Software Quality Journal (2024) 32:5–26
https://doi.org/10.1007/s11219-023-09616-6

MUTTA: a novel tool for E2E web mutation testing

Maurizio Leotta[1] · Davide Paparella[1] · Filippo Ricca[1]

Accepted: 18 January 2023 / Published online: 17 April 2023

Abstract
Mutation testing is an important technique able to evaluate the bug-detection effectiveness of existing software test suites. Mutation testing tools exist for several languages, e.g., Java and JavaScript, but no solutions are available for managing the mutation testing process for entire web applications, in the context of end-to-end (E2E) web testing. In this paper, we propose MUTTA, a novel tool able to automate the entire mutation testing process. MUTTA mutates the various server source files of the target web application, runs the E2E test suite against the mutated web applications, and finally collects the test outcomes. To evaluate MUTTA, we designed a case study using the mutated versions of the target web application with the aim of comparing the effectiveness of two different approaches to E2E web testing: (1) test cases based on classical assertions and (2) test cases relying on differential testing. In detail, MUTTA has been executed on two web applications, each equipped with different test suites to compare assertions with differential testing. In this scenario, MUTTA generated a large number of mutants (more than 15k overall), took into account the coverage information to consider only the mutants actually executed, deployed the mutated web app, ran the entire E2E test suites (about 87k tests runs overall), and finally, it correctly saved the test suite results. Thus, results of the case study show that MUTTA can be successfully employed to automate the entire mutation testing process of E2E web test suites and, therefore, can be used in practice to evaluate the effectiveness of different test suites (e.g., based on different techniques, E2E frameworks, or composed by a different number of test scripts).

Keywords Mutation testing · End-to-end web testing · Web application · Selenium WebDriver · Recheck · Assertions

1 Introduction

Ensuring the quality of web applications is of paramount importance since these applications are used across many industries and businesses (e.g., commerce, entertainment, banking, work, study, etc.). Among the various approaches and methods used, the most

✉ Maurizio Leotta
maurizio.leotta@unige.it

[1] Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Genoa, Italy

adopted in practice is End-to-End (E2E) testing (Cerioli et al., 2020), realized with testing frameworks such as, e.g., Selenium WebDriver (García et al., 2020), Cypress, Katalon, Serenity, and TestRigor.

E2E testing of web applications is a type of black-box testing based on the concept of test scenario, which is a sequence of steps/actions performed on the web application under test (e.g., insert username, insert password, click the login button, etc.). One or more test cases can be derived from a single test scenario by specifying the actual data to use in each step and the expected results (i.e., defining the assertions). The execution of the test cases produced can be automated by implementing them in a programming language (e.g., Java or Python) and adopting a specific E2E testing framework able to drive a browser similar to how a human would (Leotta et al., 2016).

However, when it is possible to employ different E2E test suites to test a web application or testing frameworks to produce them, it becomes essential to have a method to objectively evaluate the effectiveness in detecting bugs and support the choice of which type of test suite or framework to adopt (Leotta et al., 2013, 2014, 2022). This often happens in practice; in fact, managers often find themselves in such a situation, for example, (1) when different E2E techniques/tools have to be evaluated to select the most suitable one in detecting bugs, or (2) when it is necessary to reduce a test suite because it is too long to run (Olianas et al., 2021, 2022), and thus finding a trade-off between execution time and ability to detect bugs.

Mutation testing is a technique used from many years to evaluate the effectiveness of test suites in detecting bugs (Jia & Harman, 2011; Offutt & Untch, 2001). Basically, the application's under test source code is automatically modified by inserting variations of the original software code simulating typical errors a developer could make during development and maintenance activities. For this reason, each mutated line is conceptually equivalent to a possible bug introduced accidentally by a developer into the code. After the generation of mutants, the test suite is run against each mutated version of the application in order to evaluate how many mutants are killed (i.e., the original version differs from the mutant). Mutants are traditionally used in testing to evaluate the quality of the produced test suites, hence the name of mutation testing. Indeed, the mutants can be used to identify the weaknesses of the test suites, e.g., by determining the parts of a software that are poorly or never checked.

While mutation testing tools exist for several languages used in the web context, no turnkey solutions are available to manage the mutation testing process for entire web applications in the context of E2E web testing.

In this paper, we propose and describe Mutta, a novel tool able to automate the entire mutation testing process for a web application. Mutta is able to mutate the various server-side sources of a web application creating different mutated web applications containing each only a single mutation. Once the mutations are produced, Mutta is able to run the E2E test suites (to be evaluated) against the target web application and collect the test outcomes.

To evaluate the effectiveness of Mutta, we devised and executed a case study with the goal of comparing two different approaches to E2E web testing: (1) test cases based on classical assertions vs (2) test cases relying on differential testing. Differential testing (Gulzar et al., 2019; McKeeman, 1998) of web applications compares the current web page under test with a snapshot considered correct taken from a previous version. This technique appears to be promising and an alternative to assertions in catching regressions due to the evolution of the web application under test (Leotta et al., 2022).

This paper extends a previous conference paper (Leotta et al., 2022) even if it has a completely different goal. Indeed, the aim of the previous conference paper was to empirically compare classical assertions and differential testing considering several factors: test script development time, test script execution time, and bug-detection capability. During the execution of that experiment we realized the importance of building and delivering a tool able to support mutation testing of web applications (which, to the best of our knowledge, is not yet available). For this reason, we have decided to extend, refine, generalize, and make freely available the tool that, in a preliminary version, we used to compare classical assertions and differential testing (Leotta et al., 2022). This way, using Mutta, anyone (both from academia and industry) can perform comparative studies between E2E test suites realised in differently and with different frameworks/tools.

This paper is organized as follows: Sect. 2 describes the details of Mutta. Section 3 describes the empirical study we carried out to evaluate the effectiveness of Mutta in supporting mutation testing of E2E web test suite and also introduces the approaches to implement oracle mechanisms (i.e., classical assertions and differential testing) used to implement the test suites analyzed with Mutta. Section 4 reports the results of the study. Sect. 5 summarizes the related literature while Sect. 6 concludes the paper.

2 Mutta: a tool for mutation testing of E2E web test suites

Implementing a mutation testing tool tailored to the E2E web testing context requires automating several procedures. Indeed, the specific kind of artefact under test, represented by an entire web application, makes the problem much more challenging than, for example, the classic mutation testing at the unit testing level involving functions or class methods.

