Contents lists available at ScienceDirect

Information and Software Technology

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

Flakiness goes live: Insights from an In Vivo testing simulation study

Morena Barboni [a,c], Antonia Bertolino [b], Guglielmo De Angelis [a,*]

[a] IASI–CNR, Rome, Italy
[b] ISTI–CNR, Pisa, Italy
[c] University of Camerino, Camerino, Italy

ARTICLE INFO

Dataset link: https://github.com/IASI-SAKS/groucho/releases/tag/FastJsonInvivoFlakiness-v1.0, https://github.com/gulyx/jvm-serializers/tree/fastJSON-benchmark-only

MSC:
68-04
68-U01

Keywords:
Flaky test
Intermittent failure
Field testing
In Vivo testing
Software testing

ABSTRACT

Context: Test flakiness is a topmost concern in software test automation. While conducting pre-deployment testing, those tests that are flagged as flaky are put aside for being either repaired or discarded.
Objective: We hypothesise that some flaky tests could provide useful insights if run in the field, i.e., they could help identify failures that manifest themselves sporadically during In House testing, but are later experienced in operation.
Method: We present the first simulation study to investigate the behaviour of flaky tests when moved to the field. The work compares the behaviour of known flaky tests from an open-source library when executed in the development environment vs. when executed in a simulation of the field.
Results: Our experimentation over 52 test methods labelled as flaky provides a first confirmation that moving from the development environment to the field, the behaviour of tests changes. In particular, the failure frequency of intermittently failing tests can increase, and we could also identify few cases of field failures that would have been hardly detected during In House testing due to the numerous combinations of inputs and states. In most cases, such flakiness was rooted in the design of the test method itself, however we could also identify an actual bug.
Conclusion: The results of our study suggest that the identification of an intermittently failing behaviour could be a valuable hint for a test engineer, and hence flaky tests should not be dismissed right away.

1. Introduction

The problem of flaky tests, i.e., tests that present intermittent outcomes when re-executed, is increasingly drawing researchers' attention, as testified in a recent systematic review of scientific literature [1]. Far from being a purely academic concern though, test flakiness is in the first place a diffuse practitioners' pain. In his ICST 2017 keynote titled "The state of continuous integration at Google",[1] Micco reported that almost 16% of their 4.2M tests showed some level of flakiness, which blocked or delayed releases. Some years later, Pirocanac wrote a Google Testing blog referring to test flakiness as *one of the main challenges of automated testing.*[2] In Microsoft [2], data collected over five projects for 30 days to analyse the prevalence of flaky tests showed that, across those projects, the percentage of builds in which flaky tests were observed varied between 14% up to 52%. In Meta, a recent blog by Machalica et al.[3] deals with how to test the tests themselves, given that in their experience "all *real-world tests are flaky to some extent, even if they are implemented following best engineering principles*".

In short, flakiness is a prevalent software testing challenge both in theory and in practice, which is normally depicted, in both white and gray literature, as a purely negative phenomenon to eradicate, or at least to mitigate. In a recent survey [3], many developers qualify flaky tests as a non-negligible problem, which may impact resource allocation, test scheduling, and test suite reliability. In his popular blog, Fowler [4] calls them useless and warns that they could infect the entire test suite.

While acknowledging the perils of flakiness, in this work we take a different position: we insinuate that not all flaky tests are necessarily detrimental, on the contrary, we hypothesise that some flaky tests could be useful resources for the developers. We think this because in some cases test flakiness could stem from an elusive bug in the code under test that is only triggered in specific circumstances; in other words, some flaky tests could be symptoms of *hard-to-detect* failures [5].

Hard-to-detect failures escape In House testing, which is performed in the development environment, but are later eventually experienced

* Corresponding author.
E-mail addresses: morena.barboni@unicam.it (M. Barboni), antonia.bertolino@isti.cnr.it (A. Bertolino), guglielmo.deangelis@iasi.cnr.it (G. De Angelis).

[1] Slides available from http://aster.or.jp/conference/icst2017/program/jmicco-keynote.pdf (accessed on Feb 07, 2022).
[2] See at https://testing.googleblog.com/2020/12/test-flakiness-one-of-main-challenges.html (accessed on Feb 07, 2022).
[3] See at https://engineering.fb.com/2020/12/10/developer-tools/probabilistic-flakiness/ (accessed on July 19th, 2022).

https://doi.org/10.1016/j.infsof.2023.107373
Received 5 August 2022; Received in revised form 18 October 2023; Accepted 20 November 2023
Available online 23 November 2023

in the field (i.e., in production). Gazzola et al. [5] investigated their characteristics and potential causes, finding that the faults causing such field failures: (i) are inherently impossible to activate In House, or (ii) descend from unknown conditions, or (iii) depend on "uncountable" many configurations (combinatorial explosion). Since *hard-to-detect* failures might be triggered by unpredictable combinations of settings and events, they might manifest sporadically during In-House testing campaigns, causing some tests to exhibit non-deterministic pass and failure outcomes. However such tests are likely to be dismissed, as test engineers are more inclined to ignore a test that fails sporadically. This inclination to ignore sporadic test failures is rooted in established industry practices, as highlighted by a recent multivocal review on test flakiness [6]. Facebook's Probabilistic Flakiness Score (PFS) [7] and GitHub's impact score [8] are examples of metrics used to quantify the level of flakiness and prioritise fixes. Sporadically failing tests may receive lower severity ratings, relegating them to a lower priority for resolution. Interviews with practitioners conducted by Habchi et al. [9] also highlight the importance of the flake rate in guiding decisions and prioritising tests for immediate attention. As a result, sporadic failures, which could be indicative of hard-to-detect issues, may go unaddressed, potentially leading to more significant problems when the software is deployed in the field.

Our research aims to discover if some flaky tests seen in the laboratory can be useful for a test engineer, because their intermittent behaviour could actually be the symptom of future field failures. One way to do this is by moving the testing activities to the field and looking for behavioural changes in the flaky tests. In the last decade, many approaches are being proposed to continue the testing in production [10], referred to as *In Vivo testing*. In our vision, running the flaky tests In Vivo means taking advantage of many different configurations and object states that naturally occur during production usage, and that cannot be fully explored during In House testing.

In Vivo studies, however, are notoriously complex to set up, as they would require a real production system to experiment with. Considering the practical difficulties in accessing a system in operation, we instead reproduce a production-like environment in which we can stimulate the System Under Test (SUT). We can then rely on simulated data to run the tests with different parameters (and from different starting states), so as to expose hidden variability and assess which conditions might lead to a test failure. To carry out one such study, we need to adapt the test cases designed for execution In House to be launched In Vivo, precisely we do not try to reproduce in the field a setting in which the test can be launched (which would be counter-intuitive), but transform the test into a parametric format, thus making it executable from *wherever status it is launched during the simulation*.

This work provides several contributions. Conceptually, we introduce here and investigate for the first time the hypothesis that flaky tests could be useful to hint at *hard-to-detect* failures. Technically, we provide: (i) a detailed illustration of the steps for setting up an In Vivo testing simulation study; (ii) the first simulation study comparing the behaviour of known flaky tests from the Fastjson library (our subject of study, described in Section 4.1) when executed In House vs. In Vivo; (iii) a replication package with the full data and software.

The paper proceeds as follows: in Section 2 we provide background notions and in Section 3 present the research questions we aim to answer. The study subjects are shown in Section 4, while Section 5 and Section 6 describe the methodology used for this study and the experimental set-up, respectively. In Section 7 we discuss the results of our study, and potential threats to validity are addressed in Section 8. Finally, Section 9 outlines related work and Section 10 gives a brief summary of the findings and identifies areas for further research.

2. Background

In this section, we introduce some notions related to the testing-related topics that we combine: test flakiness (Section 2.1) and a framework (Section 2.3) supporting testing In Vivo (Section 2.2).

2.1. Software test flakiness

Test flakiness has received a lot of attention from researchers and industry practitioners in recent years. However, while all agree on the relevance of the problem, the same term "flaky test" may be used to refer to many different natures of non-deterministic behaviour of test cases [1,3]. Our own recent review of literature [11] showed clearly that no shared formalised definition yet exists to which we can refer.

To further complicate things, we attempt here to study the characteristics of this phenomenon in a novel context that was never addressed in the literature before (and we ourselves encountered some difficulties in mapping the notion of flakiness to this novel context). Thus, before expressing our research questions, we must clearly state what we consider to be a flaky test, and how this concept evolves for (a simulation of) In Vivo testing, aiming at leveraging the operating conditions in production.

According to our review [11], the most widely accepted definitions for a **flaky** test are (1) *"A test that exhibits pass and failure outcomes despite exercising unchanged code"* and (2) *"A test that exhibits pass and failure outcomes although neither the code nor the test has changed"*. Such definitions are also confirmed by the outcomes of a recent empirical study surveying views and experiences of software developers about flaky tests [12]. Thus, when we describe a test as flaky, we imply that both the test *and* the exercised code stay the same across different executions.

As we better explain in Section 5, an In House test generally requires some modifications to become executable In Vivo. In particular, the method must be made parametric so that it can take advantage of the mutable execution context. We must therefore employ a different, "looser" definition of test flakiness for describing the outcomes of such tests. In this sense, the definition that better fits our requirements is the one introduced in Strandberg et al.'s study about test flakiness in the embedded systems domain [13]. In this work, the authors define an **Intermittently Failing** test as "*A test case that has been executed repeatedly while there is a potential evolution in software, hardware or testware, and where the verdict changes over time*". These authors distinguish between intermittently failing and flaky tests, in that the latter are more specific and do not allow changes in the SUT or in the testware, which they instead allow for the former. According to the proposed definition, flaky tests are a subset of intermittently failing tests, i.e., a flaky test is also an intermittently failing test, but not vice versa.

For the rest of this paper, we use the term intermittently failing to denote the potential non-deterministic behaviour of In Vivo tests. According to the above explanation, to find out whether an In Vivo test for which we observed an intermittently failing behaviour is flaky as well, it must be run In House.

2.2. Field testing

Several works proposed during the last years by independent research groups advocated the need to extend testing activities beyond the boundaries of both software development and deployment phases. A recent survey on field-based testing [14] set out to provide a consistent terminology for addressing such kinds of activities. Here, we recall such definitions and provide some background to better contextualise the objectives of this study.

Firstly, the term **Field** (or **In Vivo**) testing encompasses those techniques that aim to *validate the behaviour of an application in the* ***field***, that is the set of all its (possible) production environments. The **production environment** of an application can be further defined as *"any environment where the application can be fully operational"*.

As opposed to In Vivo testing, **In House testing** activities are *performed in a testing environment completely separated from production.* Sometimes, these can also be empowered by information gathered during the actual usage of the system (e.g. [15–17]) in which case we talk about **Ex-vivo** software testing.

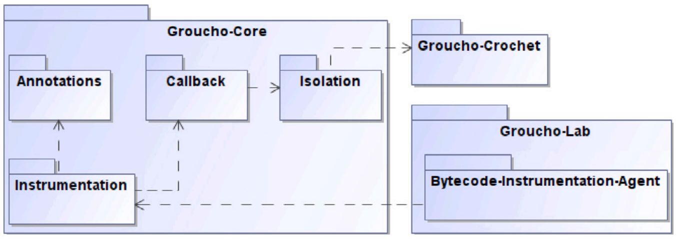

Fig. 1. Groucho: High level architecture.

Often testing in the field is motivated by the opportunity of leveraging the participation of a wide number of actors/machines. Indeed, it permits to enact test invocations on the SUT directly in its operational environment, in some cases even while it is serving actual requests. Such an opportunity should allow for testing the SUT under many different configurations which are generally difficult when applying traditional testing approaches.

2.3. Supporting framework for In Vivo testing

Groucho [10] is a framework for In Vivo testing of Java-based SUTs that allows a tester to control and safely execute tests at run-time, when the SUT is running in its operational environment. Groucho leverages flexible Aspect-Oriented (AO) solutions to modify the SUT so as to include both the In Vivo testing features and the test logic to be processed In Vivo.

The high-level architecture of the framework is depicted in Fig. 1: its core part is structured around four main building blocks grouped in the logical package `Groucho-Core`, while other supplemental functionalities are organised in the logical packages: `Groucho-Crochet` and `Groucho-Lab`. Specifically, the `Annotations` block defines the meta-information that can enable or regulate the In Vivo testing activities.

