance, it captures HTTP REST APIs and MongoDB Wire protocol to record interactions with RESTful services and MongoDB databases.

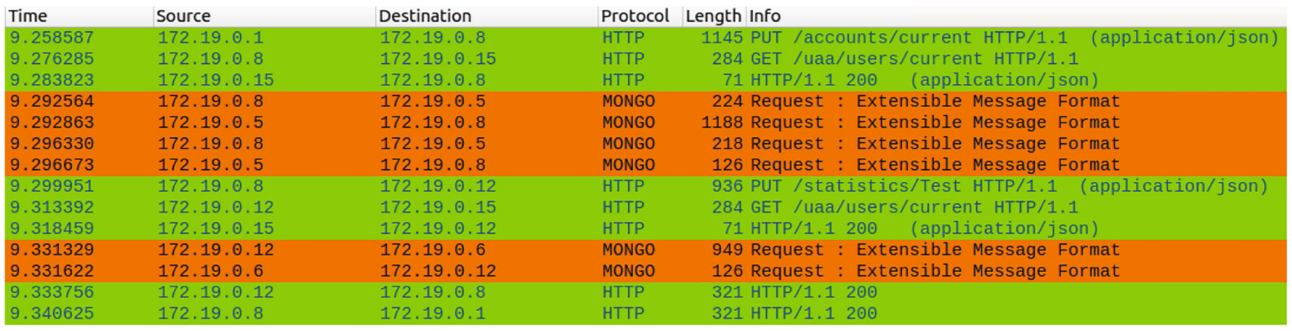
Example Figure 3 shows a portion of a trace for PiggyMetrics. The sample interaction starts with a request produced by the gateway to the Account Service. As a consequence, the Account Service first authenticates with the Authenticate Service, then it executes two queries on its database, and it interacts with the Statistics service. The Statistics service interacts with the Authenticate service and with its database. Finally, the Statistics service interacts with the Account service, which returns a result to the Gateway. Every time tracing is activated, a trace like the one in Figure 3 is recorded.

5 | ExVivoMicroTest: IN-HOUSE STEPS

5.1 | Splitting

The recorded traces may include data about the communication between multiple services. For instance, the trace recorded for PiggyMetrics shown in Figure 3 includes interactions between six services. For the purpose of building ex vivo nonregression test cases for a specific service, it is important to distinguish the requests received by the monitored service and the corresponding responses produced by the service, from the rest of the communication. In fact, the sequence of the received requests can be intuitively turned into a test case and the observed responses can be used to define an oracle that checks for the correctness of the results produced by the tested service. In more detail, the splitting step produces an *I/O trace* from each recorded trace by including all and only the messages where the target service is the receiver of the message (i.e., the target service is the destination) and corresponding responses.

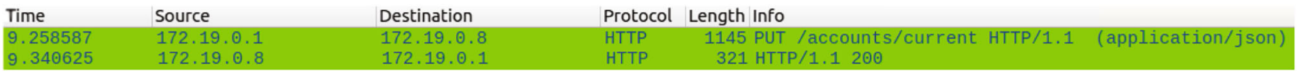
Example For example, the I/O trace for the Account Service resulting from the trace shown in Figure 3 is shown in Figure 4. The trace includes a single request, produced by the Gateway to the Account service and the corresponding response generated by the Account service. In general, the response to a recorded request depends on the many interactions between the target service and the rest of the services of the system that might be produced by the target service while serving the request. To obtain traces that can be fully reproduced, ExVivoMicroTest analyzes the recorded trace and identifies the pairs of interacting services where the target service is the producer of the messages (i.e., the target service is the source). Each pair of interacting services generates a different *interaction trace* that contains the requests produced by the target service and the corresponding responses. Intuitively, each interaction trace represents a dependency of the target service that is turned into a mocked service that is able to replay the observed interactions.



```
Time       Source       Destination   Protocol Length Info
9.258587   172.19.0.1   172.19.0.8    HTTP     1145 PUT /accounts/current HTTP/1.1  (application/json)
9.276285   172.19.0.8   172.19.0.15   HTTP      284 GET /uaa/users/current HTTP/1.1
9.283823   172.19.0.15  172.19.0.8    HTTP       71 HTTP/1.1 200   (application/json)
9.292564   172.19.0.8   172.19.0.5    MONGO     224 Request : Extensible Message Format
9.292863   172.19.0.5   172.19.0.8    MONGO    1188 Request : Extensible Message Format
9.296330   172.19.0.8   172.19.0.5    MONGO     218 Request : Extensible Message Format
9.296673   172.19.0.5   172.19.0.8    MONGO     126 Request : Extensible Message Format
9.299951   172.19.0.8   172.19.0.12   HTTP      936 PUT /statistics/Test HTTP/1.1  (application/json)
9.313392   172.19.0.12  172.19.0.15   HTTP      284 GET /uaa/users/current HTTP/1.1
9.318459   172.19.0.15  172.19.0.12   HTTP       71 HTTP/1.1 200   (application/json)
9.331329   172.19.0.12  172.19.0.6    MONGO     949 Request : Extensible Message Format
9.331622   172.19.0.6   172.19.0.12   MONGO     126 Request : Extensible Message Format
9.333756   172.19.0.12  172.19.0.8    HTTP      321 HTTP/1.1 200
9.340625   172.19.0.8   172.19.0.1    HTTP      321 HTTP/1.1 200
```

172.19.0.1 = Gateway, 172.19.0.8 = Account Service, 172.19.0.15 = Authentication Service, 172.19.0.12 = Statistics Service, 172.19.0.5 = Accounts Service Database, 172.19.0.6 = Statistics Service Database.

FIGURE 3 Example trace for PiggyMetrics (the content of messages is reported partially)



```
Time       Source       Destination   Protocol Length Info
9.258587   172.19.0.1   172.19.0.8    HTTP     1145 PUT /accounts/current HTTP/1.1  (application/json)
9.340625   172.19.0.8   172.19.0.1    HTTP      321 HTTP/1.1 200
```

172.19.0.1 = Gateway, 172.19.0.8 = Account Service.

FIGURE 4 IO trace of the Account service derived from the trace in Figure 3 (the content of messages is reported partially)

If interactions between services are not controlled during testing, a tested service may try to interact with services that are not available, that cannot be used without producing side effects, or that are in states different from the states of the services used in the field. ExVivoMicroTest exploits the mock technology as described later in this section to properly handle these interactions.

Example As shown in the trace reported in Figure 3, the Account service communicates with the Authentication Service, the Statistics service, and its database service. To achieve a reproducible test case, it is thus necessary to also reproduce these interactions; otherwise, the in-house execution will diverge from the field execution. Figure 5 shows the interaction trace that captures the interactions between the Account and the Authentication services, as observed in the trace shown in Figure 3. The mocked Authentication service generated from this interaction trace must be able to accept the two requests produced by the Account service and respond with the same responses present in the trace.

The I/O trace and the interactions traces obtained from a same trace recorded from the field constitute an *execution slice*.

Definition 4 Traces and Execution Slices. In a nutshell, given a trace t, a target service S, and a potentially empty set of services S_i that received requests from S in t,

- The I/O trace I_{io} derived from t is an ordered sequence of events that include every request received by S and the corresponding responses.
- For each S_i, the interaction trace I_{S-S_i} derived from t is an ordered sequence of events that include every request produced by S to S_i and the corresponding responses.
- The execution slice E derived from t consists of the I/O trace I_{io}, the interaction traces I_{S-S_i}, and a function $pos: I_{S-S_i} \rightarrow [0...|I_{io}|]$ that are associated each interaction trace with its starting point of execution with respect to events in I_{io}, that is, $E=(I_{io}, \{I_{S-S_i} | \forall S_i\}, pos)$.

5.2 | Test filtering

An execution slice E captures every interaction that happened during a recorded trace. Test filtering determines the ones that must be turned into test cases. This is achieved by analyzing the I/O trace in E. The sequences of requests and responses in the I/O trace of a service S are elements in $Rep(S)_N$. In particular, an I/O trace with $2N$ events contains $N+1$ sequences of N events in S_A. ExVivoMicroTest checks if any of these subsequences appear in $\overline{C_S}$, that is, if there are sequences that have not been covered yet. These subsequences and the corresponding interaction traces become candidates for test case generation. The remaining sequences and subsequences are discarded.

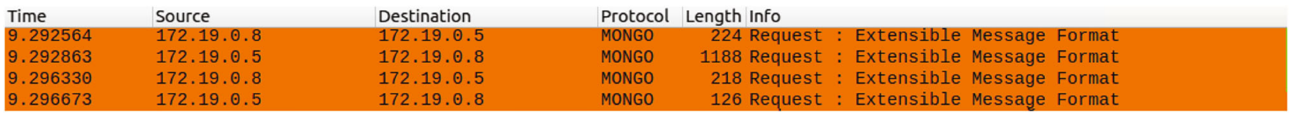
More rigorously, given an execution slice $E=(I_{io}, I_{int}, pos)$, if there is a sequence $EV=e_1...e_N$ such that EV is a subsequence of I_{io} and $EV \in \overline{C_S}$, a filtered execution slice $E'=(EV, I'_{int}, pos')$ with $I'_{int} \subseteq I_{int}$ representing the interactions originated, while EV is executed according to pos, and pos' representing the straightforward restriction of pos to I'_{int} is generated as execution slice eligible for test synthesis (filtered execution slice in Figure 2). If none of these sequences can be derived from an execution slice, the execution slice is dropped. Thus, once the filtering step is complete, we may remain with fewer execution slices and with smaller execution slices since only the uncovered subsequences are used for test synthesis.