2.1 Mutation testing

Now we briefly recap the concept of mutation testing and mutants. Mutants are variations of the original software code simulating typical errors a developer could make during development and maintenance activities (Jia & Harman, 2011; Offutt and Untch, 2001). For this reason, each mutated line (a line containing a mutant) is conceptually equivalent to a possible bug. Mutants are traditionally used in testing to evaluate the quality of the produced test suites, hence the name of mutation testing. Indeed, the mutants can be used to identify the weaknesses in the verification artefacts by determining the parts of a software that are poorly or never checked (Kochhar et al., 2015). As a consequence, if a test can detect the mutants is likely to have a good chance of catching bugs.

Mutation operators are the core of the mutation phase. They affect small portions of code, simulating several kinds of typical programming mistakes, like a change in a logical/mathematical operator (e.g., AND/+ instead of OR/-), a boolean substitution (e.g., from true to false), or a conditional removal (e.g., an IF condition statement is set to true). Manually generating the mutants in a realistic scenario is clearly infeasible (and possibly biased in the context of an experiment). However, there exist automated tools (used in the context of mutation testing, such as Pitest, as we will see in the next of the paper) that provide operators for generating a large number of mutants starting from the original code.

2.2 Mutta overview

A tool for mutation testing of E2E web test suite should be able to:

- Define the actual source code mutations for the various server-side source files composing the web application under test
- Apply each mutation to the source code (one at a time)
- Deploy the modified web application on the Web Server
- Run the modified web application under test
- Run the test suites against the modified web application
- Collect, elaborate and save all the test results
- Stop the web application and restore the original app and return to the second step if other mutations have to be considered

Mutta is the tool we developed to automate all these procedures and so to implement E2E web mutation testing effectively. The name stands for "automatic MUTator for web e2e Test Automation". Our tool can be downloaded from the following repository: https://sepl.dibris.unige.it/MUTTA.php.

The high-level goal of our tool is to enable the tester to compare the bug-detection capability of different test suites: from those, for instance, differing only for the number of test scripts they contain (helpful in evaluating the trade-off between test suite completeness and execution time), to those implemented with different E2E test approaches (useful to compare the bug-detection capability of different technical solutions in the context of E2E web testing). Our tool allows to automatically analyze how many mutants are killed by the considered test suites, and thus it provides an estimation of the capability of the different test suites in detecting actual bugs in the web application when such test suites are employed in production, for instance, in regression testing. Mutta is actually formed by a set of different components that automate various steps and operations, such as the source code mutation of the web application under test, the execution of test suites against (hundred or even thousands) versions of the mutated web application, and finally the retrieval of test results. A high-level overview of the architecture of Mutta is summarized in Fig. 1. The Mutta orchestrator supervises the execution of the various components of the tool (depicted in green). Mutta requires in input the test suites to compare with mutation

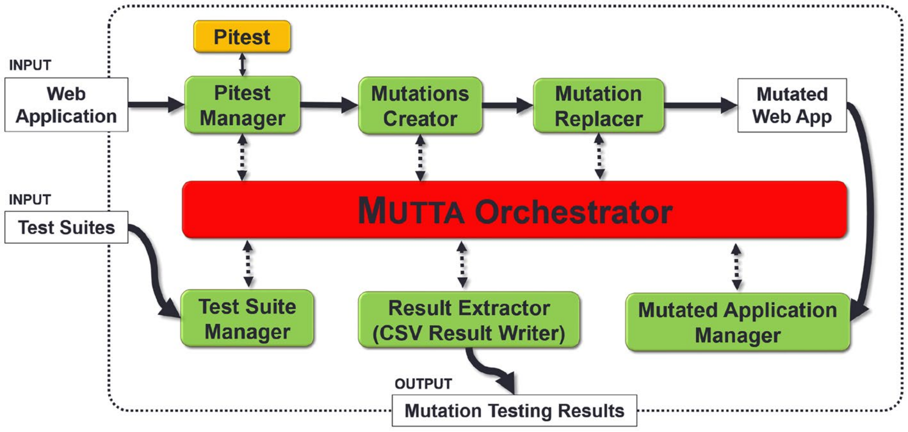


Fig. 1 Diagram of the Mutta high-level architecture

testing, the corresponding web application under test and provides in output the mutation testing results in the form of a report.

The current version of MUTTA has the following requirements concerning the Web applications (and the test suites):

- Open source: clearly all the source code files of the web application must be accessible by MUTTA since it has to mutate them;
- Java-based: since in this preliminary version of MUTTA we targeted web applications implemented server-side in Java, we employed a specific source code mutator able to operate on this programming language;
- *Apache Maven* as build automation tool: to automatize the application of the mutations to the source files, we relied on Maven, so the web app should be manageable with such a tool.

2.3 MUTTA execution steps

In the following, we provide an overview of the steps performed by MUTTA to implement mutation testing of E2E web application.

The first step consists in manually extracting the web application's source code and providing it to Pitest (additional details on this tool are in Sect. 2.3.1). Pitest executes mutation testing at the unit level (so it mutates the code inside class methods) and produces for each mutation a detailed description. Thus the output of this first step is a detailed list of Mutation descriptions. The Pitest Manager is the MUTTA component that is responsible for managing the execution of Pitest and collecting the mutation logs.

At this point, the main procedure of MUTTA can start. It is graphically summarized in Fig. 2.

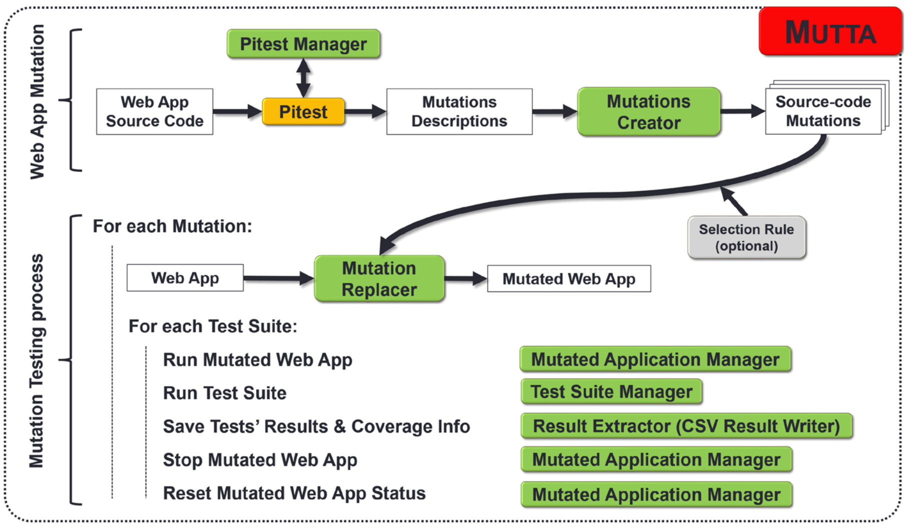


Fig. 2 Diagram of the MUTTA main operations and pseudocode of the mutation testing loop process

As said before, Mutta is composed of a set of components (highlighted in green in the figure) implementing specific functionalities (details on them are provided in the following of the section).