`Instrumentation` encapsulates the constructs controlling the activation of testing aspects while the SUT is running in its production environment. Specifically, the instrumentation of the SUT relies on the Java Annotations technology: any instance of the annotation `@TestableInVivo` declares the methods to be subject to In Vivo testing and, among its parameters, specifies the test program to be executed when an In Vivo testing session is launched. The association of the annotation `@TestableInVivo` with some methods in the SUT can be performed either at the source-code level or by means of on-the-fly manipulation of plain SUT binaries (i.e., the `Bytecode-Instrumentation-Agent` in Fig. 1), including 3rd party or legacy libraries [18].

The In Vivo interactions between the SUT and Groucho are structured according to the callback design pattern. `Callback` is also responsible for the activation of the isolation policies, and the execution of the test cases actually codified as test programs.

Groucho structures the common API for the isolation mechanisms within `Isolation`. Once an In Vivo testing session has been enabled, the Groucho framework relies on lightweight isolation mechanisms that allow testers to checkpoint the current state of the SUT, then to test it, and finally to restore the SUT back to its state before the activation of the In Vivo testing session. Groucho distinguishes between two main approaches to isolation depending on whether the data are stored in-memory or persisted/exported out-of-memory. In the former case, `Groucho-Crochet` automatically grants for a secure preservation of in-memory data by performing lazy deep copies of the objects involved in the testing session; all other objects are left unaffected. The copying relies on lightweight technologies able to introduce a negligible impact in terms of both consumed resources and overhead. The in-memory isolation policies that are automatically enabled by `Groucho-Crochet` aim to prevent any field/user data corruption due to the In Vivo testing session. Testers planning to execute multiple independent test cases starting from the same run-time state can programmatically request `Isolation` to create (and then to revoke) additional isolation layers within the same In Vivo testing sessions. The in-memory isolation is native for multi-thread applications without any extra responsibility for the In Vivo testers. In the latter case (persisted/exported out-of-memory data), Groucho does not offer any general solution. Specifically, while it offers some abstract primitives and high-level recommendations, Groucho delegates to In Vivo testers the responsibility to define SUT-specific strategies for preserving out-of-memory data possibly affected during In Vivo testing sessions.

Within the scope of this work, we only referred scenarios with in-memory data persistence and we only rely on the checkpoint/rollback strategy offered by `Groucho-Crochet`. Under these conditions, it has been proved that the overhead introduced by Groucho is very low or negligible when the In Vivo test activation probability is kept reasonably small (e.g., 10–4) [10].

Further technical details about the architecture, use cases, usage scenarios, detailed performance analysis, and on-the-fly manipulation of the byte code can be found in [10], or in [18].

3. Objectives and research questions

This study sets the scene for the long-term goal of recognising in future those types of flakiness that can help identify hard-to-detect bugs and prevent future operational failures. In the following, we describe the research questions that we address in this work. Note that, for the sake of brevity, we use the term In Vivo (or, more generally, *in the field*) to denote the simulation study we performed.

RQ1: *How do flaky tests behave when they are run using inputs from the field, and, more specifically, how does their outcome vary with respect to executing them In House?*

This study aims to understand how flaky tests behave if the context changes from the tester's defined configuration, while its inherent objectives, preconditions and expected outcome stay the same. Table 1 summarises the scenarios that we might observe when simulating an In Vivo testing session. In this work we only consider tests that start from a flaky state (i.e., they were marked as flaky either by a developer or by a flakiness detection tool), and we execute them both In House and In Vivo. The rows report the observable combinations of test outcomes In House (Col. 2) and In Vivo (Col. 3) and the relative case ID (Col. 4). Moving a test to the field might give us different insights depending on the scenario: for instance, a test that consistently passes on our machine, despite being marked as flaky, might indicate the presence of a *hard-to-detect* bug. By moving the test to the field we can re-run it in different states and pinpoint the conditions under which it possibly fails.

Note that the labelling introduced in Table 1 are limited to the specific outcomes we observed from our empirical study. Hence, our terms *Consistently Passing* or *Consistently Failing* only express that the tests (either In House or In Vivo) behaved stably across all the runs we launched. Such a labelling should not be considered as a new assessment of the flakiness of the tests; indeed the whole study stems from the fact that all the tests are already known to be flaky (i.e., as detailed in Section 4.1).

RQ2: *Can In Vivo testing impact on the perception of flakiness, and, more specifically, do the flaky tests fail more frequently In Vivo than In House?*

In the hypothesis that some intermittent test behaviour could be originated by a *hard-to-detect* bug, by this question we aim to understand whether running In Vivo a test labelled as flaky could help to more quickly uncover the hidden source of flakiness, by amplifying its manifestation. In fact, several studies have shown that some flaky tests must be re-run "very many times" before being detected [19]. From a practical point-of-view, flaky tests that fail quite rarely are less

Table 1
Test Scenarios.

Marked as	Observed outcome in house	Observed outcome in vivo	Case ID
Flaky	Intermittently Failing	Intermittently Failing	IF-2-IF
		Consistently Passing	IF-2-CP
		Consistently Failing	IF-2-CF
	Consistently Passing	Intermittently Failing	CP-2-IF
		Consistently Passing	CP-2-CP
		Consistently Failing	CP-2-CF
	Consistently Failing	Intermittently Failing	CF-2-IF
		Consistently Passing	CF-2-CP
		Consistently Failing	CF-2-CP

interesting than frequently failing ones, and tend to be ignored [6]. For example, at Apple [20] a flakiness scoring service has been introduced, which quantifies flakiness and is referred to support flaky handling tasks. In this sense, if their frequency of failure is indeed amplified, moving flaky tests to the field could contribute to more properly evaluate their relevance. To answer this question, we hence introduce a measure of the degree of non-determinism of a test, which captures how frequently a flaky test fails when re-run, i.e., intuitively the perception of flakiness. Precisely, we define the notion of *Frequency of Failure* (i.e., FoF), which measures the percentage of observed failures of a test over the total number of re-runs.

***RQ3**: Can some flaky tests be useful indicators of hard-to-detect failures?*

Flakiness might stem from badly designed test code, but it could also be a symptom of a complex and hard-to-detect bug. These are completely different situations that require different treatments. In particular, by this question we aim to find initial evidence of the potential usefulness of some flaky tests in revealing hard-to-detect failures.

4. Study subjects

In this section, we introduce the subject on which we conducted our study of flakiness in the field. To do this we need both a SUT which exposes a useful set of flaky tests (described in Section 4.1) *and* a benchmark application through which we can simulate the invocation of the SUT in operation (described in Section 4.2).

4.1. SUT: Fastjson

To find a viable subject for our experiment we compiled a list of criteria that should be respected by the target project:

- **C1**: the project must be an open-source Java product;
- **C2**: the project must have a test suite with several flaky tests of various nature (e.g., Order-Dependent, Implementation Dependent).
- **C3**: there must exist an open-source, independent benchmark application for stimulating the project.

We specified C1 as Groucho supports In Vivo testing of Java applications, while C2 ensures that we run the experiment on a reasonably-sized sample of heterogeneous flaky tests. Lastly, C3 helps to avoid bias and to improve the reproducibility of the study.

We then searched the **International Dataset of Flaky Tests**[4] (IDoFT) for a project that meets the aforementioned criteria. This popular repository catalogues flaky tests found in real-world projects, providing a variety of information about each method. The database is broadly split into a list of unfixed tests and a list of fixed ones, together with the relative project, category, and eventual Github pull request. Many of the reported tests were marked either as Implementation Dependent (ID) or as a type of Order Dependent (OD), however other categories such as Non Deterministic Order Dependent (NDOD) and Independent (NDOI) - as defined in [21] - can be found as well. The database includes contributions from state-of-the-art tools such as IDFlakies [22] and iFixFlakies [23] and has represented an important resource for the research community.

After manually analysing all the available projects we found 3 candidates: Biojava,[5] wildfly[6] and Fastjson.[7] These three Java applications include several flaky methods belonging to different categories. wildfly has 86 flaky tests, most of which are Implementation or Order Dependent. Fastjson includes 63 flaky tests, again mostly ID or OD with the exception of a few methods belonging to the Non Order Dependent (NOD) category. Lastly, Biojava includes 52 flaky tests, however none of them were marked as OD making this project less representative. Overall wildfly has the biggest test set to experiment with, but only the Fastjson project also complies with C3. Further motivations about the exclusion of both Biojava and wildfly are discussed in Section 7.6

Fastjson is a popular Java library for converting Java Objects into their JSON representation (and vice versa). Its methods, such as `toJSONString()` and `parseObject()`, allow to quickly convert arbitrarily complex objects (with deep inheritance hierarchies and extensive use of generic types). Since the system states and configurations observed in production might substantially differ from the ones explored In House, Fastjson makes a suitable subject for our experiment. Besides, its Github repository currently features over 25k stars, 6k forks, and 170 contributors, testifying its extensive usage by the developer community. Table 2 lists the subjects of our experiment. This initial test pool includes 52 out of the 63 flaky methods of the Fastjson project collected in the IDoFT database. 11 methods were excluded from the experiment due to some configuration issues that prevented us from re-running them locally. Specifically, for each flaky test method belonging to a certain class, the table reports a unique ID and the considered project version (V.). The latter indicates the Fastjson release for which each considered test can exhibit a flaky behaviour. This information was derived from the data (i.e., GitHub issue or flakiness-fixing commit) provided by the IDoFT database. Lastly, the table shows the specific category (Cat.) of flakiness attributed to each test method by the IDoFT database. As can be seen, the flaky tests of Fastjson that we were able to re-run locally can be broadly split into two groups: **Implementation-Dependent (ID)** and **Order-Dependent (OD)**. The former includes 37 tests detected by *NonDex* [24]. These are tests that might behave non-deterministically due to ADINS code, that is, code that Assumes a Deterministic Implementation of a method with a Non-deterministic Specification. The remaining 15 tests were marked as Order-Dependent, that is, tests whose only source of non-determinism is order dependency [22]. This group includes 11 tests that were further classified as **Brittle (OD-Brit)** and 2 tests that were marked as **Victim (OD-Vic)**. As specified by Shi et al. [23], a Brittle is a test that fails when run in isolation but passes when run with some other test(s); a Victim, on the other hand, passes when run in isolation but fails when run with some other test(s). These can be used for In Vivo testing activities after some adaptation steps described in Section 5.

4.2. Benchmark: `jvm-serializers`

In Vivo testing consists of launching testing sessions in the production environment [14]. As we cannot rely on real end-user sessions for our experiments, we must retrieve an application for stimulating the SUT and reproducing a production-like environment.

As reported above, C3 restricted the SUT selection to those case studies in IDoFT for which known independent applications were available. Fastjson's GitHub page mentions `jvm-serializers`[8] : a tool for estimating and comparing the performance of several JSON processors.

[4] http://mir.cs.illinois.edu/flakytests

[5] https://github.com/biojava/biojava

[6] https://github.com/wildfly/wildfly

[7] https://github.com/alibaba/fastjson

[8] https://github.com/eishay/jvm-serializers

Table 2
Experimental subjects.