This step finally moves the elements of $Rep(S)_N$ that are covered in the selected execution profiles from the set of uncovered sequences $\overline{C_S}$ to the set of covered sequences C_S.

5.3 | Test synthesis

Given an execution slice, test synthesis turns the sequence of requests and responses contained in the I/O trace, such as the one shown in Figure 4, into an actually executable test case. The resulting test must thus map the requests in the I/O trace into actual service requests and the responses into checks performed in the test. A response different from the recorded one would indicate the presence of a regression problem, that is, an operation that is supposed to not be affected by a change but is actually behaving differently after the update.

ExVivoMicroTest synthesizes Python test cases from I/O traces. The observed requests are simply reproduced.



Time	Source	Destination	Protocol	Length	Info
9.292564	172.19.0.8	172.19.0.5	MONGO	224	Request : Extensible Message Format
9.292863	172.19.0.5	172.19.0.8	MONGO	1188	Request : Extensible Message Format
9.296330	172.19.0.8	172.19.0.5	MONGO	218	Request : Extensible Message Format
9.296673	172.19.0.5	172.19.0.8	MONGO	126	Request : Extensible Message Format

172.19.0.5 = Authentication Service, 172.19.0.8 = Account Service.

FIGURE 5 Interaction trace that captures the communication between the Account Service and the Account Service Database, derived from the trace in Figure 3 (the content of messages is reported partially)

After each request, the synthesized test case includes two checks, one on the status code of the response and the other one on the content of the response, which must match the same response in the IO trace with the exception of the fields reporting time, date and ids, which might change arbitrarily.

Example Figure 6 shows a regression test case synthesized by ExVivoMicroTest. The observed request is mapped into the `headers` variable which is then used to invoke the target operation of the target service with a `requests.get` operation. In our running example, the test executes the demo operation of the account service. The `response.status_code` is then checked with an assertion. Part of the content of the response is filtered out by removing fields that should not be considered by the oracle (operation `re.sub`). The result is then loaded in the `data_cont` variable. Similarly, the `packet_data` variable is initialized with the actual response observed in the field, the variables that should not be considered by the oracle are filtered out, and the resulting content is stored in the `data_pkt` variable. These two variables are finally compared with an assert statement. Note that manipulating the values of specific fields that cannot be reused as-is is common to approaches for deriving tests from field data.[20] This pattern is iterated for every request that must be produced and for every response that must be checked.

The result of this step is a regression ex vivo test case that checks if operations that are supposed to be not affected by a change still behave the same after an update. Of course, test execution is possible only if combined with mock synthesis, which is described in the following section.

5.4 | Mock synthesis

When a test is executed, the service under test may interact with other services, as captured by the interaction traces associated with an I/O trace within an execution slice. To exactly reproduce the observed scenarios, also the interactions between the service under test and the external services must be reproduced. To this end, ExVivoMicroTest generates a mock service for each interaction trace in the execution slice, that is, for each service involved in the execution that must be turned into a test case. Each mock service is instructed to receive requests and send responses coherently with the interaction trace used to generate it.

Example Starting from the PiggyMetrics trace shown in Figure 3, ExVivoMicroTest generates and instantiates three mock services, corresponding to the three services that directly interact with the Account: the Authentication, the Statistics, and the Account database services. If we consider the interaction trace shown in Figure 5, the corresponding mock is expected to receive two requests and produce two responses.

We should note that by synthesizing the mock objects, we in fact implicitly bring state information into the test scenario. For instance, in the running example, one of the mock objects replaces the database and responds to the request of the service under tests in the same way that the service responded in the field. The usage of mock services allows to store interactions, which include just the state information necessary to replay the execution, rather than saving and restoring the full database, which might be expensive.

A mock implementation is similar to the test implementation shown in Figure 6, with the role of the operations logically inverted. That is, a mock has a `REST` interface through which it receives requests from the service under test. Each request is checked and compared against the request recorded from the field. In case the received, request does not match the recorded request, a misbehavior is identified in the service under test, and a failure is reported to the developers. If the received request passes the checks, the mock produces a request based on the content of the interaction trace, using code similar to the one produced for the test case.

```
headers={'Content—type': 'application/json', 'Accept': 'application/json'}

print('sending get request to http://localhost:6000/accounts/demo')
response = requests.get('http://localhost:6000/accounts/demo', headers=headers)
print('response: {0}'.format(response.content))

assert response.status_code == 200

content = re.sub(r'"id".*?(?=,)', '"id":null',response.content.decode('utf—8'))
content = re.sub(r'"timestamp".*?(?=,)', '"timestamp":null',content)
content = re.sub(r'"lastSeen".*?(?=,)', '"lastSeen":null',content)
content = re.sub(r'"date".*?(?=,)', '"date":null',content)
data_cont = loads(content)
packet_data = '{"name":"demo","lastSeen":"2019—01—21T08:30:10.141+0000","incomes":[{"title":"Salary","amount":42000.0,
        "currency":"USD","period":"YEAR","icon":"wallet"},{"title":"Scholarship","amount":500.0,"currency":"USD",
        "period":"MONTH","icon":"edu"}],"expenses":[{"title":"Rent","amount":1300.0,"currency":"USD",
        "period":"MONTH","icon":"home"},{"title":"Utilities","amount":120.0,"currency":"USD",
        "period":"MONTH","icon":"utilities"},{"title":"Meal","amount":20.0,"currency":"USD",
        "period":"DAY","icon":"meal"},{"title":"Gas","amount":240.0,"currency":"USD",
        "period":"MONTH","icon":"gas"},{"title":"Vacation","amount":3500.0,"currency":"EUR",
        "period":"YEAR","icon":"island"},{"title":"Phone","amount":30.0,"currency":"EUR","period":"MONTH",
        "icon":"phone"},{"title":"Gym","amount":700.0,"currency":"USD","period":"YEAR","icon":"sport"}],
        "saving":{"amount":5900.0,"currency":"USD","interest":3.32,"deposit":true,"capitalization":false},
        "note":"demo note"}'
packet_data = re.sub(r'"timestamp".*?(?=,)', '"timestamp":null', packet_data)
packet_data = re.sub(r'"lastSeen".*?(?=,)', '"lastSeen":null', packet_data)
packet_data = re.sub(r'"date".*?(?=,)', '"date":null', packet_data)
data_pkt = loads(packet_data)
assert data_cont == data_pkt
```

FIGURE 6 An example of automatically generated regression test case

5.5 | Tracing profile update

The tracing profile tp_S of a monitored service S is regularly updated based on (a) the (updated) set of covered sequences C_S and (b) the sequences of events $EV = e_1 \ldots e_m$ that have determined the activation of the tracing from the last update of the tracing profile. The tracing profile is updated every time significant new information is collected, that is, after a number of events have been processed by the monitor. For instance, the tracing profile tp_S of a service S could be updated every 1000 events processed by the monitor of that service. In the experimental evaluation, we investigate the impact of this parameter on the results to setup the update frequency properly.

The probability of activating tracing immediately after an event e is influenced by two main factors: *likelihood*, that is, the probability for the execution to continue with a sequence of requests and responses that has not been covered yet (i.e., an element of $\overline{C_S}$), and *confidence*, that is, the actual evidence collected so far about the capability of an event to be the predecessor of uncovered sequences. These two factors alter the probability $p_e^{tracing}$ of an event, such that $p_e^{tracing} \in (0, p_{max}]$, where $p_{max} \leq 1$. Intuitively, likelihood exploits historical data to consider the probability an event e can be the predecessor of a sequence that has not been observed yet, considering how the execution may continue from e, while confidence exploits historical data to give more weight to the events that have been already able to reach novel areas of the behavioral space (e.g., because they exercise important decision points in the application).

In particular, since it is impossible to precisely predict future executions, a minimum probability greater than 0, although small, guarantees that after every event, there is always a chance to record the field execution. The value p_{max} guarantees that the probability of tracing events is never too high to interfere with the operation of the monitored service. The choice of p_{max} depends on the characteristics of the monitored services, their workload, and the available resources.

When ExVivoMicroTest is first deployed, the probability to turn tracing on is set to p_{max} for all the events. In fact, since the set of covered sequences C_S is initially empty, executions can be collected eagerly to quickly populate the ex vivo test suite.

Every time a set of traces is processed, the probability associated with each event is updated based on the likelihood and the confidence values.

The *likelihood* computation requires ExVivoMicroTest to first compute for every event e the probability $p(e)$ that an event $e \in S_{Ireq}$ might start an unobserved sequence. This is estimated as the ratio between the number of uncovered sequences that starts with e with respect to the overall set of sequence starting with e, that is, $p(e) = \frac{|\{ee_2 \ldots e_N \in \overline{C_S}\}|}{|ee_2 \ldots e_N \in Rep(S)_N|}$.