First, Mutta analyzes the mutation log descriptions provided by Pitest, interprets and transforms them into actual statements in order to be applicable to the original source code. This step is carried out by the Mutations Creator component.

At this point, the main loop implementing the mutation testing process is executed. Basically, for each mutation originally generated by Pitest and concretely defined by the Mutations Creator, Mutta applies it to the source code, deploys the mutated web app, executes the entire E2E test suite, saves the test suite results, and finally reverts to the original version of the web app. This is done by the several components highlighted in green in the figure.

In conclusion, the results of the entire mutation testing process are provided to Mutta user.

2.3.1 Source code mutations with Pitest

As aforementioned, to actually define the mutations to apply to the source code of the web application Mutta relies on the *Pitest*[1] mutator that is called by the Pitest Manager component. Pitest is a Maven plug-in that can create code mutations working at the bytecode level. Pitest currently provides many built-in mutators able to modify the bytecode in many ways; the complete list of mutators, all supported by Mutta, can be found in the tool web site[2]. Since Pitest was thought as an independent and complete mutation testing tool at unit level[3], it does not provide in output the mutated source files but only a detailed log description of the mutations it applied during the mutation testing process. The description is provided at the unit level, detailing which kind of change has been applied to the class method source code. This is reasonable since the source code mutation can be considered an internal step in a standard mutation testing process, and it is unnecessary to save all the mutant versions of the source code generated.

Listing 1 shows a real example of *mutation* information for the Shopizer web application source code. We can see that Pitest provides information concerning the source file, the class, and the specific method mutated as well as the description of the mutation applied.

```
<mutation>
    <sourceFile>UserReset.java</sourceFile>
    <mutatedClass>
        com.sales.shop.admin.model.userpassword.UserReset
    </mutatedClass>
    <mutatedMethod>generateRandomString</mutatedMethod>
    <lineNumber>22</lineNumber>
    <mutatorType>ConditionalsBoundaryMutator</mutatorType>
    <description>changed conditional boundary</description>
</mutation>
```

Listing 1. Example of mutation information for Shopizer from Pitest tool

[1] https://pitest.org/

[2] https://pitest.org/quickstart/mutators/

[3] Unit testing is a method which tests individual units of source code.

2.3.2 Mutations creator

Starting from the logs generated by Pitest and collected by the Pitest Manager, Mutta converts the description of mutations to actual *mutations* in the source code of the *AUT* (application under test). For each mutation description, this component searches for the original version of the statement line in the source code, interprets the mutation description, and produces a novel mutated statement line ready to be inserted in the source code. This process is repeated for each mutation described in the Pitest logs. The final output is a *json* file with all the information about mutations ready to be used by the following components of Mutta. The Mutations Creator is able to generate the exact mutated statement ready to be inserted in the source code. The precise meaning of the original Pitest textual descriptions, in terms of actual source code mutations, are defined in the Pitest documentation: this allowed us to implement the Mutations Creator covering the mutations that Pitest can generate. To actually define the novel mutated statements, the Mutations Creator uses strings replacement using regular expressions (*regex*). From our analysis on the various web applications used to evaluate Mutta we discovered that about 95% of the mutations are supported using this kind of solution that has the merit to be extremely fast compared with more advanced syntax-based changes (note that the mutations to manage can be in the order of tens of thousands, as for one of the considered web apps in the empirical evaluation, but even potentially more for large industrial applications).

The example reported in Listing 2 is a piece of real mutation information created in *json* format by Mutta.

```json
{
    "id": 6,
    "sourceFilename": "UserReset.java",
    "sourceRootPath":
      "/home/.../shopizer-mutations/sm-shop/src/main/java",
    "relFolderPath": "com/sales/shop/admin/model/userpassword/",
    "lineNumber": 22,
    "originalLine": "for (int i=0; i < RANDOM_STRING_LENGTH; i++) {",
    "mutatedLine": "for (int i=0; i <= RANDOM_STRING_LENGTH; i++) {",
    "mutatorTag": "CONDITIONAL_BOUNDARY",
    "MasterID": 7841,
    "methodName": "generateRandomString"
}
```

Listing 2. Example of Mutta mutation information for Shopizer

In case the number of mutations generated is too high, we implemented an optional selection rule in Mutta to limit the number of mutants to consider. This is important (as we will see in the empirical evaluation) since, for every mutant generated, all the entire E2E test suites under comparison must be run. This process typically requires an execution time in the order of several minutes for each mutation, making the evaluation over thousands (or tens of thousands) of mutants simply infeasible. The optional selection rule applies a cap to the number of mutants to consider sampling the entire set while maintaining (as much as possible) invariant the proportion of the various types of mutants generated and their distribution among different methods.

2.3.3 Mutation replacer

This component applies the *mutations* to the source code, one at a time. Basically, it replaces the original line of code with the mutated one. Moreover, when necessary, it is able to modify other parts of the source code to make the mutated code correct for the Java compiler.

2.3.4 Mutated application manager

This component is in charge of controlling and managing the application under test. It exposes methods to the rest of Mutta to easily control the web application execution life cycle. Thus it can run and stop the application using Maven, capturing the console output, especially to know when the application is ready and has completed the boot-up: in this way, Mutta can be aware of when the test suite execution can be started. Further, this component has also to reset the state of the application under test and remove any evidence of previous runs: this is a fundamental feature that allows running the test suite against each mutation in isolation, without any influence due to the previous runs.

2.3.5 Test suite manager

This component is in charge of controlling and managing the Test Suite execution. Basically, it provides methods to run the test suite against a specific web application and to save detailed test suite execution results. Additionally, it is able to save the output of the plug-in “Surefire”.

2.3.6 Result extractor/CSV result writer

This component provides methods to extract additional information from the output of Surefire plug-in. It reads the *xml* file and gets the field needed by Mutta: the number of tests passed and failed with error and skipped, details for each test executed (name and detailed result), and test suite execution time. This component also provides the coverage of the test suites against the mutants: it analyzes whether each mutated line in the web application is executed or not for each test suite execution. This information is fundamental to compute the actual bug-detection effectiveness: indeed, all mutants generated but that are not covered during the test suite execution should not be considered in the final evaluation. Indeed, it is impossible for a test suite to kill a mutant on a specific source code line (i.e., detect the corresponding bugs) if that specific line of the web application is not triggered (and thus covered) by the test suite execution.