ID	V.	Test class	Test method	Cat.
(1)	1.2.73	ArrayListMultimapTest	test_for_multimap	ID
(2)	1.2.78	Bug_for_issue_447	test_for_issue	ID
(3)	1.2.62	Bug_for_smoothrat6	test_set	ID
(4)	1.2.75	Bug_for_xiayucai2012	test_for_xiayucai2012	ID
(5)	1.2.73	Bug_for_yangzhou	test_for_issue	ID
(6)	1.2.57	DateParseTest9	test_dates_different_timeZones	ID
(7)	1.2.54	DateTest	test_date	OD-Brit
(8)	1.2.54	DateTest_tz	test_codec	OD-Brit
(9)	1.2.54	DateTest4_indian	test_date	OD-Brit
(10)	1.2.54	DateTest5_iso8601	test_date	OD-Brit
(11)	1.2.54	DefaultExtJSONParser_parseArray	test_7	OD-Brit
(12)	1.2.54	DefaultExtJSONParser_parseArray	test_8	OD-Brit
(13)	1.2.73	FeatureCollectionTest	test_geo	ID
(14)	1.2.78	HashMultimapTest	test_for_multimap	ID
(15)	1.2.51	Issue_717	test_for_issue	OD
(16)	1.2.73	Issue1177_1	test_for_issue	ID
(17)	1.2.54	Issue1177_2	test_for_issue	ID
(18)	1.2.54	Issue1298	test_for_issue	OD-Brit
(19)	1.2.54	Issue1298	test_for_issue_1	OD-Brit
(20)	1.2.73	Issue1363	test_for_issue	ID
(21)	1.2.73	Issue1363	test_for_issue_1	ID
(22)	1.2.78	Issue1368	test_for_issue	ID
(23)	1.2.54	Issue1480	test_for_issue	ID
(24)	1.2.73	Issue1492	test_for_issue	ID
(25)	1.2.75	Issue1493	test_for_issue	ID
(26)	1.2.78	Issue1584	test_for_issue	ID
(27)	1.2.54	Issue1679	test_for_issue	OD-Brit
(28)	1.2.51	Issue1769	test_for_issue	OD-Brit
(29)	1.2.73	Issue1780_JSONObject	test_for_issue	ID
(30)	1.2.73	Issue1780_Module	test_for_issue	ID
(31)	1.2.73	Issue1972	test_for_issue	ID
(32)	1.2.54	Issue1977	test_for_issue	OD-Brit
(33)	1.2.75	Issue2428	test_for_issue	ID
(34)	1.2.73	Issue2447	test_for_issue	ID
(35)	1.2.73	Issue2447	test_for_issue2	ID
(36)	1.2.78	Issue3082	test_for_issue	ID
(37)	1.2.78	Issue3655	test_inherit_from_abstract_class_1	ID
(38)	1.2.78	Issue3655	test_inherit_from_abstract_class_2	ID
(39)	1.2.51	JSONFieldTest5	test_jsonField	OD
(40)	1.2.73	JSONObjectTest_readObject	test_6	ID
(41)	1.2.73	JSONPath_reverse_test	test_reserve	ID
(42)	1.2.73	JSONPath_reverse_test	test_reserve3	ID
(43)	1.2.54	JSONPParseTest2	test_f	ID
(44)	1.2.54	JSONPParseTest3	test_f	ID
(45)	1.2.51	JSONSerializerTest2	test_0	OD-Vic
(46)	1.2.78	MaxBufSizeTest	test_max_buf	OD-Vic
(47)	1.2.78	SortFieldTest	test_1	ID
(48)	1.2.73	SqlDateDeserializerTest2	test_sqlDate	ID
(49)	1.2.75	TypeUtilsTest	test_cast_to_Timestamp_1970...	ID
(50)	1.2.73	WriteClassNameTest_Map	test_list	ID
(51)	1.2.54	WriteDuplicateType	test_dupType2	ID
(52)	1.2.73	WriteDuplicateType	test_dupType	ID

This benchmark is also quite popular with over 3k stars and 500 forks on the GitHub repository. `jvm-serializers` can realistically simulate a large number of end-user invocations, and as such is a good fit for our experiment. While the benchmark `jvm-serializers` runs, it stimulates Fastjson by performing different kinds of invocations against the library and by referencing different usage scenarios that the designers of the benchmark found interesting for any JSON processor.

The benchmark has been enabled to host In Vivo testing sessions (more details are available from an online technical report [25]). Also, we configured the benchmark so that it only runs against Fastjson, and not the other supported JSON processors. Such configurations do not alter the logic of the original benchmark. This version of the `jvm-serializers` is part of the replication package presented at the end of this article in the section titled: "Replication package and data availability".

5. Methodology

We aim to investigate how the non-deterministic behaviour of the tests executed in the development environment is reflected in the production environment. The experiment foresees two phases: (1) **In House**, and (2) **In Vivo**. In the following, we introduce the purpose of each phase (Section 5.1) and the transformation steps needed to enable In Vivo testing (Section 5.2).

5.1. Study phases

In House phase. The In House phase aims to assess the manifestation of test flakiness in the development environment. To this end, we re-run all the flaky tests of Fastjson without modifying their code. For each class affected by some flakiness according to the IDoFT repository, we launch all the test methods in the class. This step is repeated several times; at each iteration, we guarantee that each launch (i.e., a single run of all the tests in a class) is isolated from the others. To preserve the test behaviour originally implemented by the Fastjson developers, we do not put in place any additional isolation measures between tests belonging to the same class, or between different test classes executed in the same run. Notably, some of the most popular frameworks supporting automation in tests execution (e.g., the Maven Surefire

Plugin, or JUnit 4 with its runner classes) advise that there is no guarantee on the ordering of tests executions under their default configurations. In other words, testers must always assume non-determinism about tests executions scheduling; alternatively, they must explicitly declare a specific selection criterion. Nonetheless, this assumption does not necessarily imply that tests ordering will actually change across different runs. In order to mitigate any potential bias in our study, we executed the OD flaky tests (for which flakiness manifestation depends on their execution ordering) with both default (i.e., non-deterministic) and randomised mechanisms for scheduling the tests to be executed. We do not, however, use any particular tool for detecting flakiness (e.g., *NonDex)*, as we aim to observe the natural behaviour of these tests in a real development setting.

In Vivo phase. For In Vivo testing we use a parametric version of each test case. Specifically, each Fastjson test is enabled to be launched from the operational context reached by a Java application that it is making use of the Fastjson library.

Each test has been revised so to achieve a parametric version of the same test logic originally implemented by the Fastjson developers. Reference examples about test case parametrisation are discussed below, while the mitigation of the risks about potential modifications in the test logic or the injection of biases is discussed in Section 8.

5.2. Enabling In Vivo testing

In the following, we explain the preparation work for our study. For the sake of length, we only provide here a brief summary of the needed actions, whereas a detailed description including also some code examples is included online [25].

Selection of methods to be tested. As described in Section 2.3, we rely on the Groucho framework for In Vivo testing. Groucho requires test engineers to indicate those methods whose invocation will trigger an In Vivo testing session, and for each of them, to refer to an In Vivo test case to be run when the In Vivo session starts.

```
1  public boolean invivoIssue1480(Context c) {
2          ...
3          byte[] contextData = (byte[]) c.
                 getOtherReferencesInContext().get(0);
4          JSONObject obj = InputGenerator.
                configureAlibabaJSONObject(contextData);
5          ...
6          RuntimeEnvironmentShield shield = new
                RuntimeEnvironmentShield();
7          try {
8                  shield.applyCheckpoint(obj);
9                  Issue1480 unitTest = new Issue1480();
10                 unitTest.configure(obj);
11                 unitTest.test_for_issue();
12                 System.out.println("Success");
13         }catch(Throwable t){
14                 System.out.println(t.getMessage());
15                 System.out.println("Failure");
16         } finally {
17                 shield.applyRollback(obj);
18         }
19     ...
20 }
```

Listing 1: In Vivo Test Method

In our context, we played the role of test engineers and established the relation between the `jvm-serializers` methods enabling In Vivo testing sessions and the In Vivo tests hosting the execution of the flaky tests from Fastjson. The applications of the annotation instances `@TestableInVivo` are not further detailed in this paper; but briefly, we analysed each flaky test in our dataset and we established this relationship based on the information required by the flaky test and on the kind of information that would be on top of the run-time stack just before the invocation of the enabling method in the benchmark.

Listing 1 shows an example of In Vivo test that retrieves the operative context on the run-time stack of the benchmark (line 3). If needed, the data are used to set-up the parameters of the flaky test from Fastjson (line 4). Even though Groucho already offers in-memory isolation between the operational context from the benchmark and the In Vivo testing session, the In Vivo test can implement further isolation layering; for example, to isolate the execution of different flaky tests within the same In Vivo session (i.e. line 8, and line 17). Once completed its set-up, the flaky test is launched (lines 10–11). At the end of the In Vivo testing session, Groucho restores the run-time stack.

Fetching the input configurations. Each Fastjson flaky test is executed starting from a specific state/configuration reached by the `jvm-serializers` benchmark at run-time. Thus, each test must be supplied with information retrieved from the current operational context. Although generally the in-memory objects accessible from the run-time stack by the In Vivo testing session can be directly used to set-up the parameters of the flaky test, this is not always the case. Sometimes they could not exactly match the parameter types expected by the considered flaky test. For example, the flaky test `Issue1480` in Listing 1 expects to be configured with a `JSONObject` instance. However, if the In Vivo testing session is triggered during a deserialization operation, the run-time stack will hold contextual data in the form of a byte array (see line 3 of Listing 1), which have to be converted in a `JSONObject` instance before being used to configure the considered flaky test. It is under the responsibility of test engineers to grant that the conversion process of the contextual data as accessible from the run-time stack is correct. Specifically in the case of `Issue1480`, we carefully verified that the instantiating procedure in `JSON.parseObject` was correct for its purpose and is was not affected by any known issue in Fastjson.

To handle such situations we defined a dedicated `InputGenerator` class. More details are available online [25].

Parametric test cases derivation. As anticipated, we converted the flaky test methods in configurable and parametric versions but without altering their test procedures as originally implemented by the Fastjson developers. Although costly in terms of effort for both the development and the validation of compliance with the original test code, this process is fundamental to achieve the true potential of In Vivo testing.

Listing 2 shows the parametric version of a test class, whereas Listing 3 reports the In House version of the same test class, as conceived by the Fastjson developers. The original test code must undergo different modifications. Most importantly, the instantiation of the objects that are the subject of the tests must be decoupled from the implementation of the testing procedures. Our suggestion is to include in the test class a dedicated attribute, if not already present (see line 6 in Listing 2). In addition to the default constructor (line 9), the test class should also offer a configuration method (line 16) that can be used to set-up such an instance when needed. Any declaration or operation that might pollute the instance under test must be carefully considered and removed; in fact, the initial value of the instance under test should be exclusively determined during the In Vivo testing session from the contextual data as accessible from the actual run-time stack (line 22). The last step requires a thorough analysis and redesign of the test statements and of the underlying oracle mechanism. Indeed, the body of the test code may need to be formulated based on any arbitrary state admissible for the instance under test, and not just based a specific configuration (i.e., a fresh instantiation of the subject of the test). Similarly, the expected test outcome, which is usually expressed as a simple hard-coded value, must be dynamically computed based on the input values observed during the In Vivo testing session (line 25).

```
1  package it.cnr.iasi.saks.groucho.lab.instrument.test.
       experiments.fastjson.test.V1273;
2  ...
3  public class Issue1780_Module {

    //Instance under test
     org.json.JSONObject req;

    //Default configuration
     public Issue1780_Module(){
        this.req = new org.json.JSONObject();
        req.put("id", 1111);
        req.put("name", "name11");
     }

    //In vivo configuration
     public void configure(org.json.JSONObject object){
        this.req = object;
     }

     @Test
    public void test_for_issue() {
        org.json.JSONObject req = this.req;
        SerializeConfig config = new SerializeConfig();
        config.register(new myModule());
        String expected = buildExpected(req);
        Assert.assertEquals(expected, JSON.toJSONString(req,
              config));
    }

    public String buildExpected(JSONObject input) {
        ...
    }
    public class myModule implements Module {
        ...
    }
}
```

Listing 2: Example of parametric test

```
package com.alibaba.json.bvt.issue_1700;
...
public class Issue1780_Module extends TestCase {
        public void test_for_issue() {
                org.json.JSONObject req = new org.json.JSONObject
                        ();

                SerializeConfig config = new SerializeConfig();
                config.register(new myModule());
                req.put("id", 1111);
                req.put("name", "name11");
                Assert.assertEquals("{\"name\":\"name11\",\"id
                        \":1111}",
                JSON.toJSONString(req, config));
        }

        public class myModule implements Module {
        ...
        }
}
```

Listing 3: Example of test designed for In House execution

According to the general suggestions discussed above, the specific process we followed to adapt each target test class can be summarised as follows:

1. **Analyse test class:** We analysed the target test case(s) to identify the areas of code that require manual intervention.
2. **Isolate test set-up:** We included the instance under test as an attribute of the test class. We defined a default constructor for the test class, if not already present. Then, we separated the configuration part of the test from its actual execution. To this end, we:
    - moved the default configuration performed by the target test case(s) into the default constructor;
    - cleaned the test code from unwanted set-up and configuration operations;
    - replaced any reference to the hard-coded instance under test with the attribute of the test class.
3. **Include configuration mechanism:** We defined a custom `configure()` method, which is responsible for initialising the instance under test with the value retrieved from the execution context;
4. **Redesign test oracle:** we updated the logic for computing the expected output;
5. **Parametric test assertions:** we parameterised all the test assertions accordingly.