Since the tracing must be turned on before the first event of the sequence is observed, ExVivoMicroTest considers the already covered sequences and the events that have been observed to occur before the first event of an uncovered sequence. In particular, given an event e', ExVivoMicroTest retrieves the events that have been observed to occur just after e' and uses their probability to start an unobserved sequence to compute the updated probability of turning tracing on. We therefore define *likelihood* as follows:

Definition 5 Likelihood. Given event e' and the set $Next(e') = \{e_{k+1} | e_1 \ldots e_k, e_{k+1}, \ldots e_N \in C_S, k = 1 \ldots N-1, e_k = e'\}$, the probability of turning on the tracing when e' is observed is $likelihood(e') = (p_{min} + max_{e \in Next(e')} p(e)(p_{max} - p_{min}))$.

In addition to using the value p_{max} for the upper bound, the formula of the likelihood uses a $p_{min} > 0$ to guarantee that a minimum positive likelihood is always assigned with an event e'. In fact, the likelihood of an event ranges between p_{min} and p_{max}.

Regarding *confidence*, ExVivoMicroTest considers the sequences of events $EV = e_1 \ldots e_m$ that determined the activation of the tracing from the last update of the tracing profile. For each event e in S_A, ExVivoMicroTest starts with a confidence factor $conf(e)$ equals to 1. EV is then processed sequentially. Every time an event $e_i = e$ is encountered, if the trace recorded after the activation of e_i increased the number of covered sequences, its confidence $conf(e)$ is again assigned with 1. Otherwise, it is multiplied by 0.9. In this way, events that regularly anticipate novel scenarios are assigned with a confidence close to 1; otherwise, events that regularly miss to anticipate novel scenarios are gradually assigned with lower confidence values.

The confidence factor is finally used to scale the value of the likelihood and obtain the final probability of turning on tracing: $p_e^{tracing} = conf(e) \cdot likelihood(e)$. Since $conf(e)$ is in the range $(0,1]$ and $likelihood(e)$ is in the range $[p_{min}, p_{max}]$, with $0 < p_{min} \leq p_{max} \leq 1$, the final probability $p_e^{tracing}$ can vary in the range $(0, p_{max}]$. These values establish the new tracing profile, that is, $tp_S(e) = p_e^{tracing}$, and are dispatched to the monitor that uses them in the field.

5.6 | Test execution

The execution of the ex vivo test cases requires setting up the right runtime environment. In our work, we refer to container-based technologies to obtain the runtime environment. However, this result can be obtained also with other technologies, for instance, using virtual machines.

ExVivoMicroTest produces the runtime environment that is schematically illustrated in Figure 7. The actual service under test is deployed in a container. Additionally, each mocked service is deployed in its own container. These mocked services typically include the gateway, the external

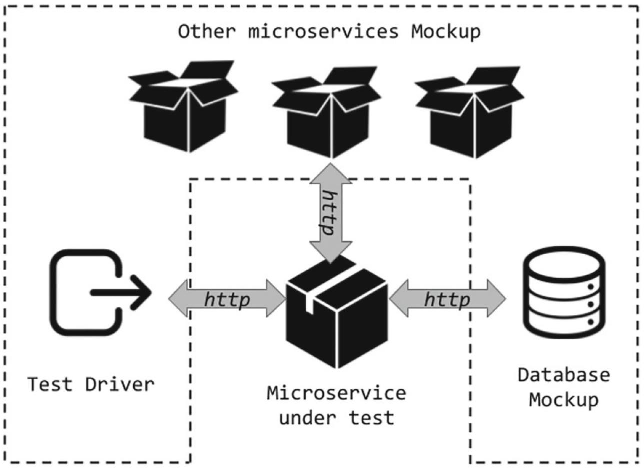


FIGURE 7 ExVivoMicroTest regression testing architecture

database, and others. The driver finally executes the test case. The responses provided by the service under test and the requests produced by the service under test are checked by the driver and the mocks, respectively. If none of these checks fails, the test is reported as passed; otherwise, a failure is returned.

6 | IMPLEMENTATION DETAILS

This section provides details about the implementation of ExVivoMicroTest.

The monitoring component is implemented with HTTP filters[21] to nonintrusively and efficiently intercept requests and responses. Depending on the protocol and data to be monitored, we also envision the possibility to exploit other technologies in the Cloud context. For instance, Elastic Stack[22] and Prometheus[23] are two popular monitoring systems that can be used to probe cloud applications, while Monasca,[24] CloudHealth,[25] and Varys[26] provide more sophisticated monitoring features, including the capability to dynamically turn on and off probes.

Tracing is implemented with Tcpdump,[27] which can inexpensively monitor and record data from network interfaces. We configure Tcpdump to capture in/out network packets of the target services. Interestingly, Tcpdump can also monitor and record traces for multiple services at once, if they interact through the same network interfaces. At the moment, our implementation filters out the recorded network packets that do not match with the relevant protocols from the record traces, but more sophisticated implementations could also remove them on-the-fly.

The current implementation of ExVivoMicroTest supports the HTTP REST APIs and the MongoDB Wire application protocols. The choice of these protocols is due to their popularity (e.g., RESTful services account for more than 80% of the services hosted in the ProgrammableWeb directory[28]) and relevance for the subject application selected for the evaluation. Indeed, support to other protocols can be added to the tool to extend the range of microservices that can be addressed.

We implement Splitting and Filtering steps using Bash and Python scripts, respectively. Specifically, PyShark[29] and tshark[30] have been exploited to manage PCAP files recorded by Tcpdump.

Test Synthesis synthesizes Python test cases from I/O traces as reported in Figure 6, while Mock Synthesis creates microservices mocks exploiting mountebank[31] and mongoreplay.[32]

7 | EVALUATION

To assess ExVivoMicroTest, we investigate the following three research questions about the cost-effectiveness of the approach.

7.1 | Research questions

RQ1 - How does the update frequency of the tracing profile affect the capability of ExVivoMicroTest to cost-effectively collect test cases? This research question studies how different update frequencies may impact the capability of ExVivoMicroTest to cover new sequences while opportunistically limiting the frequency of activation of the tracing. The outcome of RQ1 is exploited to determine the value of the update frequency used to study RQ2.

RQ2 - Is ExVivoMicroTest able to cost-effectively collect test cases? This research question studies the ability of ExVivoMicroTest to extract test cases from field executions for multiple scenarios and configurations.

RQ3 - What is the overhead introduced by monitoring and tracing? This research question studies how running monitoring and tracing may impact the target service.

7.2 | Subject applications

To answer the three research questions, we selected two microservices-based open source systems: PiggyMetrics,[11] already described in Section 2, and Train Ticket,[12] which is a train ticket booking system containing 41 microservices.

The PiggyMetrics core business components are shown in Figure 1. Service communication is based on REST APIs[33] and MongoDB Wire Protocol.[34] PiggyMetrics is a relatively popular system, as can be witnessed from the nearly 5000 forks listed in its source code repository.

Train Ticket has been already used as benchmark microservice system by several studies so far,[35–37] and it has been forked nearly 150 times.

As for PiggyMetrics, service communication is mostly based on REST APIs[33] and MongoDB Wire Protocol.[34]

Both the systems include an in-house test suite for each microservice that we used to define realistic execution scenarios. They are therefore excellent targets for studying the effectiveness of ex vivo testing.

We focused the testing activity on three microservices for each system, selecting all the services that implement some business logic from PiggyMetrics and three services with dependencies supported by our prototype implementation in Train Ticket. We report in Table 5 data about the number of API endpoints, size of the alphabet, and size of the reproducible sequences of length between 2 and 4 for the three services, with the exception of the Consign Price and Contacts services of Train Ticket system. Since these services have a large alphabet, covering sequences of length 4 is infeasible, and thus, we considered sequences of maximum length 3.

7.3 | Scenarios definition

To generate ex vivo test cases, we needed to exercise the subjectapplications. To this end, we defined a set of executable use cases that invoke the APIs of the three services covering positive and negative cases. In PiggyMetrics, we identified eight use cases for the Account Service and four use cases for both the Notification and Statistics Services.In Train Ticket, we identified three uses cases for TS_Contacts and four use cases for both TS_Consign_Price and TS_Food_Map. To run these use cases, we implemented a client that interacts with the services.

Since our objective is studying the effectiveness of ExVivoMicroTest applied to long running systems, we generated three types of scenarios starting from the use cases:

- *Regular*: We randomly combine the different use cases multiple times; each use case is entirely executed before adding the next one.
- *With Random Events*: We use the same strategy of the regular scenario, but we add a random but valid sequences of length 2 (one request and the corresponding response) randomly between use cases.
- *Overlapped*: We interleave use cases and random events, as in the case of a service exercised by many clients simultaneously. In this scenario, after an event of a use case is processed, we have the same probability to continue with any of the started use cases, to start a new (overlapped) use case, or to add a valid sequence of length 2 as in the With Random Events scenario.

In all three cases, the full scenario consists of 10 millions events.