3 Empirical evaluation of Mutta

This section describes the definition, design, and settings of the case study we conducted following the guidelines by Wohlin et al. (2012) and Runeson et al. (2012) to design experimental studies.

The *goal* of the study is to investigate the effectiveness of Mutta in supporting mutation testing of E2E web test suites and its applicability to real case studies. The results of the experiments are interpreted from two *perspective*s: (1) researchers interested in empirically evaluating the effectiveness of Mutta in implementing mutation testing for E2E Web test suites, and (2) software quality managers who want to understand whether Mutta can be used in practice to evaluate and compare the bug-detection capability of different test suites (e.g., implemented using different technical solutions or having different sizes).

To evaluate if the mutation testing process implemented by Mutta is actually able to effectively highlight differences in the bug-detection capability of different E2E web test suites, we need a set of different test suites. To provide a comparison that could be considered valuable by itself, and not only for evaluating Mutta, we have decided to compare the effectiveness in detecting bugs of test suites based on different oracle mechanisms. The following Sect. 3.1 briefly describes the two approaches to implement oracle mechanisms (i.e., classical assertions and differential testing) while Sect. 3.2 briefly describes the tools and frameworks implementing the three test suites to compare for each web application.

3.1 Classical assertions and differential testing

Assertions The classical way for evaluating tests' results is to employ assertions. A *test assertion* is a boolean expression that asserts if the output of the system under test is correct (i.e., assert expression is *true*) or not (i.e., assert expression is *false*). Usually, a single assertion verifies a chunk of independent information and asserts if the expected information is the actual output information of the AUT. Some examples of value-based assertions taken by JUnit are the following: *assertTrue, assertFalse, assertNull, assertNotNull, assertEqual, assertNotEqual*. Some, like the equality ones, need two values (*expected* and *actual*), while others need only one. To create an assertion, a human tester must have a good knowledge of the AUT because they must know the expected result to be checked in advance.

Differential testing (or *diff testing*) is a testing technique formally introduced by McKeeman as a new method for regression testing of large software systems (Gulzar et al., 2019; McKeeman, 1998). Differential testing consists of a comparison of outcomes: these are generated by the execution of two different systems' versions, both using the same system under test and the same inputs. One is the *test* version, modified and needing to be tested, and the other is the *base* version that is previously verified and guaranteed to produce a correct outcome (Gulzar et al., 2019). The *base* version could be a live version of the software that can be executed whenever the testing procedure is launched; alternatively, it can be just statically stored in case of a fixed expected outcome. In practice, this approach verifies that the behavior of the software remains unchanged. Thus, a difference between the two versions (test and base) highlights a likely bug in the new version. Differential testing is generally closely associated with regression testing due to the natural ability to catch bugs introduced in newer software versions. The main peculiarity of diff testing is that no manually created *oracle* is required: the *base* version of the system under test, that is verified to be correct, represents the oracle itself. "A base version is chosen with the assumption that it is bug-free" (Gulzar et al., 2019) is an ambitious assumption and maybe either the strong or the weak point of diff testing. On the contrary, the assertions created by test developers play the role of the oracle. Diff testing attempts in part to solve the problem of the generation of the oracle; due to the issue about its generation, differential testing is more applicable to software whose quality is already under control, with few known errors (McKeeman, 1998). Indeed, applying differential tests to software with many active development bugs and changes between two versions is harmful to the testing process. When a tester approaches the creation of a new test suite, they usually analyze the "functional requirements", considering them a complete and correct specification, like an "*oracle*". The following step is to "translate" the requirements specification in a test suite that checks and validates them. The assertions can be more or less thorough, but the procedure is always the same: i.e., asserting if the application implements the described functionality correctly. A fundamental difference between differential testing and assertions in the web application context concerns "what is tested". With assertions, usually, only the functional part of the SUT is verified. On the contrary, differential testing also considers other aspects, not only functional ones, such as the style or GUI-related changes, since it compares the entire web pages.

3.2 Testing tools and framework considered to implement the test suites

Here we describe the tools and frameworks we used to actually implement the test suites analyzed by MUTTA and following the two approaches: classical assertions vs differential testing.

Selenium WebDriver[4] is a testing framework belonging to the Selenium ecosystem[5] which "drives a browser natively, as a user would" (Project, 2021); more specifically, it is an object-oriented API that allows test developers to write test scripts able to drive browsers effectively. This framework is used to automate web-based application testing to verify that the AUT performs as expected (Unadkat, 2021).

The great success of this framework is mainly due to two aspects:

1. it is open-source and thus freely modifiable and usable;
2. test scripts can be written in any programming language (e.g., Java or Python); therefore a test suite can be developed and maintained like every other software project.

We chose Selenium WebDriver in our experiment because it is a mature, open-source, and widely used state-of-the-art framework for web application testing (García et al., 2020; Leotta et al., 2016). The assertions were produced using JUnit 5.

Differential Testing with Recheck *Recheck* is a testing framework supporting differential testing in the context of web testing. It is proposed by *Retest*, a company[6] based in Germany founded in 2017 that provides a specific set of tools for test automation. *Recheck* is constituted by four different software products. In our experiment, we used *Recheck-web Maven plug-in*, which integrates with Selenium WebDriver and replaces assertions with differential testing[7].

In practice, Recheck is a library written in Java that is importable in any JVM-based test suite project. It provides methods to apply differential testing on Selenium Web Elements. Given a Selenium Web Element (or also the entire driver), Recheck does the following: (1) generates a snapshot if one does not already exist, (2) compares the snapshot previously generated with the current one. Recheck fails the test if (1) no snapshot is present, so there is nothing to compare with (it happens in the very first run of the test); and (2) at least one

[4] www.selenium.dev/documentation/webdriver/

[5] www.selenium.dev/

[6] https://retest.de

[7] https://github.com/retest/recheck-web

difference is found between the previously stored snapshot and the current one. **Golden Master** is the name chosen to call the stored snapshots of a web element (or a whole web page). This snapshot is elaborated exclusively from the HTML and CSS code. Recheck creates the Golden Master with its own format (in *xml* language), storing all the information that is needed to represent the page itself. The Golden Master is generated the first time the test script is executed, or more generally, whenever the execution does not find the corresponding Golden Master to compare with. A single test can have one or more Golden Masters associated; every check required in the test has its own Golden Master. This is the *oracle* mechanism of Retest's differential testing.

Recheck provides two differential testing approaches[8] inside Selenium tests:

1. *Explicit check*. By means of the Recheck object, it is possible to explicitly call the Recheck check in the Java test method at any point of the test. With it we can create an instance of Recheck, e.g., via `Recheck re = new RecheckImpl()` and check the complete current webpage via `re.check(driver, “check-name”)` or individual web elements via `re.check(webElement, “check-name”);`
2. *Implicit check*. Using the Recheck WebDriver that wraps the Selenium WebDriver, Recheck implicitly performs automatic checks inside the test, basically one check after each WebDriver action.