At the end of the process, we modified 40 Java classes: each one including at least one of the 52 test methods studied. Overall these classes counted around 1200 code lines (comments, headers, and `import` or `package` statements excluded). After the parametrisation process the resulting set of classes counted in total almost 2900 code lines. Even though in most of the cases the implementations of the changes were easy and quite systematic to apply (e.g. adding the method `configure()` and moving few setting statements in the constructor of the class), certainly the structural complexity of the classes increased. In addition, few classes reported major modifications as for the inclusion of (at least) an additional ad-hoc private method realising the logic for computing the expected outcome of the parametric tests.

Note that, in some cases, it is not possible to fully apply Steps 1–5 (we further elaborate on this aspect in Section 7.2).

Clearly, the new test implementation should be functionally equivalent to the one originally conceived by the Fastjson developers. To mitigate the risk of affecting the test logic, each modification applied to the test has undergone a validation phase. Besides reducing the risk of distorted test logic, this validation phase helps to ensure that undesired failing/passing tests are not caused by any bug introduced during our adaptation of the code. Specifically we structured the validation phase around a simplified peer code review process: one of the authors (i.e. the analyst) is responsible to analyse the original test and to promote a first sketch of solution for its parametric version; the others play the role of code reviewers supporting the analyst in assessing if the code of the two test versions realise the same logic.

Fig. 2 illustrates the steps of the validation phase. First, the analyst (i.e., one of the authors), that already proposed an initial implementation, performs several runs (in the order of few tens) of the parametric test with its default input (i.e, the same input used during In House testing). The results are then compared to those of the original test run In House. Any discrepancy among the outcomes is considered as a potential distortion of the test logic during the adaptation process.[9] In such a case, the analyst revises the parametric test code until the results are conforming, or until she reaches a maximum effort value, which it has been prudentially set up to one week. Such a massive period has been planned because none of the authors is involved in the development of Fastjson, thus this week period also considered the time possibly required to acquire proper understanding on the SUT code. Nevertheless, in practice, all the considered tests were parameterised in few working hours.

When a parameterised test appears to conform to its original version, the analyst and at least one reviewer together revise the proposed implementation. If they agree the two implementations conform, the analyst performs an additional check to increase our confidence in the correctness of the adaptation step. Indeed, the tests we considered are already known to be flaky (i.e., by the IDoFT repository). In some cases, the Fastjson developers addressed the reported flakiness by including patched versions of such flaky tests in later versions of the project. When available, the analyst used these refinements to estimate if the parametric test implementations were sufficiently close to their original versions. In particular, with reference to a given flaky test T whose

[9] In our specific setting we are aware that flakiness may have an impact on the test outcome, and during the adaptation phase we critically revised any manifested discrepancy. Each time we also checked if the documented source of that specific flakiness might have influenced such discrepancy.

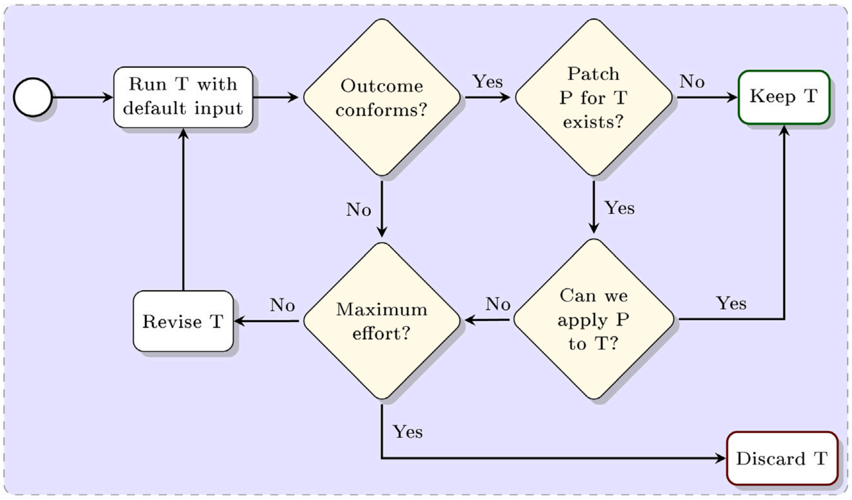

Fig. 2. Validation process for parametric In Vivo tests.

parametric version shows outcomes conforming to T (see Fig. 2), the analyst first searches for any accepted patch P for T on the Fastjson's GitHub code-base. If there are no patches available, no further checks are applied and we keep the current parametric version of T in our dataset. Otherwise the analyst checks if P can be applied to the parametric version of T: if not, the current parametric test version is too far from the original implementation. In this case the analyst (assisted by a reviewer) revises the current parametric code until the patch becomes applicable (or until we reach the maximum effort value). After several iterations we obtain a set of parametric tests to be run In Vivo.

6. Experiment set-up

Section 4.2 already introduced the usage of an open-source benchmark for Fastjson (i.e. `jvm-serializers`) as the reference application we used in order to simulate significant usages of the library. Specifically, the benchmark includes five JSON Objects of different size and complexity to be serialised and deserialised multiple times.

In planning the study, we aimed at guaranteeing fairness of execution across the In House, and the In Vivo phases. Indeed, we kept the number of repetitions of each test equal in both settings, although this was not immediate as we could not exactly decide the number of times an In Vivo test is repeated. As reported in Section 5.2, an In Vivo testing session is triggered upon the invocation of some configured methods of the benchmark. The selection of such methods though depends on the information available on top of the run-time stack in the very moment of their invocation. Thus, the number of times the In Vivo testing sessions will be activated depends on the actual frequency such enabler methods are actually called while the benchmark is running.

To guarantee fairness, for each flaky test we first executed the In Vivo phase and calculated the times its enabler method were invoked, then we matched this result with the number of In House repetitions (which we could control). For example, the In Vivo testing session for test 45 in Table 2 has been associated with a method in the benchmark that was frequently invoked; thus test 45 has been run more than 54,000 times In Vivo. Consequently, the same number of repetitions has been granted for its execution In House.

Nevertheless, in most of the cases, we are able to associate the activation of In Vivo testing sessions with methods that are invoked just once during a serialisation or deserialisation in the benchmark. This set-up allow to easily determine the number of repetitions during the In Vivo, and the In House runs. Specifically, for each of the five JSON Objects it includes, the benchmark triggers 2000 iterations of both the serialisation and the deserialisation procedures. Therefore, this set-up allows us to simulate sufficient realistic end-user invocations for all the phases (i.e., 10,000 repetitions in the general case).

As anticipated in Section 4, our initial test pool includes 52 flaky methods (see Table 2) of the Fastjson project collected from the IDoFT database.

During the In House execution phase we re-run the initial pool of 52 flaky tests on our local machine. As introduced in Section 5, each flaky test is run together with all the methods present in the corresponding test class.

In addition, each Order Dependent-Victim (OD-Vic) test was re-run together with the test class that contains the relative Polluter, that is, the test that modifies the state on which the Victim depends. For instance, the `MaxBufSizeTest` class was run together with the non-flaky `SerializeWriterTest_14` class, as the latter performs operations that can impact the outcome of test 46 (see Table 2).

Unfortunately, some Order Dependent tests present in the dataset have not been fixed, and we do not have enough information to track down the related Polluter test(s). In particular, the test classes `Issue_717` and `JSONFieldTest5` (i.e. tests 15, and 46 in Table 2) were executed in isolation throughout the experiment.

In order to replicate an unbiased execution setting we never forced any specific test ordering; for this reason, we referred to the same configurations in place at the Fastjson building system. This way, we performed seven separate In House testing campaigns: one for each Fastjson release involved in the experiment. The time required for running In House all the considered test classes was 222,37 h. The computational effort was significant as for each test class (and related Polluters), the isolation has been guaranteed by running 10,000 independent build processes each one exercising just the considered test class (and related polluters) on a fresh JVM instance.

All the tests were launched on a dedicated host running Ubuntu 20.04.5 LTS and equipped with a quad-core processor, and 6 Giga-byte of memory.

In order to replicate the set-up described above, please refer to the section titled: "Replication package and data availability" at the end of this article.

7. Discussion of results

In this section we discuss the results of our study in relation to the experimental phases described in Section 5: In House (Section 7.1), and In Vivo (Section 7.2); we then answer in detail the initial research questions: RQ1 (Section 7.3), RQ2 (Section 7.4) and RQ3 (Section 7.5). As introduced in Section 4, our initial test pool includes 52 flaky methods (see Table 2) of the Fastjson project collected from the IDoFT

Table 3
Experimental results.

Test ID	Test category	Case ID	Test ID	Test category	Case ID
(1)	ID	CP-2-CP	(27)	OD-Brit	CF-2-CF
(2)	ID	CP-2-CF	(28)	OD-Brit	CF (In-House)
(3)	ID	CP-2-CF	(29)	ID	CP-2-CP
(4)	ID	CP-2-CP	(30)	ID	CP-2-CP
(5)	ID	CP (In-House)	(31)	ID	CP-2-CP
(6)	ID	IF-2-CP	(32)	OD-Brit	CF-2-CF
(7)	OD-Brit	CF-2-CF	(33)	ID	CP-2-CP
(8)	OD-Brit	CF-2-CF	(34)	ID	CP-2-CP
(9)	OD-Brit	CF-2-CF	(35)	ID	CP-2-CP
(10)	OD-Brit	CF-2-CF	(36)	ID	CP-2-CP
(11)	OD-Brit	CF-2-IF	(37)	ID	CP (In-House)
(12)	OD-Brit	CF-2-CF	(38)	ID	CP (In-House)
(13)	ID	CP (In-House)	(39)	OD	CP-2-CP
(14)	ID	CP-2-CP	(40)	ID	CP-2-CP
(15)	OD	CP (In-House)	(41)	ID	CP-2-CP
(16)	ID	CP-2-CP	(42)	ID	CP-2-IF
(17)	ID	CP-2-IF	(43)	ID	CP-2-CF
(18)	OD-Brit	CF-2-CF	(44)	ID	CP-2-CF
(19)	OD-Brit	CF-2-CF	(45)	OD-Vic	IF-2-IF
(20)	ID	CP-2-CP	(46)	OD-Vic	CP-2-CP
(21)	ID	CP-2-CP	(47)	ID	CP-2-CF
(22)	ID	CP (In-House)	(48)	ID	CP-2-CP
(23)	ID	CP-2-CP	(49)	ID	CF-2-IF
(24)	ID	CP-2-CP	(50)	ID	CP (In-House)
(25)	ID	CP-2-CP	(51)	ID	CP-2-CP
(26)	ID	CP-2-CP	(52)	ID	CP-2-CP

database. Table 3 presents the experimental results; each flaky test method is associated with a category, and a Case ID according to the notations shown in Table 1. We recall that the Case ID describes how the test behaved throughout the In House and In Vivo experiments. For example, CP-2-IF denotes a method that Consistently Passed In House, but exhibited an Intermittently Failing behaviour In Vivo. Instead, a notation like CP (In-House) indicates that it was only possible to retrieve the execution results for the In House phase. (Additional details about the tests excluded from the In Vivo execution are provided in Section 7.2.)

7.1. In House phase results

During the In House phase we re-run the initial pool of 52 flaky tests on our local machine.

Interestingly, most tests (~73%) exhibited a *consistently passing* (CP) behaviour despite being marked as flaky in the IDoFT database. This set includes 35 Implementation Dependent (ID) tests and 3 Order Dependent (OD) tests, 1 of which was further categorised as Victim (OD-Vic). For example, let us consider test (23) in Table 3. As described in the relative GitHub pull request,[10] this method checks whether the string representation of a HashMap matches a hard-coded value. However, as explained in the official Java specification,[11] HashMap makes no guarantee about the iterating order of map. Therefore, the instability of the string representation asserted by test (23) constitutes a latent source of flakiness. Our results seem to confirm that ID flakiness requires specialised tools like NonDex to be discovered. Similarly, the consistently passing OD tests in Table 3 never suffered from an improper state set-up or tear-down operation.