TABLE 5 Services details

Service name	# endpoints	$\lvert S_A \rvert$	$\lvert Rep(S)_2 \rvert$	$\lvert Rep(S)_3 \rvert$	$\lvert Rep(S)_4 \rvert$
Account	4	1,320	117,118	42,065,795	15,487,669,432
Notification	2	61	760	22,777	932,626
Statistics	3	36	484	11,732	296,212
TS_Consign_Price	4	8208	8,409,108	25,846,635,899	-
TS_Contacts	7	3047	2,254,209	4,153,870,331	-
TS_Food_Map	5	27	77	879	11,445

7.4 | RQ1: Update frequency

To answer RQ1, we study the impact of the update policy on the Notification service for $N = 4$, which represents a case of average complexity of the service under test. We focus on the *With Random Events* scenario, since it also captures the case of an average magnitude of variability. We use $p_{min} = 0.03$ and $p_{max} = 0.1$ to consider a moderately intrusive scenario. We study the results for an update frequency of 100, 1 K, 10 K, and 100 K events.

To measure the cost-effectiveness of ExVivoMicroTest, we consider the following two metrics.

- *Traced events*: This metric counts the number of events traced by ExVivoMicroTest. Since tracing produces traces that must be stored, shipped, analyzed, and turned into test cases, it represents the cost introduced by a specific configuration.
- *Covered Sequences*: This metric computes the percentage of sequences of a given length that have been covered with respect to the total number of distinct sequences observed. For instance, a solution that records every single interaction results in a value always equals to 100% for this metric.

These metrics are computed after every single event that is processed, for the 10 million events in the analyzed scenario. Figure 8 shows the results. The experiment is repeated 10 times, so the reported values are averages on 10 executions.

Updating the tracing profile every 100 K events introduces a high cost; that is, a high number of events are traced without a benefit in terms of covered sequences. This is due to ExVivoMicroTest updating the tracing probabilities late with respect to the amount of collected evidence, and thus, several irrelevant events are traced before the updated probabilities are available in the field. In fact, the trend of the curve for 100 K is similar to the other curves but the adjustments occur with a delay. The other three possible values perform comparably, both in terms of number of traced events and covered sequences. Considering that values differ by orders of magnitude, this suggests that the choice of the update frequency has a mild impact on the results. Updating the probabilities too frequently (e.g., every 100 events) seems to cause a small instability in the probabilities that may have negatively affected the results, but based on the results, we cannot make strong claims about the factors that caused them. Since an update frequency of 1 K events produced results slightly better than the other update frequencies (slightly higher percentage of covered sequences without an impact on the traced events), we select this value for the investigation of RQ2.

7.5 | RQ2: Cost-effectiveness

To answer RQ2, we systematically study a range of configurations and scenarios, applied to the six services, based on an update frequency of 1 K events. In particular, we consider the following cases:

- The three core services in PiggyMetrics (the Statistics service, the Notification service, and the Account service) and the three selected services from Train Ticket (TS_Consign_Price, TS_Contacts, TS_Food_Map).
- Three scenarios for each service: Regular, With Random Events, and Overlapped.
- Three values of N that result in a behavioral space of increasing size to be covered: 2, 3, and 4 (a value of 4 is not considered forthe TS_Consign_Price and TS_Contacts services due to the excessive size of the resulting behavioral space).
- Three configurations for the tracing probabilities p_{min} and p_{max}: eager ($p_{min} = 0.05$ and $p_{max} = 0.2$), medium ($p_{min} = 0.03$ and $p_{max} = 0.1$), and lazy ($p_{min} = 0.01$ and $p_{max} = 0.05$).

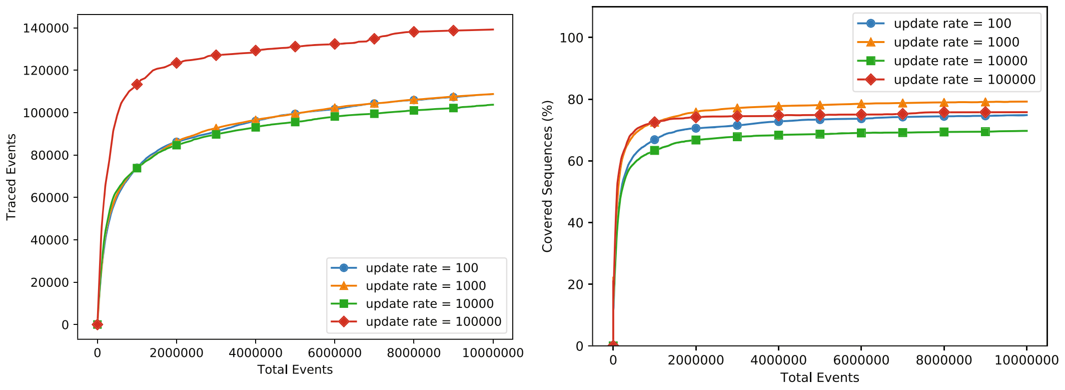


FIGURE 8 Four values of the update frequency compared on the Notification service with $N = 4$

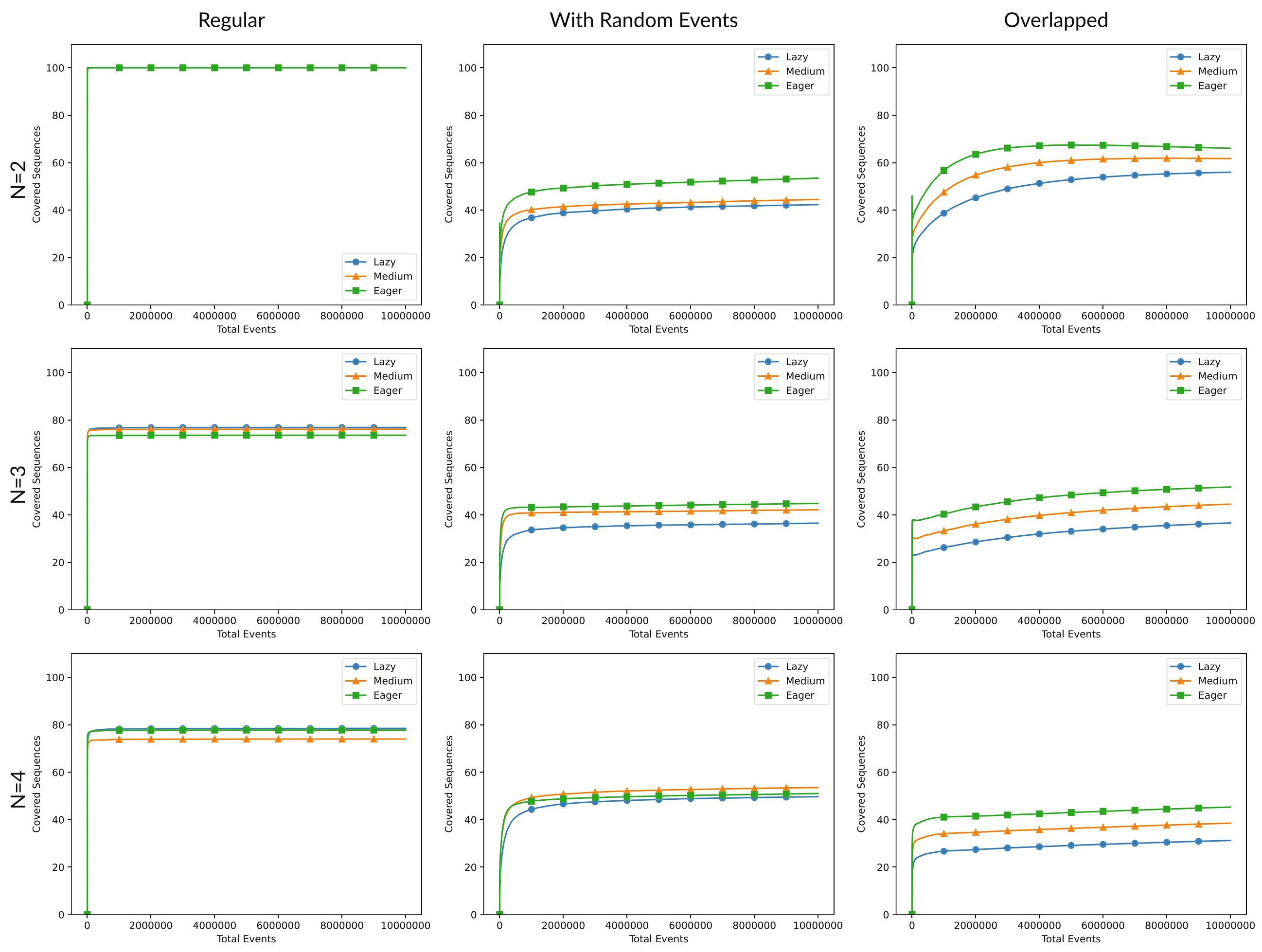


FIGURE 9 Average-covered sequences

This produces a total of144 cases. We executed each case 20 times, for a total of 288 experiments, to mitigate the impact of any source of randomness. Since the number of covered sequences grows with consistent patterns across services, we simply report average results across the 20 executions and the 6 services in Figure 9. Although the shapes of the curves are consistent across services, the number of covered sequences stabilizes to different values for different services and configurations. We thus report the number of covered sequences reached at the end of the experiment for each services with the eager configuration in Table 6.