3.3 Context

The *context* of the study is constituted by a professional software tester that developed the three different test suites and then executed Mutta, and two open-source web applications. The **professional tester** conducted the experiment under the supervision of the researchers. He is a full-stack developer and software tester with more than five years of experience in the E2E web testing field. He has good knowledge of developing Selenium WebDriver test suites. Moreover, before performing the experiment, he practiced with Recheck by creating several sample test suites for various web applications.

3.4 Web applications

The two web applications included in the study are: **Petclinic** and **Shopizer**. Both applications have common technical properties important for our experiment, such as: being open-source, written in Java programming language, based on *Spring Boot* framework[9], rely on *Apache Maven* as build automation tool, are bootable by Maven command locally, support the *Pitest* Maven plug-in (important for applying mutations automatically with Mutta), support the *Surefire* Maven plug-in (important for collecting testing reports automatically with Mutta). Some of these characteristics were fundamental to automate the mutation test with Mutta as described in Sect. 2.

Petclinic (@spring-projects/spring-petclinic on GitHub[10]) is a sample application of Spring. It is the official application built by the developers of Spring Boot framework to demonstrate how to use it. It allows managing basic data simulating a veterinary clinic.

8 https://retest.de/feature-unbreakable-selenium/

9 https://stackify.com/what-is-spring-boot/

10 https://github.com/spring-projects/spring-petclinic

Shopizer (@shopizer-ecommerce/shopizer on GitHub[11]) is a customizable e-commerce web application. It provides the creation of accounts and several functionalities about e-commerce. The HTML forms are rich and dynamic: there are actions with animations, therefore delayed, and there is a loading overlay in many parts of the app.

3.5 Research questions

The research questions of the study are the following:

RQ1 (Mutants Generated) How many mutants are generated by Mutta?
RQ2 (Mutants Coverage) How many of the generated mutants are actually covered by the test suite execution?
RQ3 (Test Suite Execution Time against Mutants) Which is the execution time required to execute the test suites against the generated mutants?
RQ4 (Bug-detection assessment) Is Mutta able to effectively highlight differences in the bug-detection capability of different E2E web test suites?

The metrics used to answer the RQs are: the overall number of mutants generated (RQ1), the coverage analysis of the generated mutants (RQ2), the execution time of the test suites against the mutants (RQ3), and the number of bugs detected by the different test suites (RQ4). The three test suites developed for each web app contain the same test cases (i.e., steps) but different test oracle mechanisms (i.e., assertions, Recheck with explicit checks, and Recheck with implicit checks).

3.6 Experimental procedure

Since Mutta requires test suites to evaluate, we asked the professional tester to develop three test suites for each of the two applications under test; one for the assertions, one for differential testing using the explicit checks of Recheck, and one using the implicit checks of Recheck. This means that there are six test suites in total. We, therefore, have three test suite types: (1) Assertions, (2) Recheck explicit, and (3) Recheck implicit. There are 53 test scripts in each Shopizer test suite type, whereas Petclinic has 31 test scripts. The test suites are realized as 'Java 8' projects using JUnit 5 as unit testing framework. Web element locators have been created using ChroPath[12], a Chrome plug-in that automatically generates XPaths inspecting the web element using the browser's developer tools. Selenium WebDriver has been adopted to perform the actions in the web browser for all three test suite types. To develop the three kinds of test suites for each web app, the developer applied the following procedure:

1. Analyze the AUT and select the functionalities to be tested, trying to reach a good coverage of the most important features available for a user;
2. Describe the test cases in Gherkin language;
3. Develop the test cases in Selenium WebDriver test scripts without any oracle mechanisms;
4. Forking the test suite in three different test suites (one for each treatment) and:

[11] https://github.com/shopizer-ecommerce/shopizer
[12] https://www.autonomiq.io/deviq-chropath.html

(a) Add the specific test oracle mechanism to each test script in order to have three distinct test suites (i.e., one with Assertions, one using Recheck implicit, and one using Recheck explicit). Assertions typically check a value on the current page, such as the total value of a cart, while the other test oracle mechanisms check multiple values as described in Sect. 3.2;

(b) Validate the application with the test suite; all test scripts must pass.

For Recheck tool, the tester tuned the ignore files (i.e., *ignore-rule*) so that the minimum number of rules is used to pass the tests. An ignore-rule[13] is a filter used to ignore volatile elements, attributes, or sections, using a Git-like syntax. This mechanism is very useful for avoiding false positives in the testing phase. For example, the portion of a page showing the current time changes from one snapshot to another, thus without an ignore-rule Recheck would highlight the difference causing the test script to fail without the presence of a real bug.

To answer RQ1, RQ2, and RQ3, the tester executed Mutta and noted down respectively: (1) the number of generated mutants per Web app; (2) the coverage analysis of the generated mutants; and (3) the execution time required to run the considered test suites against the generated mutants (we report the average times computed on five repeated executions to mediate any fluctuations).

To answer RQ4 the tester employed Mutta to measure the bug-detection capability of the three considered test suites (for each app) as the number of mutants detected by each test suite over the total number of mutants generated by Mutta. In particular, the metric we used to evaluate the overall test suite quality is the percentage of mutants killed out of the total (i.e., the higher, the better).

4 Results

The following sections report the results for answering the research questions of this study. In particular, to answer RQ1, RQ2, and RQ3, we analyzed respectively three main factors: (1) the number of generated mutants per Web app; (2) the coverage analysis of the generated mutants; and (3) the execution time required to run the considered test suites against the generated mutants. These analyses will provide (1) an overview of how the various components of Mutta work in a realistic mutation testing scenario and (2) some considerations on the applicability of Mutta to web applications of different sizes. Then, with RQ4, we provide an overview of the bug-detection assessment supported by Mutta that enables the comparison between different kinds of test suites: to this end, as a case study, we compared test suites differing for the adopted oracle mechanism (i.e., Assertions, and Recheck Explicit and Implicit).