A relatively large percentage of In House tests, about 23%, *consistently failed* (CF); this includes all 11 Order Dependent-Brittle (OD-Brit) tests present in the dataset and 1 ID test. This result is quite interesting as each CF test seems to be failing on our local machine only, which suggests a configuration-dependent type of flakiness. This suspicion seems to be confirmed by the flakiness-fixing patches available on GitHub. Indeed, each test requires the initialisation of the Timezone and/or Locale properties to pass. Since this set-up either does not exist or is never scheduled in the appropriate order, each test caused a build failure during each In House run. However, the same test methods would pass on a machine in the right Timezone (and/or Locale). This is an interesting scenario as the flakiness might remain hidden for a long time due to its specific activation conditions.

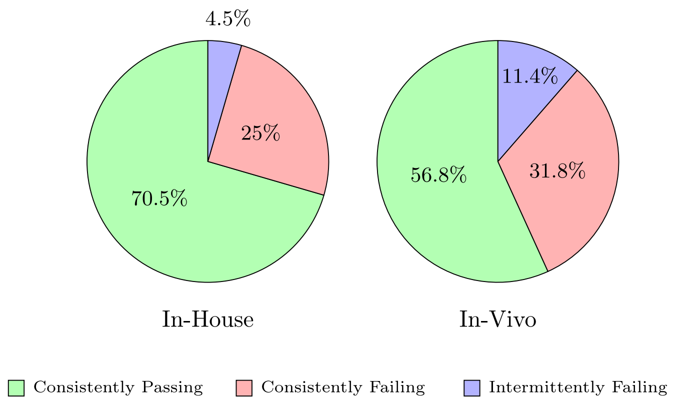

Fig. 3. Percentage of CP, CF, and IF tests observed during each experimental phase (calculated considering the 44 adapted tests).

As can be seen in Table 3 at Col. 3, only test (6) and test (45) exhibited an *intermittently failing* (IF) behaviour when run on our machine. According to IDoFT, the former method contains an implementation dependence, but there are no developer-confirmed patches that can help us to pinpoint the flakiness source. On the other hand, test (45) was marked as OD-Victim. In this case, the reordering of the test methods introduced state pollutions that resulted in intermittent failures.

Considering the high number of re-runs, we expected to observe more intermittent methods at the end of this step. Whether this is due to some environmental settings or configuration is something we aim to investigate during the In Vivo phase.

7.2. In Vivo phase results

In this section, we present the results of the parametric tests run In Vivo. As shown in Table 3, we were able to adapt and re-run 44 out of the 52 flaky methods in the initial test pool. 8 tests were excluded from this phase because it was not possible to apply the adaptation Steps 1–5 described in Section 5.2, whereas no tests were ruled out due to issues encountered during the validation phase (i.e., the effort threshold was never exceeded). For instance, test (37)[12] in Table 3 addresses the automatic serialisation and deserialisation of classes in a specialisation hierarchy. However, since our data model consists of unrelated JSON objects, it was not possible to exploit the run-time context of the SUT. For such reason, we excluded this test (and other similar cases) from the experiment.

From Fig. 3, the difference between the In Vivo and the In House experiment is evident; the former resulted in the highest percentage of both **Consistently Failing** (~32%) and **Intermittently Failing** (~11%) tests. Indeed, switching to parametric methods allowed us to leverage the huge variability proper of the field environment, which would have been too costly to reproduce In House.

In Table 4 we show the observed test outcomes based on their category. For each Case ID we report the number of methods that were initially categorised as Implementation-Dependent (ID), Order-Dependent

[10] https://github.com/alibaba/fastjson/pull/3025
[11] https://docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html
[12] https://github.com/alibaba/fastjson/blob/1.2.78/src/test/java/com/alibaba/json/bvt/issue_3600/Issue3655.java

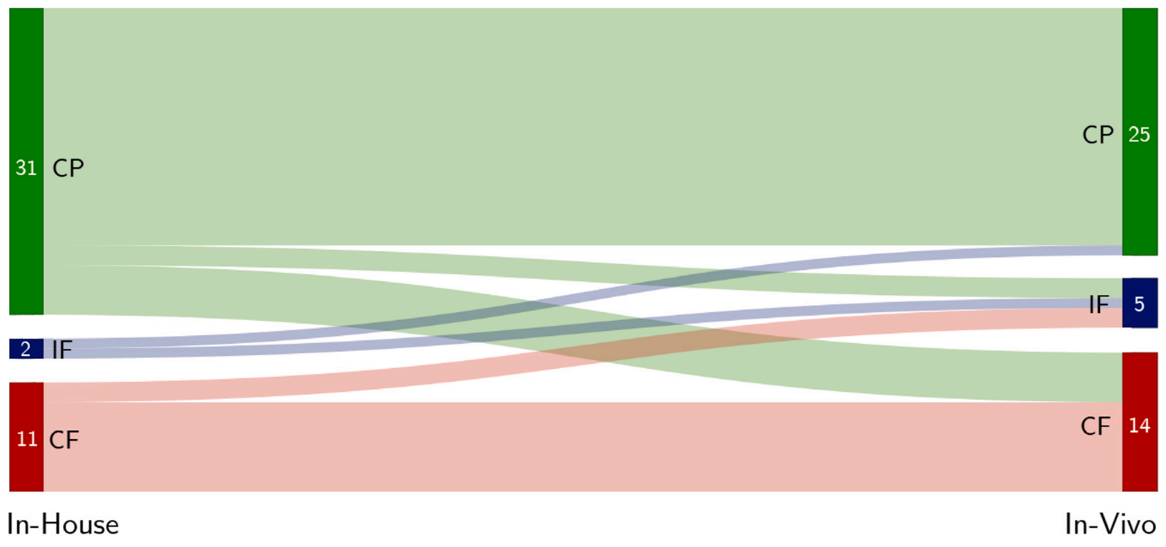

Fig. 4. Behavioural evolution of the tests moved In Vivo.

Table 4
Test methods belonging to each case.

Case ID	ID	OD	OD-Vic	OD-Brit	Tot.
IF-2-IF	0	0	0	0	0
CP-2-IF	2	1	1	0	3
CF-2-IF	1	1	0	1	2
In Vivo IF	3	2	1	1	5
IF-2-CP	1	0	0	0	1
CP-2-CP	22	2	1	0	24
CF-2-CP	0	0	0	0	0
In Vivo CP	23	2	1	0	25
IF-2-CF	0	0	0	0	0
CP-2-CF	5	0	0	0	5
CF-2-CF	0	9	0	9	9
In Vivo CF	5	9	0	9	14

(OD), Order-Dependent Victim (OD-Vic), and Order-Dependent Brittle (OD-Brit). As shown, 25 test methods never failed when moved to the field, against the 31 test methods that passed In House. Whereas, the number of Consistently Failing (CF) and Intermittently Failing (IF) tests both increased by 3 In the following we analyse the behavioural evolution (Section 7.3), the Frequency of Failure (Section 7.4), and the Cause of Failure (Section 7.5) of each In Vivo test to answer the research questions posed in Section 3.

7.3. Answering RQ1

With RQ1 we aim to investigate the behaviour of flaky tests when they are run using inputs from the field. We must therefore focus on the 44 tests we were able to run both In House and In Vivo. To make the analysis more understandable, we illustrate the outcome evolution of each test in Fig. 4. The Sankey diagram helps us to visualise how each method behaved in relation to the cases described in Table 1. Here, the nodes on the left and the nodes on the right represent the possible In House and In Vivo test outcomes, respectively. The nodes are linked by arcs that have a width proportional to the number of observed cases.

In the following we discuss the behaviour of the considered tests, grouping them according to the outcome observed In Vivo: Consistently Passing (CP), Consistently Failing (CF), and Intermittently Failing (IF).

In Vivo tests that consistently pass. Most of the tests that consistently pass In Vivo belong to case CP-2-CP, that is, they produced consistently passing outcomes throughout each In House and each In Vivo test run. Although the number of passing test cases decreased by ~21,8% with respect to the In House experiment, the amount of methods marked as CP-2-CP is still quite surprising. The source of intermittence might have stayed hidden In Vivo for different reasons; the testing scenario may be very specific, and thus, it may not reflect real system usage. In this case, re-running the tests In Vivo does not entail any substantial difference. The consistently passing behaviour might also depend on the original flakiness category of the test. Indeed, 22 CP-2-CP methods were marked as Implementation Dependent. For any such tests, it might be difficult to observe a failure if the underlying implementation does not evolve. As shown in Table 4, case CP-2-CP also includes a single OD-Vic and one uncategorised OD method. These tests always passed In Vivo because they were never executed starting from an improperly set state. This means that those settings that sometimes cause the tests to fail do not correspond to conditions that will occur in production. This also applies to the single IF-2-CP method – test `(6)` – that produced intermittently failing outcomes In House, and turned consistently passing once it was moved to the field. Since any test is stripped of its configuration operations during the adaptation phase, running test `(6)` In Vivo narrowed down the source of flakiness to the specific `set-up` and/or `tear-down` operations performed within its class.

In Vivo tests that consistently fail. Most of the consistently passing tests belong to case CF-2-CF, that is, they already produced consistently failing outcomes during the In House test runs. A severe bug in the SUT or an important problem in the test code is the major cause of a consistently failing test. However, the 9 methods belonging to case CF-2-CF were all categorised as OD-Brit. Because this type of test can only pass if we execute it from a specific state, their behaviour acquires a different meaning In Vivo. The failures are no longer attributable to a bad set-up because the tests rely on the state encountered during real executions. On the other hand, the expected state set-up reflects a very specific scenario that the SUT may never reach at run-time. This makes us question the actual usefulness of the test cases. Meanwhile, all the tests belonging to case CP-2-CF were initially marked as ID by the IDoFT contributors. Their failing behaviour In Vivo might be a symptom of a severe issue resulting from a shallow pre-release testing campaign.

In Vivo tests that intermittently fail. As shown in Table 4, 3 ID and 2 OD methods produced non-deterministic outcomes In Vivo. The tests belonging to case CF-2-IF are not particularly relevant because their failing behaviour is easily reproducible in the development environment. A software developer would deal with these tests as soon as

Table 5
Tests whose was impacted by In Vivo execution.

ID	Cat.	In House		In Vivo		Case ID
		FoF	Out.	FoF	Out.	
(6)	ID	0,85	IF	0	CP	IF-2-CP
(11)	OD-Brit	100	CF	54,2	IF	CF-2-IF
(17)	ID	0	CP	20	IF	CP-2-IF
(41)	ID	0	CP	89,92	IF	CP-2-IF
(45)	OD-Vic	67	IF	72,58	IF	CP-2-IF
(49)	ID	100	CF	98,72	IF	CF-2-IF

Table 6
Tests belonging to Case CP-2-IF and CP-2-CF.

ID	Cat.	In House	In Vivo	CoF	Case ID
(2)	ID	CP	CF	Too specific scenario	CP-2-CF
(3)	ID	CP	CF	Implementation dependence	CP-2-CF
(17)	ID	CP	IF	Implementation dependence	CP-2-IF
(41)	ID	CP	IF	Bug in the CUT	CP-2-IF
(43)	ID	CP	CF	Implementation dependence	CP-2-CF
(44)	ID	CP	CF	Implementation dependence	CP-2-CF
(45)	OD-Vic	IF	IF	Order dependence	IF-2-IF
(47)	ID	CP	CF	Implementation dependence	CP-2-CF

possible, either by fixing the problem or by discarding the method altogether. On the other hand, the two tests belonging to case CP-2-IF would be perceived as reliable In House. Nonetheless, they start producing inconsistent results as soon as they are moved to the field. The intermittent nature of the observed failures should be further investigated because it might signal a hard-to-detect bug in the SUT. We investigate this core aspect while answering RQ3. Lastly, we observed one IF-2-IF test method that produced inconsistent outcomes both In House and In Vivo. In Section 7.4 we further investigate the frequency of failure of the considered test methods to understand if and how it is impacted by the production environment.