The traced events also grow consistently across services, with a perfect consistency in the case of the Regular and the Overlapped scenarios and two distinct patterns for the With Random Events scenario. We thus plot average results distinguishing between these two cases in Figure 10.

Due to space constraints, we do not report the results per service in the paper, but they are accessible through our online appendix[*].

The Regular scenario is the easiest to address with ExVivoMicroTest. In fact, in all the cases, the covered sequences quickly saturate to high coverage values (Figure 9 column Regular), regardless of both the service and the size of the behavioral space to be covered (Table 6, column Regular). The frequency of activation of tracing diminishes accordingly (Figure 10 column Regular). Indeed, the interactions in services with a mostly regular behavior are easy to predict, and ExVivoMicroTest can act optimally by quickly collecting many sequences in the early phase of the execution, to then limit the number of times tracing is activated.

In the With Random Events scenario, ExVivoMicroTest behaves similar to the Regular scenario in terms of the covered sequences (Figure 9 column With Random Events); that is, it successfully covers many new events at the beginning, when most of the sequences are uncovered and then it sporadically covers new sequences (i.e., it sporadically generates new ex vivo tests). However, ExVivoMicroTest performs quite differently depending on the complexity of the behavior of the monitored service. In fact, for the Notification, Statistics, and TS_Food_Map services, ExVivoMicroTest reaches high coverage values (Table 6, column With Random Events) and successfully decreases the frequency of activation of

*https://doi.org/10.5281/zenodo.6351506

TABLE 6 Percentage of sequences covered with the eager configuration

	Regular			With random events			Overlapped		
Service	**N = 2**	**N = 3**	**N = 4**	**N = 2**	**N = 3**	**N = 4**	**N = 2**	**N = 3**	**N = 4**
Account	100.00%	89.04%	73.08%	14.90%	12.88%	15.83%	39.52%	37.59%	26.29%
Notification	100.00%	70.00%	85.71%	85.79%	77.52%	68.31%	99.07%	54.91%	38.99%
Statistics	100.00%	73.33%	77.27%	91.41%	65.34%	60.42%	99.87%	61.84%	42.12%
TS_Consign_Price	100.00%	55.56%	-	18.53%	15.13%	-	25.20%	28.53%	-
TS_Contacts	100.00%	78.57%	-	18.22%	20.61%	-	32.80%	31.45%	-
TS_Food_Map	100.00%	75.00%	75.00%	91.95%	77.45%	59.57%	100.00%	96.22%	73.93%

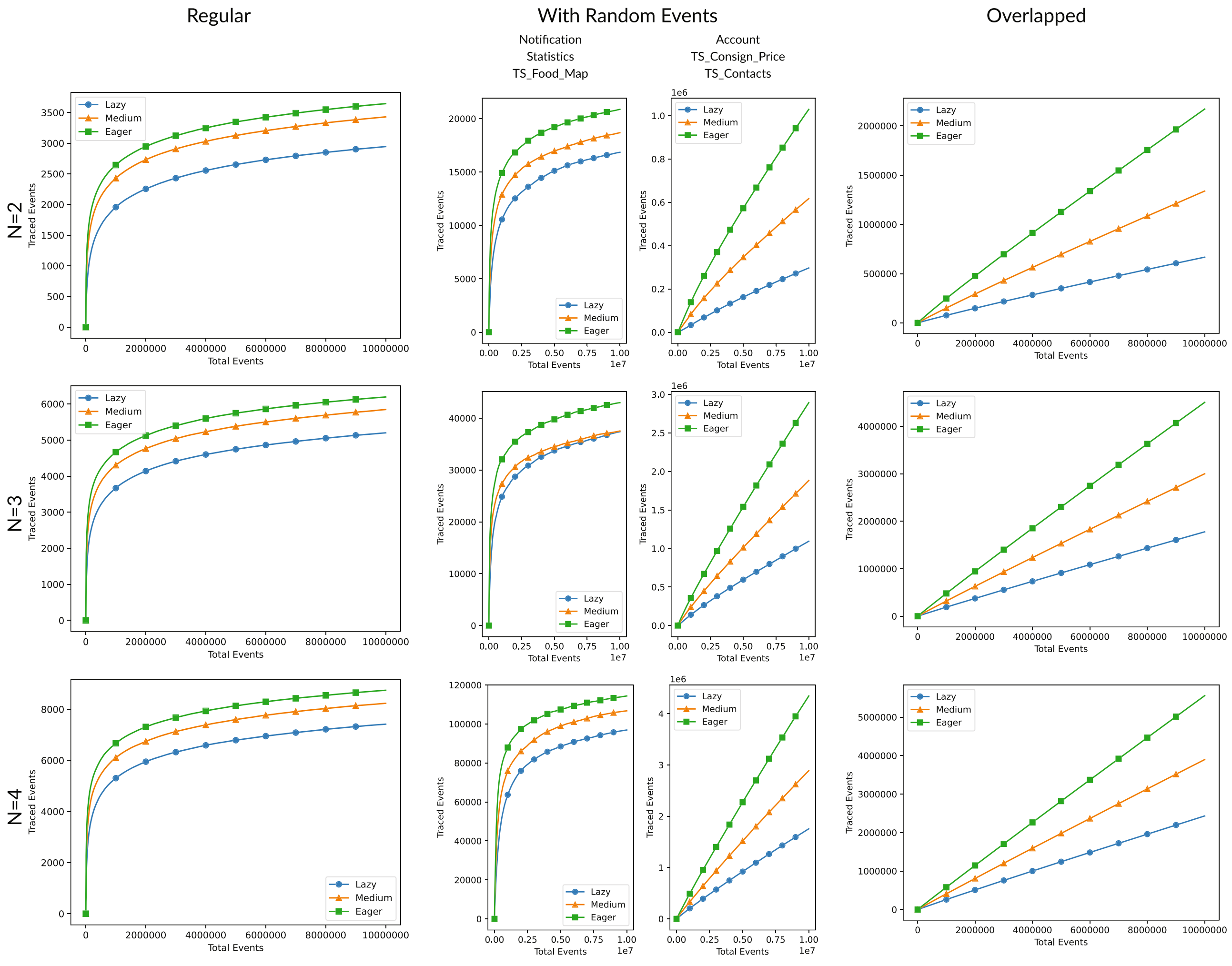


FIGURE 10 Average-traced events

tracing (Figure 10 left part of column With Random Events). While for the rest of the services (Account, TS_Consign_Price, and TS_Contacts), ExVivoMicroTest reaches lower coverage values (Table 6, column With Random Events) and consequently maintains a high frequency of activation of the tracing in the attempt of covering more sequences (Figure 10 right part of column With Random Events). Indeed, capturing rare random events that may appear with little cause–effect relation with the behavior of a service is extremely hard for ExVivoMicroTest that would adapt probabilities of activation based on the frequently observed sequences, likely missing these rare random events.

The Overlapped scenario, due to the number and to the articulated combination of behaviors that it includes, performs similarly to the most challenging cases of behaviors that include random events. In fact, the tracing in the overlapped scenario is consistently high for all services and configurations (Figure 10 column Overlapped). The covered sequences reach different levels depending on the target service, with the smaller services extremely well covered compared with the larger services (Table 6, column Overlapped).

Overall, we can notice that the size of the service is relatively important as long as its behavior is quite regular. On the contrary, if the observed behavior is challenged by random events or many overlapped scenarios, ExVivoMicroTest can well address the smaller services and can only partially address the larger services, although it still reaches nontrivial coverage values, generating ex vivo test cases for around one third of the behavioral space.

The size *N* of the behavioral space is naturally an important variable. Our experiments suggest that values equal to 2 or 3 are reasonable targets, depending on the available resources for monitoring and testing, while a value of 4 can be afforded only for the simplest services.

Finally, the strategies (eager, medium, and lazy) perform quite similarly, with differences that are more significant for the most challenging scenarios and services. As expected, the eager strategy consistently covers more sequences than the medium and lazy strategies. The opposite is true for tracing, with eager activating tracing more often than medium and lazy. Overall, the eager and medium configurations seem to offer the best trade-off in terms of covered sequences and traced events.

7.6 | RQ3: Overhead

This research question investigates the overhead introduced by monitoring and tracing on the target service. To this end, we compute the memory and CPU consumption on the target service while running all the uses cases. We also compute the latency of the target service for every request during the execution of the use cases. We perform these measurements in three cases: when neither the monitor nor the tracing is active (Baseline), when only the monitor is active (Monitor), and when only the tracing is active (Tracing). We repeat all measurements 30 times for all services and report the collected values in the boxplots shown in Figures 11 and 12. We finally report in Table 7 the data about the size of the recorded traces for the executed scenarios, to discuss the feasibility of tracing.

We can clearly notice that neither the monitor nor tracing introduces any significant difference in both latency and CPU consumption for any of the services. This confirms the low intrusiveness of the approach. Tracing also has negligible impact on the memory consumption. Monitoring may have an impact due to the tracing profile and the extra software components that must be maintained in memory. While the impact is nearly nonobservable for PiggyMetrics services, it is noticeable for the Train Ticket services, which require a slightly different implementation of the monitor. Still, the absolute memory consumption of the data structure is affordable, since it consists of few megabytes only.