4.1 RQ1 (Mutants Generated) How many mutants are generated by Mutta?

Concerning the two considered web applications, Mutta managed the generation through Pitest of 107 mutants for the Petclinic application: in total, the web app is composed of

[13] https://github.com/retest/recheck-web

Table 1 Distribution of the type of mutations generated for the Petclinic web application

Mutation type	%
RTN_EMPTY_STR	31%
REMOVE_CALL	24%
NEGATE_COND	21%
RTN_NULL	9%
RTN_EMPTY_COLLECTION	8%
RTN_TRUE	2%
RTN_ZERO_INTEGER	2%
RTN_ZERO_INT	1%
RTN_FALSE	1%

1932 lines of code, not counting blank lines and comments (overall about 5.5% of the lines have been mutated). Instead, in the case of Shopizer, 15025 mutants were generated since the web app is by far larger: 86333 lines of code, not counting blank lines and comments (overall, about 17.4% of the lines have been mutated). The reason why, in proportion, for Shopizer more mutants have been created (i.e., 5.5% vs 17.4%) is that when dealing with more complex code (as in the case of Shopizer) the number of statements that can be actually mutated increases: indeed, the standard boilerplate code is usually not mutated, so simple functions provide only a few mutants in general.

As an example, Table 1 provides an overview of the types of mutations actually generated for the Petclinic web application. We can observe that most mutations involve returning empty strings in class methods, removing calls to methods, and negating conditions (for instance, in conditional statements or loops). We highlight that Pitest can be configured to generate different types of mutations. In this study, we adopted the Pitest "default" settings, but it is possible to apply more mutation types by activating the "stronger" or "all" configurations; clearly, these choices led to a higher number of generated mutations and so higher execution time for the entire mutation testing process as we will discuss in RQ3.

Summary RQ1. From the execution of Mutta on the two considered web applications, we can observe that **Mutta** ***is able to correctly manage the execution of Pitest on each source file composing the web apps***, and finally, to *generate a reasonably large number of mutants* that can be employed for a detailed analysis of the difference in bug-detection capability among the considered test suites associated with the web applications.

4.2 RQ2 (Mutants Coverage) How many of the generated mutants are actually covered by the test suite execution?

Thanks to the coverage analysis functionality implemented in Mutta, we analyzed the effective mutation coverage reached by the test suites developed in the context of the empirical evaluation. Basically, Mutta labelled (in the original version of the app) all the lines mutated as covered if they were executed during the test suite execution against the original version of the app. Note that the coverage is the same for all three versions of each test suite (Assertions, and Recheck explicit and Implicit) since the Selenium WebDriver actions are the same and only the final check changes. During the evaluation, Mutta was able to provide a detailed analysis of the coverage from the test suites execution. In detail, the mutation coverage for Petclinic is 98 mutants out of 107

(91.6%). On the contrary, for Shopizer we found that the coverage is lower: only 1882 mutants were covered (12.5%) by the test suites. We analyzed the reasons behind this difference and found that the low percentage is due to the fact that (1) Shopizer is very complex and (2) the developer of the test suites focused only on the main features of the Shopizer web application; thus, the test suites cover a limited number of functionalities and cases managed in the mutated source code. This means that several mutate statements are not executed when running the test suites against the Shopizer application.

Summary RQ2. The obtained results highlight that for some applications *the coverage of the generated mutant can be really low* (as in the case of Shopizer, which is 12.5%). From this, it appears evident the ***importance of the coverage analysis functionality implemented in*** **MUTTA**: in large real web applications, it could be pretty hard to reach a high level of coverage of all the mutated lines of code: without a coverage analysis mechanism, like the one implemented in MUTTA, the mutation testing results can be unreliable since the majority of the mutants (that are actually never executed) are labelled as not killed, thus incorrectly standardizing the performances of the different approaches compared through mutation testing.

4.3 RQ3 (Test Suite Execution Time against Mutants) Which is the execution time required to execute the test suites against the generated mutants?

We analyzed the execution of the test suites against the mutated versions of the considered web applications generated by MUTTA, to better understand the feasibility and the potential costs of the mutation testing process implemented by our tool. In the case of the Petclinic test suites (composed of 31 test scripts), the execution against the 98 mutants (i.e., the mutants covered by the test suites) took about 6 hours: this is a reasonable value since the average time required for each test suite execution (i.e., against each mutated version of the web app) exceeds a minute.

On the other hand, in the case of Shopizer, the execution of the test suites (composed of 53 test scripts) against all the 1882 covered mutants generated would have taken too long: estimated at about 220 hours per test suite. This result highlights the importance of implementing an optional selection rule in MUTTA, in order to limit the number of mutants to consider. The application of the rule discarded a relevant part of the mutants by reducing the set of considered mutants to 491 (we set up the rule to select up to three mutations for each Java method, and therefore we have eliminated about the 74% of the mutations covered by the test suite). At this point, executing 491 mutants still took about 62 hours of computation to run the test suites. So, we decided to use virtual machines in order to parallelize the computation and speed up the evaluation. To this end, we set up a cluster of 10 virtual machines (based on Oracle VirtualBox) on which we executed MUTTA starting from the result of the selection rule. For each instance of MUTTA, executed on a virtual machine, we manually selected about 50 mutants in order to reduce the execution time to about 6 hours. Clearly, this solution requires to have several physical machines available or, as an alternative, a cluster on the cloud.

Summary RQ3. From the results obtained, we can say *that the execution time required to complete the mutation testing process can be very high*, in particular when considering real web applications since the number of generated mutants grow a lot (e.g., 220h estimated time for the Shopizer web application). These results highlight ***the importance of (1) the "selection rule" mechanism implemented in*** **MUTTA** and ***of (2) the possibility provided by*** **MUTTA** ***of executing in parallel, on different machines, disjointed partitions of the entire set of mutants***.

4.4 RQ4 (Bug-detection assessment) Is Mutta able to effectively highlight differences in the bug-detection capability of different E2E web test suites?

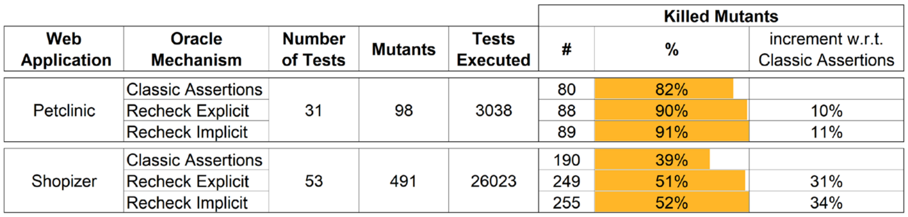
After having collected the result of the execution of Mutta for the two considered applications and for each of them of the three test suites implementing different oracle mechanisms (i.e., the case study comparing Assertions, and Recheck Explicit and Implicit), we are able to analyze the effectiveness of Mutta as mutation testing tool for E2E web test suites.