Answer to RQ1: The execution of flaky tests by simulating inputs from the field can concretely affect their behaviour; the experiment on the simulation of In Vivo testing resulted in the highest percentage of Consistently Failing (∼32%) and Intermittently Failing (∼11%) tests. Specifically, ∼16% of all test methods always passed In House, but consistently or intermittently failed In Vivo, suggesting an issue that can only be triggered by specific input or environmental conditions.

7.4. Answering RQ2

With RQ2 we aim to investigate whether running a test In Vivo has some impact on the perception of flakiness. To this end, we analyse those test methods that exhibited some intermittence (i.e., In Vivo, In House, or both) and we compare their resulting **Frequency of Failure** (FoF). Recall that in Section 3 we defined the FoF as the percentage of observed failures of a test over the total number of re-runs. The considered tests are listed in Table 5, together with their original flakiness category (Cat.) and Case ID. For each method, the table also shows the FoF and the test outcome observed both during the In House (Col. 3 and Col. 4) and the In Vivo (Col. 5 and Col. 6) phases. In formulating RQ2 we were expecting to be able to observe some different FoF values, but for most test cases we studied, when their behaviour changed, this was from consistent to flaky, or the vice versa. Nevertheless, for 6 test methods that we marked as intermittent, each one of them displayed either an increase – tests (17), (41), and (45) – or a decrease – tests (6), (11), and (49) – in FoF. The remaining 38 tests were non-intermittent throughout the experiments, that is, they produced deterministic outcomes both In House and In Vivo (although they might have switched from a CP to a CF state, or vice versa).

Although we observed few instances of intermittent tests, the variation in the FoF between the In House and the In Vivo phase is often significant. Indeed, most methods exhibit a deterministic behaviour In House and only become intermittent in the field. These state changes suggest that a field testing approach can help to (re)create those conditions necessary for the flakiness to emerge. Additionally, there is one test method (45) that stayed non-deterministic during both runs. However, its FoF increased by ∼8,3% during the In Vivo experiment. This increase is particularly interesting considering that test (45), marked as OD-Vic, required us to apply randomised order scheduling (As explained in Section 5.1) to observe intermittent failures In House. Instead, when we used the default, non-deterministic scheduling, we observed that the tested instance was always exercised from a clean state, resulting in consistently passing outcomes (please note that we did not include the latter in Table 5 for the sake of clarity). Even so, the FoF still increased In Vivo. The reason is simple: the instance under test traverses many – possibly polluted – states in production. As explained in our answer to RQ2, a higher In Vivo FoF should always be taken seriously by the test engineers, especially if it is not constant.

Notably, even a test with a decreased intermittence level can yield helpful information. For instance, the behavioural evolution of test (6) reveals a connection between the flakiness and the specific set-up and/or tear-down operations performed by the original test class.

To conclude our discussion, we focus on those intermittent tests that consistently failed In House. Running such methods in production might seem counter-intuitive, but we argue it might be useful in some situations. For instance, the In House outcome of the CF-2-IF tests reported in Table 5 depends on the testware set-up. In particular, both test (11) and test (49) must run in the "Asia/Shanghai" time zone, but the respective test classes lack this fundamental set-up operation. These tests regularly failed on our machine in the Rome time zone, but a developer in Shanghai would see these same tests as CP-2-IF. In the latter case, running these tests in production – in different time zones – can help to discover the failure pattern and avoid confusion.

Answer to RQ2: From our simulation study, moving a test to the field can concretely impact its *Frequency of Failures* (FoF) and possibly help to identify the relative cause of failure.

7.5. Answering RQ3

The aim of RQ3 is to determine whether some flaky tests can be seen as useful indicators of **hard-to-detect failures** [5], rather than as a nuisance for the test engineers. Indeed, field testing was introduced for triggering types of failures that are difficult to reproduce In House. Thus, we hypothesised that test intermittence observed under specific conditions can be a symptom of a deeper issue in the code under test.

To answer this question, we investigate the behaviour of some relevant test methods in relation to their **Cause of Failure (CoF)**, and also try to investigate if the latter has anything to do with the specific test category or is entirely unrelated. Therefore, we omit from the following analysis all those methods that constantly failed (CF) In House or constantly passed (CP) In Vivo. For each test ID, Table 6 shows its original flakiness category (Cat.), and the outcome observed both during the In House and the In Vivo phases. The evolution of the test outcome is encoded in the respective Case ID as previously described in Table 1. Lastly, the table summarises the Cause of Failure (CoF) observed during the In Vivo phase.

CP-2-IF*: Consistently Passing to Intermittently Failing.* As shown in Table 6, case CP-2-IF includes 2 test methods – (17) and (41) – that start from a CP state In House and become IF in the field.

Test (17) was originally marked as flaky by the contributors of the IDoFT, because it includes an implementation dependence. As it can be seen from Listing 4, the original test assertion on line 6 compares a hard-coded string against the textual representation of a JSON object implemented by a HashMap. As briefly explained in Section 7.1, the latter does not guarantee any ordering of elements. Despite this latent source of flakiness, the test started producing inconsistent outcomes only once we fed it a more complex JSON object retrieved from the benchmark's data model.

Test (41) was also categorised as ID by the NonDex tool. However, the intermittent failures observed In Vivo are not related to the incorrect ordering of the asserted elements. Unlike test (17), the CoF can be traced back to an actual bug in the SUT. In fact, the method `JSONPath.reserveToArray()` invoked by test (41) (line 5 of Listing 5) seems to return an incorrectly built string when it takes in input complex objects. We provide a detailed description of the bug in the corresponding issue[13] raised on GitHub. Thus, running the test In Vivo helped us to identify a problem in the application code that remained undetected due to an inappropriate In House testing campaign.

```
1   public class Issue1177_2 extends TestCase {
2       public void test_for_issue() throws Exception {
3           String text = "{\"a\":{\"x\":\"y\"},\"b\":{\"x\":\"y\"}}";
4           Map<String, Model> jsonObject = JSONObject.parseObject(
                text, new TypeReference<Map<String, Model>>(){});
5           ...
6            assertEquals("{\"a\":{\"x\":\"y2\"},\"b\":{\"x\":\"y2\"}}",
                 JSON.toJSONString(jsonObject));
7       }
8       ...
9   }
```

Listing 4: Original Issue1177_2 test class

```
1   public class JSONPath_reverse_test extends TestCase {
2       public void test_reserve() throws Exception {
3           JSONObject object = JSON.parseObject("{\"id\":1001,\"name
                \":\"ljw\",\"age\":50}");
4
5           assertEquals("[1001,\"ljw\"]", JSONPath.reserveToArray(
                object, "id", "name").toString());
6           assertEquals("[\"ljw\",1001]", JSONPath.reserveToArray(
                object, "name", "id").toString());
7           assertEquals("[\"ljw\",[\"ljw\",1001,50]]", JSONPath.
                reserveToArray(object, "name", "*").toString());
8       }
9       ...
10  }
```

Listing 5: Original JSONPath_reverse_test test class

CP-2-CF*: Consistently Passing to Consistently Failing.* This group includes 5 tests that were marked as CP In House, but consistently failed In Vivo, often because of an implementation dependence. In particular, tests (3), (44), (45) and (47) rely on a data structure, such as `HashSet`, which makes no guarantees as to the iteration order of its elements. This issue was not exposed In House due to the simplicity of the data model used for testing. Test (2) on the other hand, requires some TimeZone set-up operations to pass In House. When moved to the field, the test is stripped of its set-up method, and thus it relies on the actual TimeZone of the SUT. In this case, the In House testing is very specific and does not reflect the actual usage scenarios.

IF-2-IF*: Intermittently Failing to Intermittently Failing.* This group only includes test (45), which is an Order-Dependent Victim. As it can be seen in Listing 6, `test_0` performs operations on a serialiser instance (line 3) which is always assumed to be in a clean state. However, this is not always the case; if `test_3` (line 13) runs before `test_0`, it will modify the serialiser state on which the victim depends. This unhandled state-pollution will cause `test_0` to fail whenever the test scheduling matches the abovementioned order. When conducting the In House experiment, we intentionally introduced a random test order scheduling for the class, which led to the observation of several intermittent failures. However, by moving `test_0` In Vivo we were able to observe a higher FoF and pinpoint the issue more effectively. This is because the method now relies on a serialiser instance retrieved from the execution context of the SUT. Upon testing, the serialiser might or might not have a clean state, causing the OD test to intermittently fail.

```
1   public class JSONSerializerTest2 extends TestCase {
2       public void test_0() throws Exception {
3           JSONSerializer serializer = new JSONSerializer();
4           int size = JSONSerializerMapTest.size(serializer.
                getMapping());
5           serializer.config(SerializerFeature.
                WriteEnumUsingToString, false);
6           serializer.config(SerializerFeature.WriteEnumUsingName,
                 false);
7           serializer.write(Type.A);
8
9           Assert.assertTrue(size < JSONSerializerMapTest.size(
                serializer.getMapping()));
10          Assert.assertEquals(Integer.toString(Type.A.ordinal()),
                 serializer.getWriter().toString());
11      }
12      ...
13       public void test_3() throws Exception {
14          ...
15       }
16  }
```

Listing 6: Original JSONSerializerTest2 test class

Answer to RQ3: Running the flaky tests in a simulation of the field allowed us to spot several issues that remained hidden In House, or to facilitate their manifestation. The observed hard-to-detect bugs escaped pre-release testing due to the numerous combinations of inputs and states that cannot be addressed In House (combinatorial explosion). In most cases, such flakiness was rooted in the design of the test method itself, however, the intermittent behaviour of a test method helped us to identify an actual bug of FastJson. Thus, some flaky tests might represent a useful hint for a test engineer, and should not be light-heartedly discarded.

7.6. Practical implications for developers and researchers

One of the most attractive perspectives posed by In Vivo testing is the exploration of uncommon SUT states by leveraging execution conditions from its actual usage. Differently from other controlled experiments in the field (e.g. canary releases, dark launches, A/B testing) that support decisions based on feedback from the users [26], In Vivo testing is a proactive technique that aims at detecting failures before they manifest to actual users. However, this capability also has associated trade-offs. In several parts of this work, and with more emphasis in Section 5.2, we remarked that shifting the analysis of flaky tests In Vivo leads to several implications that flakiness analysts (e.g., researchers, testers, developers) have to cope with. For example, those implications could be roughly expressed in terms of economic costs or human efforts.

A first implication concerns the effort of setting up an environment that is able to run flaky test In Vivo. Clearly, flakiness analysts would

[13] https://github.com/alibaba/fastjson/issues/3936

refer to some available framework for In Vivo testing avoiding (if possible) to invest effort in developing any custom solution. However, even though an applicable In Vivo testing framework exists, its technical requirements could require dedicated activities for an operative integration with the specific technological stack or the application scenario. For example, while planning an In Vivo testing strategy on real scenarios (i.e., in production and not in research studies like in this work) it is important to establish an effective compensation schema through which the possible side effects from In Vivo activities can be accepted or avoided. Even though the In Vivo testing framework could offer effective isolation mechanisms, experimenting with flakiness in a running production environment must carefully undertake such a need in all its aspects: technical, or organisational.

A different class of implications concerns the development process of both the SUT and its test cases. In Vivo testing makes sense only if tests leverage parameters set from the actual execution context to any of its admissible values. On the one hand, if the SUT's development process already encourages the development of parametric tests, these tests are ready to be used also In Vivo likely with none or few modifications for the integration with the specific In Vivo testing framework. On the other hand, any analyst that is willing to study flakiness In Vivo on projects strongly built around nonparametric tests has to cope with a test case conversion process like we did in this work. We agree that this is a non trivial barrier to the systematic investigation of flakiness In Vivo, nevertheless we also acknowledge that the development of parametric tests is a strongly recommended practice in many realistic settings [27]. As from our specific experience with Groucho as In Vivo testing framework, we found that part of the modifications we made during the parametrisation process was quite systematic to apply. Thus, a possible alternative to the manual test case conversion is to invest some effort in structuring a semi-automated process that at least refactors the test by adding the method `configure()` and properly configuring the constructors of the test class.