Table 7 shows the average size of the recorded traces for the six services after executing the identified scenarios when $N = 3$. Column *Uninterrupted Tracing* reports the average size of the trace with the baseline approach consisting of recording every interaction never stopping

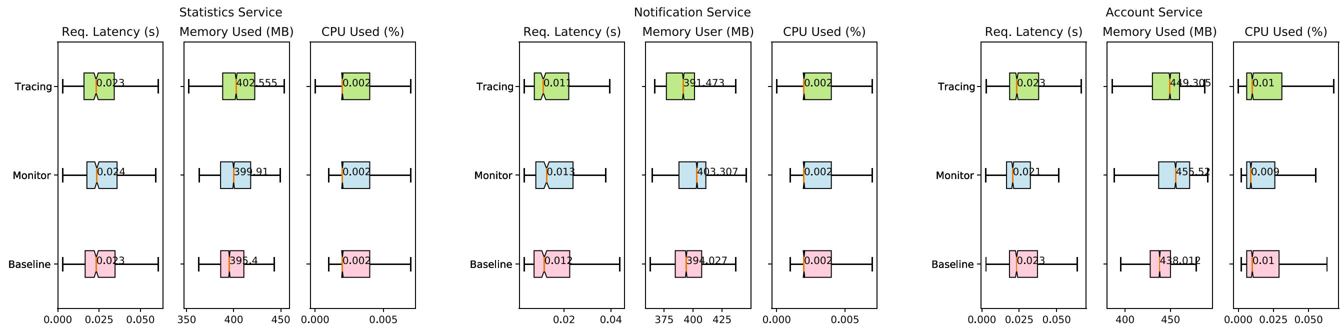


FIGURE 11 Overhead (latency, memory, and CPU) per service of the PiggyMetrics system

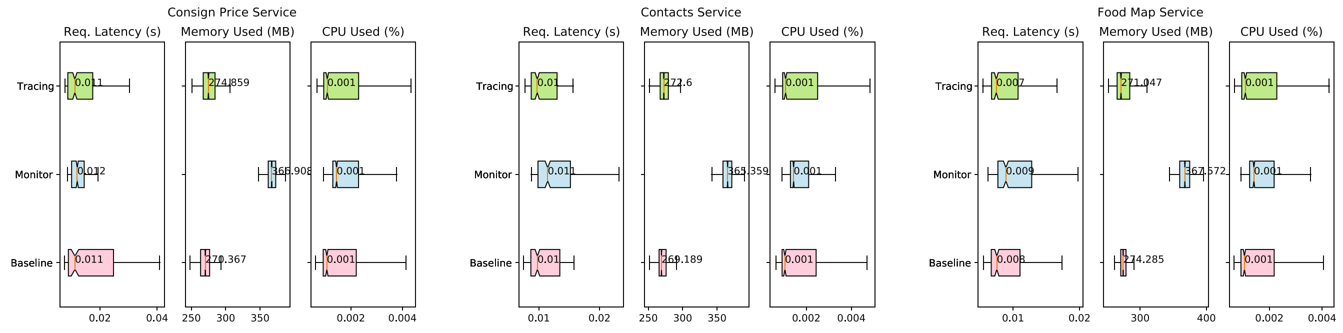


FIGURE 12 Overhead (latency, memory, and CPU) per service of the Train Ticket system

TABLE 7 Traces size

Service	Uninterrupted tracing	Scenario	Tracing configurations		
			Lazy	Medium	Eager
Account	101.34 GB	Regular	86.60 MB (0.083%)	96.45 MB (0.09%)	102.73 MB (0.10%)
		With random events	7.97 GB (7.86%)	12.30 GB (12.14%)	18.68 GB (18.43%)
		Overlapped	20.94 GB (20.66%)	34.78 GB (34.32%)	51.82 GB (51.14%)
Notification	16.83 GB	Regular	7.2 MB (0.042%)	8.29 MB (0.05%)	8.96 MB (0.05%)
		With random events	80.37 MB (0.47%)	78.92 MB (0.46%)	88.96 MB (0.52%)
		Overlapped	2.89 GB (17.2%)	4.90 GB (29.14%)	7.49 GB (44.53%)
Statistics	14.59 GB	Regular	6.9 MB (0.05%)	7.72 MB (0.05%)	8.13 MB (0.05%)
		With random events	74.13 MB (0.50%)	73.92 MB (0.49%)	87.39 MB (0.58%)
		Overlapped	2.53 GB (17.32%)	4.26 GB (29.19%)	6.45 GB (44.21%)
TS_Consign_Price	16.95 GB	Regular	7.74 MB (0.04%)	8.61 MB (0.05%)	8.78 MB (0.05%)
		With random events	2.81 GB (16.59%)	4.72 GB (27.84%)	7.01 GB (41.37%)
		Overlapped	3.52 GB (20.75%)	5.84 GB (34.43%)	8.68 GB (51.20%)
TS_Contacts	14.67 GB	Regular	7.63 MB (0.05%)	8.78 MB (0.06%)	9.39 MB (0.06%)
		With random events	1.23 GB (8.37%)	2.42 GB (16.52%)	3.96 GB (26.98%)
		Overlapped	3.07 GB (20.96%)	5.11 GB (34.81%)	7.59 GB (51.70%)
TS_Food_Map	23.84 GB	Regular	11.06 MB (0.045%)	12.19 MB (0.05%)	12.97 MB (0.05%)
		With random events	38.88 MB (0.16%)	42.30 MB (0.17%)	46.03 MB (0.19%)
		Overlapped	2.32 GB (9.75%)	4.32 GB (18.11%)	6.55 GB (27.46%)

tracing. Since all scenarios have the same length and the resulting size of the traces is always similar, we simply report average values. Considering that each scenario can be executed in few minutes, it is clear that indiscriminately collecting data in the field is infeasible. Indeed, uninterrupted tracing would quickly saturate disk occupation.

Column *Trace Configurations* reports the size of the trace recorded by ExVivoMicroTest considering the three configurations (lazy, medium, and eager) and the three scenarios (regular, with random events, and overlapped) investigated in the paper. The policies adopted by ExVivoMicroTest can dramatically reduce the size of the traces and the impact of recording. The magnitude of the reduction significantly depends on the level of variability of the behavior of the monitored service. In fact, when the behavior of the service is regular, ExVivoMicroTest can accurately sample the behavior of the services recording between 0.04% and 0.1% of the data compared with uninterrupted tracing. When the behavior has a degree of variability, ExVivoMicroTest has to record more data to capture this variability into the synthesized ex vivo test cases. Still, ExVivoMicroTest records between 0.16% and 41.37% of the data traced by uninterrupted tracing when addressing the with random events scenario, while ExVivoMicroTest records between 9.75% and 51.7% of the data traced by uninterrupted tracing when addressing the overlapped scenario. We expect that in the presence of a highly dynamic behavior, the rate of new observations incrementally decreases, and consequentially, the percentage of recorded data also tends to decrease while the service is executed.

There are instead small differences between the three configurations, with the overlapped scenario slightly maximizing these differences. Although the eager configuration records more data than the medium and lazy configurations, the size of the recorded traces mostly depends on the behavior of the service rather than the tracing configuration. This is an interesting property of ExVivoMicroTest. In fact, every configuration self-adapts tracing probabilities to reflect the actual behavior of the target service, ultimately collecting a similar number of traces. Since the overlapped scenario brings the higher variability in the behavior of the monitored service, ExVivoMicroTest tends to collect a higher number of traces (i.e., the tracing probability of the configurations is closer to p_{max}), thus resulting in slightly more evident differences among configurations.

7.7 | Findings

We can distill a few key findings from our empirical evaluation:

- *The frequency of update of the tracing profile has a mild impact on the cost-effectiveness of the approach*: The results obtained with RQ1 show how the update frequency can be changed by order of magnitudes having a small impact on the results. Developers can thus decide how frequently to download traces to enrich their ex vivo test suites (e.g., based on the size of trace files or the frequency of testing) with little concern on the impact of this decision on the effectiveness of the approach.

- *Services that show regular behaviors can be well addressed with ExVivoMicroTest regardless of their size*: The results obtained with RQ2 show that regular behaviors can be well addressed with ExVivoMicroTest that is able to quickly cover many sequences and then reduce the frequency of activation of the tracing, regardless of the size of the behavioral space of the service under test.
- *ExVivoMicroTest adapts its behavior to the microservice under test*: The results obtained with the covered and traced sequences show how ExVivoMicroTest can flexibly adapt the tracing frequency, and thus tune resource consumption, to the characteristics of the microservice under test. In case the behavioral space is large and irregular, ExVivoMicroTest tends to keep collecting traces to obtain a representative number of ex vivo test cases. The tracing profile is updated to collect fewer traces once the coverage improves.
- *Services with rare events can be challenging to address*: Rare events are not always well captured by the strategy used to update the probabilities in the tracing profile. As a consequence, tracing probabilities may decrease influenced by the most regular behaviors, causing ExVivoMicroTest to miss to capture several interesting rare events. This case requires more work to be accurately addressed with ex vivo testing solutions.
- *ExVivoMicroTest has a negligible impact on the target services*: The results obtained with RQ3 show how ExVivoMicroTest has little impact on the target service, not impacting on latency and CPU consumption, and marginally impacting on memory consumption. The amount of recorded information is also automatically balanced considering the behavior of the monitored service, drastically reducing the traced data compared to uninterrupted tracing.