More in detail, see Fig. 3, we discovered that of the 98 considered mutants for the Petclinic web application (i.e., the bugs artificially inserted by mutation and covered by the original test suite), the test suite based on the classic assertions killed 80 (about 82%) of them, the one based on Recheck explicit 88 (about 90%), and finally the one based on Recheck implicit 89 (about 91%). In the case of Shopizer, 491 of the generated mutants were finally considered after applying the selection rule and the coverage analysis. The test suite for this web application based on classical assertions killed 190 of them (about 39%), the one based on Recheck explicit 249 (about 51%), and finally, the one based on Recheck implicit 255 (about 52%). From the figure, it is also evident how computationally intensive can be the execution of an E2E test suite against a quite large number of mutants: for example, in the case of Shopizer, for each oracle mechanism, a total of 26023 test scripts runs have been performed (53 tests compose each test suite repeated for each of the 491 mutants).

In the last column, Fig. 3 shows the percentage difference in the number of mutants killed with respect to the classical assertions. There is a clear advantage from using differential testing (both Recheck implicit and explicit) with respect to the assertions. Recheck implicit is slightly better at killing mutants than Recheck explicit, which is reasonable given the much higher number of differential checks. It is worth noting that for Shopizer, a realistic and complex web app, Recheck's differential testing methods have been able to kill a much higher amount of mutants, about 31–34% more compared to assertions. This can be explained why often a bug in complex code can unpredictably affect the behavior of the app — for instance, causing a slight modification of a web page — and being that differential testing checks the entire web page content (and not just a web element as assertions usually do) it has been found to be more effective.

Thus, thanks to the application of Mutta we can say that from this preliminary analysis, Recheck implicit appears to be the most effective solution; alongside the other techniques, it performs more checks during the test case actions, which makes it the most effective in finding bugs. However, Recheck explicit is also effective.

Summary RQ4. From the results obtained with the execution of our tool, we can say that **Mutta** ***can be considered an effective solution to the mutation testing problem of E2E test suites***. To propose an interesting case study, in this work we compared test suites where the test cases perform exactly the same steps, but the final oracle mechanism



Web Application	Oracle Mechanism	Number of Tests	Mutants	Tests Executed	Killed Mutants #	Killed Mutants %	Killed Mutants increment w.r.t. Classic Assertions
Petclinic	Classic Assertions	31	98	3038	80	82%	
	Recheck Explicit				88	90%	10%
	Recheck Implicit				89	91%	11%
Shopizer	Classic Assertions	53	491	26023	190	39%	
	Recheck Explicit				249	51%	31%
	Recheck Implicit				255	52%	34%

Fig. 3 Petclinic and Shopizer test suites effectiveness computed with Mutta

for each of them differs (we considered Classical Assertions, and Recheck Explicit and Implicit). However, Mutta can be used in more standard comparison to analyze, for instance, the effectiveness in the bugdetection capability of test suites implemented using the same technical solutions (e.g., Assertions only) but differing, for example, in the number of test scripts: this can be very helpful for the testers working in the industry to find a reasonable trade-off in a test suite reduction scenario, between the number of test scripts to run (the lower, the better) and the number of bugs that can be actually detected (the higher, the better).

4.5 Threats to validity

The main threats to validity affecting our empirical study are as follows: Internal, External, Construct, and Conclusion validity (Wohlin et al., 2012).

Internal Validity threats concern confounding factors that may affect a dependent variable. Concerning RQ1, the number of generated mutants can be influenced by the characteristics of the source code of the two web applications considered. For this reason, we employed two existing open-source web apps providing a realistic scenario for the application of Mutta. Concerning RQ2, the actual coverage of the mutations that can be reached depends on the completeness of the test suites and, also in this case, the characteristics of the source code of the considered web apps. To reduce this threat as much as possible, we developed the test suites independently from the mutation process implemented in Mutta and followed a precise procedure as described in the empirical study. Concerning RQ3, the execution time of the test suite against the mutants mainly depends on the number of mutants generated (and clearly also the time required to execute the entire test suite once). In the case of Petclinic, we showed the total execution time against all the covered mutants, while in the case of Shopizer, given the very high number of such kind of mutants (1882), we decided to apply the "selection rule" inserted in Mutta that can help to maintain the total execution time required by the mutation testing process at reasonable values. Finally, concerning RQ4, the main threat is probably related to the choice of the mutation tool adopted by Mutta (i.e., Pitest). Indeed, different tools could be potentially able to generate different set of mutants. This could change the mutant-detection capability of the three considered approaches. To reduce this threat as much as possible, we decided to adopt Pitest for Mutta, a mature mutation tool already used in other scientific works (Coles et al., 2016; Laurent et al., 2017; Papadakis et al., 2018). In fact, Pitest is capable of generating a variety of possible mutations in the web apps' source code, mimicking realistic bugs.

External Validity threats are related to the generalization of results. It is well-known that case study research is criticized mainly for two reasons: for lacking scientific rigour and providing little basis for generalization (Hollweck, 2014). In our case study, there could be two threats of this type. The limited number of web apps and the fact that only one developer was involved in the study. However, both the web apps employed in our research are examples of real systems, and the involved developer is very experienced in developing Selenium WebDriver test suites. This makes the context quite realistic, even though further studies with existing, more complex applications and more developers will improve the generalizability of the results.

Construct validity threats concern the relationship between theory and observation. Concerning our RQs, they are due to how we measured the results. To minimize this threat, we decided to measure all the results provided objectively, thanks to automated procedures that do not involve human intervention.

Threats to conclusion validity concern issues that may affect the ability to draw a correct conclusion, for example, using non appropriate statistical methods. This threat does not apply to our experiment because we did not use statistical tests.

5 Related works

To the best of our knowledge Mutta is the first tool with the goal of automatizing the mutation testing process of E2E test suites completely. In the literature, several researchers proposed mutation operators especially tailored for the web context and mutation approaches for web applications.

For instance, Praphamontripong and Offutt (Praphamontripong & Offutt, 2010) present a solution to the problem of integration testing of web applications by using mutation analysis. In their work, they first define new mutation operators specific for web applications, then present a tool called webMuJava, implementing these operators, and finally describe the results from a case study applying the proposed tool to a small web application. The results show that mutation analysis can help to produce tests that are effective at finding web application faults. WebMuJava differs from Mutta from many perspectives: (1) the testing level is different, its focus is on integration testing while Mutta supports the evaluation of functional test suites at "system level''; (2) webMuJava provides novel mutation operators while Mutta supports the entire mutation testing process and relies on the state of the art mutator Pitest to mimic realistic bugs in the web app source code.