A specific class of implications mainly refers to researchers studying potential relations between flakiness and observable field failures. Differently from the previous cases, often in this context researchers have no control over the development process of the selected SUT: they must use the resources the project offers as they are. In this sense, if the subject of their investigation only uses nonparametric tests, thus researchers have to invest their effort for transforming the tests into a parametric format by following the Steps 1–5 we reported in Section 5.2. In addition, it is uncommon that researchers have access to any instance of the SUT deployed in some production environment. At the same time, it is also uncommon they could rely on operational profiles or other sources of information about the actual usage of the considered subject. Probably this is the most challenging aspect for researchers. As discussed in Section 4, in this work we had the possibility to identify an external toolkit for stimulating the SUT with those interactions (considered) the most meaningful. If no external benchmark is available for the considered SUT, then researchers will have to implement by themselves an infrastructure that can replicate the In Vivo scenario. For example in this work, in order to extend the experimentation to Biojava (i.e., one of the other two projects from Section 4 complying with C1 and C2 but not with C3), we should have implemented a workload generator either starting from scratch, or tracing custom algorithms only referred by scientific publications and integrating them as a benchmark. Not only this activity requires sufficient knowledge of the domain that Biojava targets and none of the authors currently has experience in bio-informatics, but also it exposes the work to bias concerning the In Vivo execution settings. Similar considerations hold for wildfly for which we should have defined a production-like environment where an instance of the application server is tested In Vivo while it hosts several running applications. Thus in conclusion, on the one hand, studying flakiness In Vivo when the SUT has no parametric tests is doable (e.g., Steps 1–5) even if it in general requires some effort (few working hours in this work as from Section 5.2); on the other hand, when the production environment of the SUT or its operational profiles are available, then researchers have to carefully plan activities that bring sufficient confidence on the actual emulation of the In Vivo settings.

8. Threats to validity

In the following, we present the threats that may affect the validity of the conclusions presented in this work.

8.1. Threats to construct validity

Threats to construct concern the assumptions or decisions made during the definition or the set-up of the experiment that may potentially impact the final results. We identify the following threats to construct validity:

Choice of the Case-study: Though the case study is a real-world project which is also part of an open dataset referred worldwide, its choice clearly may have some impact. We tried to reduce this threat through the formulation of neutral selection criteria to be applied over the whole IDoFT dataset, although then we had to manually analyse the resulting set of projects to verify if the actual technological stack of each project matched our target operative environment. In addition, the selection of the case-study had an impact also on the categories of flaky tests actually covered. While the IDoFT dataset includes several classes of flaky tests, the experiments on Fastjson in this work only concerned flaky tests classified as either ID or OD. In this sense, our study could be considered appropriate only for a subset of the whole spectrum of software test flakiness.

Realistic Production Environment: We simulated end-user sessions by means of an available open benchmark; also this aspect could have affected our experimentation. As explained in Section 4, we mitigated this risk by including criterion C3 among the case-study selection criteria. The `jvm-serializers` benchmark matched our eligibility criteria as it is a project developed independently from us and the Fastjson team. Yet, we are aware that real end-user sessions may provide wider opportunities for instantiating JSON objects.

Development environment: The In House phase had the purpose of observing the outcome of the original Fastjson tests reported as flaky by the IDoFT repository. In other words, we accepted their flakiness and did not look for configurations different from the ones conceived and accepted by the developers' community. Due to their flaky nature, we could have experienced different outcomes for these tests if we pursued an intensive exploration of the configurations either in Fastjson or in its operative building system (e.g., JVM, Maven). About the In Vivo phase, we use Groucho as In Vivo testing framework. Groucho's main responsibility is to suspend the execution of the SUT, isolate its current state and run the tests starting from such a state. Specifically, the activation of the flaky tests from the developers of Fastjson is mediated by the Groucho's callback mechanism (see Section 2.3). Currently we are not aware of issues on the isolation mechanism in Groucho. In this study, for each of the considered flaky tests, the activation of an In Vivo testing session just concerns the execution of one flaky test whose parameters are configured from the current state of the SUT. Even though there is not any explicit study on potential additional flakiness that Groucho may introduce, in this work each flaky test has been studied in separated In Vivo sessions. Thus, their intermittence (either manifested or not) is due to those formal parameters and variables each test implementation uses and that the In Vivo testing session sets with values reached from an observed state of the SUT.

8.2. Threats to internal validity

This category refers to the extent to which the results obtained are a function of the systematic observation/manipulation of the variables in the study.

Test Parametrisation: In order to derive the set of parametric tests to experiment with during In Vivo testing, we re-implemented each target test class shipped with Fastjson. As detailed in Section 5.2, we carefully planned a strategy guiding the conversion procedure (i.e. Steps 1–5). In addition, we elaborated a validation procedure (i.e., see Fig. 2) aiming at achieving enough confidence about the soundness of each flaky test parametrisation. However, the conversion and validation procedures are not proof of full compliance between the original flaky test and its parametric version, and potential mismatches cannot be excluded.

Input Configuration: In the general case we were able to directly set-up the parameters of the flaky test with object instances directly fetched from the current state of the SUT within the In Vivo testing session. However, as also reported in Section 5.2, in a few cases we had to extract such parameters from part of the whole context information available just after the activation of an In Vivo testing session. as also reported in Section 5.2, in a few cases we had to extract such parameters from part of the whole context information available just after the activation of an In Vivo testing session. We did our best to check that the conversion process of the contextual data as accessible from the run-time stack was correct and compliant with the In Vivo testing assumptions (i.e., usage of actual data from the execution context). In this sense, we decided to exclude from our experience all those tests whose required parameters were too far from the available context data. Indeed, they would have required extensive input manipulations which open to the possibility of severe biases. However, also in those cases where the conversion procedure was feasible, we are aware that we may have neglected aspects that could have affected the observed results.

8.3. Threats to external validity

This category refers to the extent to which the results of our study can be generalised. We identify the following threats to external validity:

Generalisation and constructs: Generalisation depends on how we defined the constructs and the variables we wanted to observe. In this work, we aimed to understand if In Vivo testing can contribute to identify any potential relation between test flakiness and *hard-to-detect* failures. Focusing on this main goal, we could have missed potential impacts on other variables not fully addressed by the experience. For example, in our case we were already aware of which tests were flaky, but this could not be always the case. Also, we are aware of the limitations and responsibilities to be faced in order to adopt In Vivo testing approaches, we agree that they are not applicable in all contexts.

Limited subject set: The whole experience considered a set of 52 tests attributed to several categories of flakiness by the IDoFT repository. Furthermore, all these flaky tests belong to only a single project. This set is too small a sample to make any meaningful generalisation. Indeed, possible hidden dependencies among the constructs referred during our experience may not hold in other projects, and so we could observe different results.

9. Related work

This is the first work that leverages field testing to get insights into the behaviour of flaky tests. Related work includes studies about characterising and detecting flaky tests, approaches and tools for field testing, and analyses of field failures.

Flaky tests. In the last decade, research on flaky tests has been attracting growing interest. For a comprehensive overview of the literature, we refer the reader to a recent systematic review conducted by Perry and coauthors [1]. In this review, the authors collect and examine 76 primary studies, which they group into four main research directions: (i) studies analysing the causes of flakiness, and the characteristics of flaky tests; (ii) studies assessing the cost and impact of flaky tests; (iii) approaches for flaky tests detection; and (iv) approaches for mitigating and fixing flakiness.

The scope of our work is orthogonal to these four directions, as with regard to flaky tests we investigate how the test environment (in particular, how moving from in house to the field) impacts on the intermittent test behaviour. In this sense, we discuss below a few works that are more closely related to this scope.

A recent study by Strandberg et al. [13] investigates the root causes of intermittently failing tests in the embedded systems domain. The authors pinpoint nine factors associated with inconsistent test behaviour, and show that determining the root causes of intermittent failures requires more effort with respect to consistent ones. Notably, they discovered that environmental factors play a relevant role in the manifestation of test intermittence, which is also the intuition that drove our research.

Along a similar line of thought, Silva and coauthors [28] show that the detection of flakiness through repeated execution of tests can be improved by adding noise in the test environment, because concurrent events happening during test execution can influence the test outcome. This is the same concept that inspired our work, however differently from their tool SHAKER that modifies the test environment by adding artificial noise, we rely on the concurrent events naturally happening in the field.

Cordy and coauthors [29] have developed FlakiMe, a laboratory-controllable environment in which different scenarios and conditions inducing test flakiness can be simulated. Their strategy clearly takes an opposite direction to ours, as their purpose is that of facilitating researchers to take into account the impact of flaky tests when experimenting with testing related techniques. In particular, they show the use of FlakiMe when studying mutation testing or automated program repair.

More recently, Parry and coauthors [30] proposed a high-level approach that leverages machine learning models for reducing the number of re-run experienced in order to detect flakiness. The ideas of the work result in the CANNIER framework whose impact has been validated in combination with several methods for flaky test detection based of repeated executions. While we acknowledge the importance of all these works that aim to detect test flakiness, the ultimate goal of our research goes in a slightly different direction: leveraging flaky tests to possibly identify faults that are hard-to-detect in the development environment.

Field testing approaches and tools. Field (or In Vivo) testing approaches operate in the production environment to uncover flaws that are missed by In House testing activities. The most prevalent field testing approaches target the operational instance of the SUT (i.e., *online* testing). For example, Netflix engineers purposefully cause interruptions in the production system to increase its resilience. This approach, known as Chaos Engineering [31], helps to ensure that complex systems can withstand turbulent conditions in production.

Murphy et al. [32] designed and implemented *Invite*, the first In Vivo testing framework for Java. *Invite* permits to isolate the test execution context to mitigate the risk of corruption during In Vivo testing activities.

In this work, we used Groucho [10], a fully automated framework for In Vivo testing of Java applications. Unlike previous approaches, Groucho can be applied to the original application code in a transparent manner. That is, there is no need to access and modify the source code of the SUT for enabling In Vivo testing. Indeed, Groucho permits fully automatic instrumentation of the SUT, even if the latter is only available as compiled code.

An exhaustive survey of field testing approaches and tools can be found in the recent systematic study by Bertolino et al. [14].

Field failures. Some recent works address the limitations of In House testing. Specifically, they analyse the complexity of different types of bugs and show how difficult it is for test engineers to reproduce field failures during pre-release testing activities.

Gazzola et al. [5] investigate the nature of field failures that escape pre-release testing and manifest themselves in the field. They analyse the bug reports of five apps from three distinct ecosystems and illustrate why failures from the identified classes must be addressed at runtime. The authors identify four main reasons that make certain faults intrinsically difficult to detect In House: cases impossible to replicate in-house, combinatorial explosion, unknown application, or environment conditions.

Rwemalika et al. [33] offer an extensive investigation on the characteristics of pre-release and post-release bugs over 37 industrial projects from BGL BNP Paribas. Their findings imply that fixing post-release defects is more difficult, requiring developers to change many source code and configuration files.

The empirical investigation by Cotroneo et al. [34] exposes the peculiarities of the bug manifestation process. The authors identify a collection of failure-exposing criteria and provide a fine-grained description of defect manifestation. Interestingly, half of the analysed bugs require at least two workload conditions to surface. Thus, bug detection activities must consider advanced techniques like field testing to capture them.

10. Conclusions and future work

We presented the first simulation study that investigates the behaviour of tests marked as flaky when executed in the field. Our study included two phases, namely **In House**, and **In Vivo**.

The results gathered from the **In House** phase were quite interesting; almost all tests selected for the experiment (∼96%) exhibited a deterministic behaviour despite being marked as flaky by state-of-the-art detection tools. This finding confirms the need for flakiness detection tools, but it also requires them to be as flakiness-agnostic as possible. In fact, most available solutions target a very specific cause of flakiness (e.g. implementation dependence [24], order dependence [23], concurrency [28]) and must therefore be combined to exhaustively identify such tests. Furthermore, a significant percentage of flaky tests (∼25%) consistently failed on our machine due to their reliance on a specific configuration. This confirms that the sources of the intermittent behaviour must be searched both in the execution context and in the configuration settings, which In Vivo testing can thoroughly explore.