7.8 | Threats to validity

Our study is affected by both internal and external threats to validity. The main internal threats to validity concern with the design of the executions used to run the services. To execute themservices in PiggyMetrics and Train Ticket, we identified a set of relevant use cases that have been combined, overlapped, and interleaved with random events, to obtain longer scenarios. Since the final scenarios consist of long executions obtained by combining the original use cases, we do not expect the choice of the use cases that may introduce major threats. Indeed, the use cases, and their recombination in the context of the scenarios, allowed us to obtain scenarios with a controlled level of variability that led to interesting findings.

It is part of our future work to experiment ExVivoMicroTest with extensive amount of real-users executions. We need, however, to address the privacy issues with the collected data before the approach can be applied to a larger scale with real user data. This paper focused on establishing the technical approach, therefore saving the privacy related issues for future work. Further, the behavior of ExVivoMicroTest is influenced by several parameters. In this paper, we either discuss the intuition behind our design choices or study the impact of parameters on the results to configure the approach (see RQ1). We cannot exclude that different choices may influence the results. The results obtained about the overhead (see RQ3) are influenced by the choices we made in terms of implementation. We cannot exclude that a better implementation could reduce the resource consumption of the monitoring and tracing components. However, this can be considered as a minor threat to the validity of the results, since the reported results already give evidence of the limited resources necessary to run ExVivoMicroTest.

The major threat to external validity concerns with the generalization of the reported results. Our evaluation is based on experiments with six services, so we cannot claim any generality for our findings. We mitigated this threat by considering services of different complexity and with different behaviorfrom two different systems, to obtain results that can be correlated to the characteristics of the service under test. Further investigation with other services and scenarios is anyway necessary to fully address this threat to validity.

7.9 | Limitations

Our approach and tool implementation have several limitations that we experienced during the evaluation and that must be discussed. First, ExVivoMicroTest assumes a mostly deterministic behavior of the tested operations. Services with operations that have a largely nondeterministic behavior cannot be automatically tested using ex vivo tests; otherwise, sporadic failures might be experienced, and testers may have to inspect failures due to nondeterminism rather than faults. Of course, if the nondeterministic operations are known, ExVivoMicroTest can be instructed to filter out the ex vivo tests with these operations and test services using deterministic operations only.

ExVivoMicroTest handles only stateless microservices, that is, a stateful microservice cannot be validated with the ex vivo test cases synthesized by ExVivoMicroTest. Note that it is possible to validate a stateless microservice that interacts with a stateful service. We consider this a minor limitation since modern microservice-based architectures are essentially composed of stateless microservices.

Our technique focuses on the synchronous interfaces offered by services (e.g., REST interfaces). Message-based asynchronous interfaces cannot be addressed with ExVivoMicroTest.

Finally, although ExVivoMicroTest can potentially address a range of protocols, our prototype implementation used in the evaluation collects data at the TCP/IP level and supports the `REST` and MongoWire protocols. We consider this a technical limitation but not a conceptual limitation of the approach that can be extended to address other protocols.

8 | RELATED WORK

The work presented in this paper relates to research in regression testing, profiling, microservice testing, and ex vivo testing. In this section, we position our contribution with respect to prior work in these related areas.

Regression testing techniques select the existing test cases that need to be re-executed to validate changes.[38–40] These test cases are used to validate if the features that are supposed to be unaffected by some changes are indeed behaving the same also after the changes. Regression testing techniques can identify the test cases to be executed according to different strategies, including strategies based on structural information,[39] on historical data,[41] on coverage of input parameters,[42] and on the cost-effectiveness of the process.[43] In some cases, it might be more convenient to prioritize test execution rather than selecting regression test cases.[44] For instance, Bryce et al[45] present a model that can support test prioritization of event-driven software. The proposed model is meant to be applicable to both GUI and Web applications.

Contrarily to these techniques, ExVivoMicroTest exploits actual field executions as the source to distill regression test cases. The resulting ex vivo test suites complement the in-house test suites with test cases that exercise the software according to the real execution scenarios that have been observed in the field.

To obtain the ex vivo test cases, ExVivoMicroTest records communication between services. Interactions between components have been already used as source of information for the regression testing process.[46] However, this information has been simply used in-house to select test cases from existing test suites, while ExVivoMicroTest implements a radically different approach that exploits field interactions to synthesize new regression test cases. Biagiola et al[47] considered diversity as a key factor in the generation of new test cases for Web applications starting from an initial test suite. While the context, target, and goal of the work are different, ExVivoMicroTest shares the intuition that new tests should be generated taking diversity into account, considering field information and a behavioral space model for ExVivoMicroTest.

When a specification is available, it is possible to generate particularly effective regression test cases. For instance, Godefroid et al[48] exploit Swagger specifications to deliver continuous differential regression testing of APIs, where differential testing is applied to detect regressions in the latest API version of a service before its deployment. ExVivoMicroTest does not require a specification of the service under test.

Profiling techniques[49,50] share a technical task with ExVivoMicroTest; that is, they also record field information. Nonetheless, the kind of recorded information and the objective are different. Profiling techniques record data useful to understand the field activity, while ExVivoMicroTest collects interactions useful to reproduce chunks of executions in-house. Moreover, profiling produces usage profiles of the software, while ExVivoMicroTest generates regression test suites that reuse actual usage data and scenarios. Related to profiling techniques, Milani et al[51] studies how to extend existing test suites by leveraging human knowledge jointly with the usage of a crawler. ExVivoMicroTest also implicitly exploits human knowledge, but it targets microservice applications.

Multiple techniques have been recently proposed to *test microservices*. For instance, Savchenko et al[52] studied cloud testing techniques, identifying the ones specific to microservices, while recognizing the aspects that characterizes microservices, such as isolation and standardized interfaces.

Rahman et al[53] defined an acceptance testing architecture for microservices in the context of Behavior-Driven development process. The proposed approach exploits a top level repository that includes acceptance tests for all the features that the individual microservices implement, to separate development and testing. Heorhiadi et al[54] presented a framework that systematically tests failure-handling capabilities of the individual microservices. Janes et al[55] studied how to use performance testing to provide feedback about the migration of a monolithic application to microservices.

Since ExVivoMicroTest also addresses microservices, some technical elements are shared with these techniques, such as the capability to test the microservices in isolation. However, none of these techniques address regression testing of microservices based on field executions, as ExVivoMicroTest does nor they provide the ability to opportunistically trace behaviors based on the monitored data.

Finally, *ex vivo testing* approaches exploit information extracted from the field to produce test cases that can be executed in-house, while taking field executions into consideration. The concept of *ex vivo testing* has been already exploited in several contexts. For instance, it has been used to test MapReduce applications based on the data computed in the field,[6] to test autonomous vehicles based on actual usages scenarios,[8] and in self-adaptive systems.[9] Furthermore field information can be exploited not only to synthesize test cases but also to influence their order of execution[56] or increase the number of tests can be executed across versions.[57] While the concept of ex vivo testing is similar for all these approaches, they are radically different on the technical point of view. ExVivoMicroTest originally proposes to use ex vivo testing for container-based applications.

9 | CONCLUSION

This paper addresses regression testing of microservice applications proposing ExVivoMicroTest, a technique that can be used to record service interactions from the field, which are turned into test cases that can be executed in-house on new versions of a microservice. Compared with regular test cases, the test cases generated by ExVivoMicroTest take the real usage scenarios into consideration, giving further validation opportunities, thus reducing the risk of experiencing failures that escaped validation. ExVivoMicroTest embeds policies to dynamically adjust the tracing probabilities that are used in the field based on the collected data, so that the cost of tracing events is scaled up and down based on the likelihood of collecting tests that cover new untested behaviors.

Our experience with the PiggyMetrics and Train Ticket services confirms the feasibility of the approach, revealing interesting trade-offs between the various configurations and different effectiveness based on the behavior of the service that is tested. In particular, services with regular or highly dynamic behaviors can be well addressed with ExVivoMicroTest, while services with rare behaviors are more challenging to test. This work provides initial evidence of how ex vivo tests could be complementary to in-house tests. Additional work must be done to thoroughly study and compare the fault-detection capability of in-house and ex vivo tests. In addition to addressing this challenge, our main research direction consists of considering privacy issues; that is, the data collected from the field may contain sensitive information that should not be visible to developers. We will thus investigate the usage and definition of obfuscation techniques that can hide the actual collected data without changing the executions.

ACKNOWLEDGMENTS

This work has been partially supported by the H2020 Learn project, which has been funed under the ERC Consolidator Grant 2014 program (grant agreement no. 646867), H2020 NGPaaS project (grant agreement no. 761557), and the GAUSS National Research Project, which has been funded by the MIUR under the PRIN 2015 program (contract 2015KWREMX). Open Access Funding provided by Universita degli Studi di Milano-Bicocca within the CRUI-CARE Agreement.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study, including the HTTP filters and the utility classes created to accomplish the monitoring task as well as the scripts used in the experiments, are openly available in the GitLab repository at https://gitlab.com/learnERC/exvivomicrotest, reference number.[58] The only exception is the test synthesis tool that is not available due to IPR restrictions.

ORCID

Leonardo Mariani https://orcid.org/0000-0001-9527-7042
Marco Mobilio https://orcid.org/0000-0002-3499-0159
Alessandro Tundo https://orcid.org/0000-0001-8840-8948

REFERENCES

1. Newman S. *Building microservices*: "O'Reilly Media"; 2015.
2. Savor T, Douglas M, Gentili M, Williams L, Beck K, Stumm M. Continuous deployment at Facebook and OANDA. In: Proceedings of the 38th International Conference on Software Engineering Companion; 2016.
3. Hunt N. Netflix and AWS lambda case study. Online: Accessed 11-02-2021; 2014.
4. Chen L. Microservices: Architecting for continuous delivery and DevOps. In: Proceedings of the IEEE International Conference on Software Architecture (ICSA); 2018.
5. Bertolino A, Braione P, Angelis GD, et al. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *ACM Comput Surv (CSUR)*. 2021;54(5):92:1-92:39.
6. Moran J, Bertolino A, de la Riva C, Tuya J. Towards ex vivo testing of MapReduce applications. In: Proceedings of the International Conference on Software Quality, Reliability and Security (QRS); 2017.
7. Elbaum S, Hardojo M. An empirical study of profiling strategies for released software and their impact on testing activities. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA); 2004.
8. Neves V, Delamaro ME, Masiero PC. Combination and mutation strategies to support test data generation in the context of autonomous vehicles. *Int J Embed Syst (IJES)*. 2016;8(5/6):464-482.
9. Fredericks EM, DeVries B, Cheng BHC. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In: Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS); 2014.
10. Gazzola L, Goldstein M, Mariani L, Segall I, Ussi L. Automatic ex-vivo regression testing of microservices. In: Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test (AST); 2020.
11. PiggyMetrics Contributors. Piggymetrics. Online: Accessed 11-02-2021; 2021.
12. Train Tricket Contributors. Train ticket. Online: Accessed 11-09-2021; 2021.
13. Docker I. Docker. Online: Accessed 11-02-2021; 2021.
14. The Linux Foundation. Kubernetes. Online: Accessed 11-02-2021; 2021.
15. Dallmeier V, Lindig C, Wasylkowski A, Zeller A. Mining object behavior with ADABU. In: Proceedings of the International Workshop on Dynamic Analysis (WODA); 2006.
16. Marchetto A, Tonella P, Ricca F. ReAjax: A reverse engineering tool for AJAX web applications. *IET Softw*. 2012;6(1):33-49.
17. Lo D, Khoo S-C, Han J, Liu C. *Mining Software Specifications: Methodologies and Applications*: CRC Press; 2011.
18. Lo D, Maoz S. Scenario-based and value-based specification mining: Better together. *Autom Softw Eng*. 2012;19(4):423-458.
19. Nie C, Leung H. A survey of combinatorial testing. *ACM Comput Surv*. February 2011;43(2):1-29.

20. Ding X, Huang H, Ruan Y, Shaikh A, Peterson B, Zhang X. Splitter: A proxy-based approach for post-migration testing of web applications. In: Proceedings of the European Conference on Computer Systems; 2010.
21. Baeldung. How to define a spring boot filter? Online: Accessed 11-02-2021; 2020.
22. Elasticsearch BV. ELK stack: Elasticsearch, Logstash, Kibana. Online: Accessed 11-02-2021; 2021.
23. The Linux Foundation. Prometheus. Online: Accessed 11-02-2021; 2021.
24. Hewlett-Packard Enterprise Development LP. Monasca. Online: Accessed 11-02-2021; 2017.
25. Shatnawi A, Orru M, Mobilio M, Riganelli O, Mariani L. Cloudhealth: A model-driven approach to watch the health of cloud services. In: Proceedings of the 1st International Workshop on Software Health; 2018.
26. Tundo A, Mobilio M, Orr M, Riganelli O, Guzman M, Mariani L. VARYS: An agnostic model-driven monitoring-as-a-service framework for the cloud. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering(ESEC/SIGSOFT FSE), Tool Demo Track; 2019.
27. The Tcpdump Group. Tcpdump/libpcap. Online: Accessed 11-02-2021; 2021.
28. Santos W. Programmableweb's most popular APIS of 2017. Online: Accessed 11-02-2021; 2018.
29. DOR Green and contributors. Pyshark. Online: Accessed 25-09-2021; 2021.
30. Wireshark Foundation. tshark. Online: Accessed 25-09-2021; 2021.
31. Brandon Byars and Contributors. Mountebank—Over the wire test doubles. Online: Accessed 24-09-2021; 2021.
32. MongoDB IL. mongoreplay. Online: Accessed 24-09-2021; 2021.
33. restfulapi.net. REST api. Online: Accessed 11-02-2021; 2020.
34. MongoDB I. Mongodb wire protocol. Online: Accessed 11-02-2021; 2021.
35. Zhou X, Peng X, Xie T, et al. Delta debugging microservice systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018; 2018:802-807.
36. Zhou X, Peng X, Xie T, et al. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019; 2019:683-694.
37. Zhou X, Peng X, Xie T, Sun J, Ji C, Li W, Ding D. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Trans Softw Eng*. 2018;47(2):24.
38. Bible J, Rothermel G, Rosenblum DS. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans Softw Eng Methodol*. 2001;10(2):149-183.
39. Rothermel G, Harrold MJ. Analyzing regression test selection techniques. *IEEE Trans Softw Eng*. 1996;22(8):529-551.
40. Legunsen O, Shi A, Marinov D. Starts: Static regression test selection. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering; 2017.
41. Kim J-M, Porter A. A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th International Conference on Software Engineering (ICSE); 2002.
42. Cohen DM, Dalal SR, Fredman ML, Patton GC. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans Softw Eng*. 1997; 23(7):437-444.
43. Miranda B, Cruciani E, Verdecchia R, Bertolino A. Fast approaches to scalable similarity-based test case prioritization. In: Proceedings of the 40th International Conference on Software Engineering; 2018:222-232.
44. Hemmati H. Advances in techniques for test prioritization. *Advances in Computers*, Vol. 112: Elsevier; 2019:185-221.
45. Bryce RC, Sampath S, Memon AM. Developing a single model and test prioritization strategies for event-driven software. *IEEE Trans Softw Eng*. 2010; 37(1):48-64.
46. Mariani L, Papagiannakis S, Pezze M. Compatibility and regression testing of cots-component-based software. In: Proceedings of the International Conference on Software Engineering (ICSE); 2007.
47. Biagiola M, Stocco A, Ricca F, Tonella P. Diversity-based web test generation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2019:142-153.
48. Godefroid P, Lehmann D, Polishchuk M. Differential regression testing for rest APIS. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis; 2020:312-323.
49. Wei X, Gomez L, Neamtiu I, Faloutsos M. Profiledroid: Multi-layer profiling of android applications. In: Proceedings of the Annual International Conference on Mobile Computing and Networking (Mobicom); 2012.
50. Elbaum S, Diep M. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans Softw Eng (TSE)*. 2005;31(4):312-327.
51. Milani Fard A, Mirzaaghaei M, Mesbah A. Leveraging existing tests in automated test generation for web applications. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering; 2014:67-78.
52. Savchenko DI, Radchenko GI, Taipale O. Microservices validation: MJOLNIRR platform case study. In: 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) IEEE; 2015:235-240.
53. Rahman M, Gao J. A reusable automated acceptance testing architecture for microservices in behavior-driven development. In: 2015 IEEE Symposium on Service-Oriented System Engineering IEEE; 2015:321-325.
54. Heorhiadi V, Rajagopalan S, Jamjoom H, Reiter MK, Sekar V. Gremlin: Systematic resilience testing of microservices. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS) IEEE; 2016:57-66.
55. Janes A, Russo B. Automatic performance monitoring and regression testing during the transition from monolith to microservices. In: 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW); 2019:163-168.
56. Sampath S, Bryce RC, Viswanath G, Kandimalla V, Koru AG. Prioritizing user-session-based test cases for web applications testing. In: Proceedings of the International Conference on Software Testing Verification and Validation (ICST); 2008.
57. Alshahwan N, Harman M. Automated session data repair for web application regression testing. In: Proceedings of International Conference on Software Testing, Verification, and Validation (ICST); 2008.
58. [dataset]Gazzola L, Goldstein M, Mariani L, Mobilio M, Segall I, Tundo A, Ussi L. Exvivomicrotest replication package; 2021. https://gitlab.com/learnERC/exvivomicrotest

SUPPORTING INFORMATION

Additional supporting information may be found in the online version of the article at the publisher's website.

How to cite this article: Gazzola L, Goldstein M, Mariani L, et al. ExVivoMicroTest: ExVivo Testing of Microservices. *J Softw Evol Proc*. 2023;35(4):e2452. doi:10.1002/smr.2452