Running the pool of flaky tests **In Vivo** was extremely insightful. Firstly, we found an evident difference between the In Vivo and the In House experiment; the former resulted in the highest percentage of both Consistently Failing (∼32% vs ∼25%) and Intermittently Failing (∼11% vs ∼4%) tests. This indicates that running flaky tests using actual inputs from the field can concretely impact their behaviour. In general, most of the Consistently Passing tests observed In Vivo were already passing In House, but we also observed an Intermittently Failing test losing its intermittent behaviour in the field, which can help to narrow down the cause of its flakiness. Most notably, we recorded several cases of tests passing In House and becoming Consistently Failing or Intermittently Failing in the field, which suggests the presence of an issue that can only be triggered by specific input or environmental settings.

We then investigated whether running a test In Vivo has some impact on the perception of flakiness in terms of the test FoF (*Frequency of Failure*). Throughout the experiment, we marked 6 test methods as intermittent, and each one of them displayed a significant increase or a decrease in FoF in the field. Besides, most of them start from a deterministic state In House and only become intermittent once they are moved to the field. This evolution suggests that a field testing approach can help to (re)create those conditions necessary for the flakiness to emerge, and possibly help to identify the relative cause of failure.

Our main goal, however, was to understand whether test intermittence could be seen as a useful indicator of **hard-to-detect failures** rather than a negative event to be eradicated. Our analysis revealed that, in most cases, the root cause of an intermittently failing test is found in the bad design of the test code itself. However, we also found evidence of a field-intrinsic bug that remained hidden during the In House phase due to the known problem of combinatorial explosion. This suggests that some intermittently failing tests could be a valuable hint for a test engineer, and should not be dismissed right away as unreliable methods.

Towards such direction, this study lays the ground to further research about non-determinism of tests that could be due to specific configurations and environment conditions and hence could be useful to dig out the possible presence of hard-to-detect failures. In future work, we aim at conducting other empirical studies to explore other possible relations between the two worlds of In House and In Vivo testing. In particular, we aim at exploring the possible connection of flakiness with the results from previous studies about Mandelbugs [35, 36]. Moreover, while in this paper we considered a whole set of flaky tests, we also aim to go deeper into the characteristics of the test cases for recognising the "interesting" causes of flakiness, i.e., for understanding which are the kinds of flakiness that could be associated with hard-to-detect failures. In this way future research could investigate what are the best tools that support the identification of these kinds of flaky tests, and help testers to distinguish them from uninteresting flakiness due to bad test design.

Our study certainly presents some limitations. Among the others, the tests reported within the IDoFT database have a strong focus on unit testing. This aspect has some, limited, consequences with respect to the investigation on if/how the behaviour of known intermittent tests changes when they are executed In Vivo, which is the main objective of this work. However, we acknowledge that this study did not cover test intermittence that could directly stem from a distributed environment (e.g. integration tests in the context of Web applications, or Web Services). Studying intermittence In Vivo in these scenarios raises concerns about the isolation of the test execution from the rest of the environment. Although Groucho offers abstract primitives for dealing with data persisted/exported out-of-memory (i.e. see Section 2.3), in future work we plan to include further qualitative and quantitative analyses covering also these scenarios.

Our study has been quite cumbersome in terms of the effort we had to put into parametrising the test cases and rerunning the flaky tests both In House and In Vivo. Clearly, this is not what we would imply that testers should do in practice. The implications for the software development process that descend from our research are related to some suggestions we could already provide to testers, as for example: encouraging the development of parametric tests; identifying when a specific state/configuration of the SUT is reached so that an In Vivo testing session has to start; or formulating policies that shift In Vivo the execution for those tests that manifested intermittent outcomes in the CI pipelines. However, all these topics certainly require better awareness of the types of flakiness that are due to hard-to-detect failures, which we plan to develop in future experimentation.

CRediT authorship contribution statement

Morena Barboni: Methodology, Software, Investigation, Validation, Writing – original draft. **Antonia Bertolino:** Conceptualization, Methodology, Investigation, Validation, Writing – original draft. **Guglielmo De Angelis:** Methodology, Software, Investigation, Validation, Writing – original draft.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Guglielmo De Angelis reports financial support was provided by Government of Italy Ministry of Education University and Research.

Replication package and data availability

The replication package that supports complete reproduction of the results reported in this paper is available from the following Github repository: https://github.com/IASI-SAKS/groucho/releases/tag/FastJsonInvivoFlakiness-v1.0.

The version of the `jvm-serializers` we referred in this study is available at: https://github.com/gulyx/jvm-serializers/tree/fastJSON-benchmark-only. The repository also provides a JAR which distributes the pre-built version of the benchmark.

Acknowledgements

This paper has been supported by the Italian MIUR PRIN 2017 Project: SISMA (Contract 201752ENYB), and partially by the Italian Research Group: INdAM-GNCS.

References

[1] O. Parry, G.M. Kapfhammer, M. Hilton, P. McMinn, A survey of flaky tests, ACM Trans. Softw. Eng. Methodol. 31 (1) (2021) 1–74.
[2] W. Lam, P. Godefroid, S. Nath, A. Santhiar, S. Thummalapenta, Root causing flaky tests in a large-scale industrial setting, in: Proc. ACM SIGSOFT ISSTA 2019, Beijing, China, ACM, 2019, pp. 101–111.
[3] M. Eck, F. Palomba, M. Castelluccio, A. Bacchelli, Understanding flaky tests: the developer's perspective, in: Proc. ACM ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, ACM, 2019, pp. 830–840.
[4] M. Fowler, Eradicating non-determinism in tests, Martin Fowler Pers. Blog (2011).
[5] L. Gazzola, L. Mariani, F. Pastore, M. Pezze, An exploratory study of field failures, in: Proc. IEEE ISSRE, IEEE, 2017, pp. 67–77.
[6] A. Tahir, S. Rasheed, J. Dietrich, N. Hashemi, L. Zhang, Test flakiness' causes, detection, impact and responses: A multivocal review, J. Syst. Softw. 206 (2023) 111837, http://dx.doi.org/10.1016/j.jss.2023.111837.
[7] M. Machalica, W. Chmiel, S. Swierc, R. Sakevych, How do you test your tests? 2020, Online on https://engineering.fb.com.
[8] J. Raine, Reducing flaky builds by 18x, 2020, Online on https://github.blog.
[9] S. Habchi, G. Haben, M. Papadakis, M. Cordy, Y. Le Traon, A qualitative study on the sources, impacts, and mitigation strategies of flaky tests, in: Proc. ICST 2022, IEEE, 2022, pp. 244–255.
[10] A. Bertolino, G. De Angelis, B. Miranda, P. Tonella, Run java applications and test them in-vivo meantime, in: Proc. ICST 2020, Porto, Portugal, IEEE, 2020, pp. 454–459.
[11] M. Barboni, A. Bertolino, G. De Angelis, What we talk about when we talk about software test flakiness, in: Proc. QUATIC 2021, Springer International Publishing, 2021, pp. 29–39.
[12] O. Parry, G.M. Kapfhammer, M. Hilton, P. McMinn, Surveying the developer experience of flaky tests, in: 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE SEIP 2022, Pittsburgh, PA, USA, May 22–24, 2022, IEEE, 2022, pp. 253–262, http://dx.doi.org/10.1109/ICSE-SEIP55303.2022.9793965.
[13] P.E. Strandberg, T.J. Ostrand, E.J. Weyuker, W. Afzal, D. Sundmark, Intermittently failing tests in the embedded systems domain, in: S. Khurshid, C.S. Pasareanu (Eds.), Proc. ACM ISSTA 2020, Virtual Event, USA, ACM, 2020, pp. 337–348.
[14] A. Bertolino, P. Braione, G. De Angelis, L. Gazzola, F.M. Kifetew, L. Mariani, M. Orrù, M. Pezzè, R. Pietrantuono, S. Russo, P. Tonella, A survey of field-based testing techniques, ACM Comput. Surv. 54 (5) (2021) 92:1–92:39.
[15] S. Elbaum, M. Diep, Profiling deployed software: Assessing strategies and testing opportunities, IEEE Trans. Softw. Eng. 31 (4) (2005) 312–327.
[16] J. Morán, A. Bertolino, C. de la Riva, J. Tuya, Towards ex vivo testing of MapReduce applications, in: 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS, IEEE, 2017, pp. 73–80.
[17] A. Bertolino, G. De Angelis, A. Guerriero, B. Miranda, R. Pietrantuono, S. Russo, DevOpRET: Continuous reliability testing in DevOps, J. Softw. Evol. Process (2020).
[18] A. Bertolino, G. De Angelis, B. Miranda, P. Tonella, In vivo test and rollback of Java applications as they are, Softw. Test. Verif. Reliab. (2023) e1857, http://dx.doi.org/10.1002/stvr.1857.
[19] A. Alshammari, C. Morris, M. Hilton, J. Bell, FlakeFlagger: Predicting flakiness without rerunning tests, in: Proc. IEEE/ACM ICSE 2021, Madrid, Spain, IEEE, 2021, pp. 1572–1584.
[20] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, A. Memon, Modeling and ranking flaky tests at apple, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, 2020, pp. 110–119.
[21] W. Lam, S. Winter, A. Astorga, V. Stodden, D. Marinov, Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects, in: 2020 IEEE 31st International Symposium on Software Reliability Engineering, ISSRE, IEEE, 2020, pp. 403–413.
[22] W. Lam, R. Oei, A. Shi, D. Marinov, T. Xie, iDFlakies: A framework for detecting and partially classifying flaky tests, in: Proc. IEEE ICST 2019, Xi'an, China, IEEE, 2019, pp. 312–322.
[23] A. Shi, W. Lam, R. Oei, T. Xie, D. Marinov, iFixflakies: a framework for automatically fixing order-dependent flaky tests, in: Proc. ACM ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, ACM, 2019, pp. 545–555.
[24] A. Shi, A. Gyori, O. Legunsen, D. Marinov, Detecting assumptions on deterministic implementations of non-deterministic specifications, in: Proc. IEEE ICST 2016, Chicago, IL, USA, IEEE, 2016, pp. 80–90.
[25] M. Barboni, A. Bertolino, G. De Angelis, Insights from Running Flaky Tests into the Field: Extended Version, Tech. Rep., ISTI-2022-TR/007, 2022, http://dx.doi.org/10.32079/ISTI-TR-2022/007.
[26] R. Kohavi, R. Longbotham, Online controlled experiments and A/B testing, Encycl. Mach. Learn. Data Min. 7 (8) (2017) 922–929.
[27] V. Massol, T. Husted, JUnit in Action, Manning, 2004.
[28] D. Silva, L. Teixeira, M. d'Amorim, Shake it! detecting flaky tests caused by concurrency with shaker, in: Proc. IEEE ICSME 2020, Adelaide, Australia, IEEE, 2020, pp. 301–311.
[29] M. Cordy, R. Rwemalika, A. Franci, M. Papadakis, M. Harman, FlakiMe: Laboratory-controlled test flakiness impact assessment, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 982–994, http://dx.doi.org/10.1145/3510003.3510194.
[30] O. Parry, G.M. Kapfhammer, M. Hilton, P. McMinn, Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models, Empir. Softw. Eng. 28 (3) (2023) 72, http://dx.doi.org/10.1007/s10664-023-10307-w.
[31] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, C. Rosenthal, Chaos engineering, IEEE Softw. 33 (3) (2016) 35–41.
[32] C. Murphy, G.E. Kaiser, I. Vo, M. Chu, Quality assurance of software applications using the in vivo testing approach, in: Proc. ICST 2009, Denver, Colorado, USA, IEEE, 2009, pp. 111–120.
[33] R. Rwemalika, M. Kintis, M. Papadakis, Y.L. Traon, P. Lorrach, An industrial study on the differences between pre-release and post-release bugs, in: Proc. IEEE ICSME 2019, Cleveland, OH, USA, IEEE, 2019, pp. 92–102.
[34] D. Cotroneo, R. Pietrantuono, S. Russo, K.S. Trivedi, How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation, J. Syst. Softw. 113 (2016) 27–43.
[35] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, K.S. Trivedi, Fault triggers in open-source software: An experience report, in: Proc. IEEE ISSRE 2013, Pasadena, CA, USA, IEEE, 2013, pp. 178–187.
[36] D.G. Cavezza, R. Pietrantuono, J. Alonso, S. Russo, K.S. Trivedi, Reproducibility of environment-dependent software failures: An experience report, in: Proc. IEEE ISSRE 2014, Naples, Italy, IEEE, 2014, pp. 267–276.