In another work, Praphamontripong et al. (2016) define a set of web mutation operators targeting interaction faults in web applications. They carried out an experimental study on 11 web applications using 15 novel web mutation operators. In that study, they analyze the effectiveness of 12 independently developed test suites in terms of the capability of killing web mutants. The authors found that none of the manually created tests was able to kill a high percentage of mutants (the mutation scores ranged from 17% to 75%, with a mean of 47%). On the contrary, the mutation-based tests killed all the mutants generated in the experiment. The authors also analyze which mutants and mutation operators are difficult to be detected by traditional tests. We too have experienced in our experiment that it is difficult to kill all mutants with test cases designed starting from the functionalities offered by the application.

Mirshokraie et al. (2015) face the problem of the equivalent mutants in mutation testing of Javascript code that leads to high computational costs associated with a large pool of generated mutants. They propose a guided mutation testing technique that leverages dynamic and static characteristics of the system under test to selectively mutate portions of the code that exhibit a high probability of (1) being error-prone, or (2) affecting the observable behavior of the system, and thus being non-equivalent. In this way, the proposed technique can minimize the number of generated mutants. Even if the proposed mutation testing approach can be adopted with any programming language, the authors implemented and evaluated it for JavaScript code in the context of web applications. This kind of work could be useful in the near future to make the mutation testing process implemented in Mutta more efficient since it can benefit from an optimized selection of the mutants to consider.

Always in the context of Javascript, Rodríguez-Baquero and Linares-Vásquez (2018) propose Mutode, an open-source tool which leverages the npm package ecosystem to perform mutation testing for JavaScript and Node.js applications. The authors empirically evaluated the tool's effectiveness by running it on 12 of the top 20 npm modules that have automated test suites. This proposal is very different from ours: different target of the mutation (npm modules vs web applications) and different kinds of tests considered (test scripts for npm modules vs E2E functional test scripts).

Habibi and Mirian-Hosseinabadi (2015) face the problem of testing Event-Driven Software (EDS), a class of software whose behavior is driven by incoming events. Web and desktop applications that respond to user-initiated events on their Graphical User Interface (GUI) are examples of EDS. Testing EDS poses great challenges to software testers. One of these challenges is the need to generate a huge number of possible event sequences that could sufficiently cover the EDS's state space. In particular, in their work, the authors adopted mutations to create variations of the functional graphs describing the structure of the application under test. This type of mutations is different from the source-based one we use in Mutta, which mimics the standard error of web developers.

The problem of enhancing the set of mutations applicable to the specific type of application has also been investigated for Mobile applications. In fact, mutation testing is considered effective in detecting defects for this type of application as well (Deng et al., 2017). For instance, Moran et al. (2018) proposed MDroid+, an automated framework for mutation testing of Android apps that includes 38 mutation operators from ten empirically derived types of Android faults. The tool has been applied to generate over 8,000 mutants for more than 50 apps.

Petrović et al. (2022) analyzed the problem of applying mutation testing in real industrial scenarios. Indeed, mutation testing has long been considered intractable since the high number of mutants that are usually generated by mutators. Such a number of mutants represent an insurmountable problem, both in terms of human and computational effort. This has hindered the adoption of mutation testing as an industry standard. The authors report the example of Google having a codebase of two billion lines of code, and more than 150 million of tests are executed on a daily basis. In this context, the traditional approach to mutation testing does not scale. Even existing solutions to speed up mutation analysis are insufficient to make it computationally feasible at such a scale. To address these challenges, the authors present a scalable approach to mutation testing based on the following main ideas: (1) mutation testing is performed incrementally, mutating only changed code during code review, rather than the entire code base; (2) mutants are filtered, removing mutants that are likely to be irrelevant to developers, and limiting the number of mutants per line and per code review process; (3) mutants are selected based on the historical performance of mutation operators, further eliminating irrelevant mutants and improving mutant quality. Finally, the authors show that the proposed approach produces orders of magnitude fewer mutants and that context-based mutant filtering and selection improve the results. This work supports our choice of introducing a "selection rule" for the mutants generated by Pitest and suggests a future improvement direction for Mutta, making such selection more advanced and based on more complex criteria.

6 Conclusions and future work

In this paper, we presented a novel tool named Mutta able to automate the entire mutation testing process for web applications. It mutates the various server-side sources of a web application using the state of the art mutation testing system Pitest, runs the E2E test suites against the mutated web applications, and finally collects the test outcomes. We evaluated Mutta with an empirical study involving three different techniques for implementing E2E test suites (and thus three different test suites) and two different applications. This case study helped us understand that (1) Mutta performs the steps necessary to implement the complex process of mutating testing of E2E test suites for web applications, and (2) Mutta can be adopted in practice to evaluate and compare different test suites (e.g., composed by a different set of test scripts) or test suites built with different frameworks (e.g., using assertions or differential testing, as in our case study).

As future work, we plan to extend Mutta in order to support web applications implemented with different languages. Indeed, Mutta currently relies on Pitest, a mutation testing system for Java. The modular architecture of Mutta allows us to replace Pitest with other mutation testing systems able to manage different languages. Moreover, we plan to add a module to automatically support the parallel execution of the test suites against the mutants on the cloud. Although this can be done manually (as shown in the empirical study), we believe it can be considered a useful feature to promote the adoption of Mutta in an industrial context where applications and test suites are large. Finally, we plan to refine the "selection rule" mechanism in order to further reduce the number of mutants considered.

Author contribution Maurizio Leotta, Davide Paparella, and Filippo Ricca contributed equally to the manuscript.

Funding Open access funding provided by Università degli Studi di Genova within the CRUI-CARE Agreement.

Data availability The data of the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors declare no competing interests.

References

Cerioli, M., Leotta, M., & Ricca, F. (2020). What 5 million job advertisements tell us about testing: a preliminary empirical investigation. In *Proceedings of 35th ACM/SIGAPP Symposium on Applied Computing (SAC 2020),* ACM. pp. 1586–1594. https://doi.org/10.1145/3341105.3373961

Coles, H., Laurent, T., Henard, C., Papadakis, M., & Ventresque, A. (2016). Pit: A practical mutation testing tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 449–452.

Deng, L., Offutt, J., & Samudio, D. (2017). Is mutation analysis effective at testing android apps? In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 86–93. https://doi.org/10.1109/QRS.2017.19

García, B., Gallego, M., Gortázar, F., & Munoz-Organero, M. (2020). A survey of the selenium ecosystem. *Electronics*, *9*(7), 1067. https://doi.org/10.3390/electronics9071067

Gulzar, M. A., Zhu, Y., & Han, X. (2019). Perception and practices of differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 71–80. https://doi.org/10.1109/ICSE-SEIP.2019.00016

Habibi, E., & Mirian-Hosseinabadi, S. H. (2015). Event-driven web application testing based on model-based mutation testing. *Information and Software Technology, 67*, 159–179. https://doi.org/10.1016/j.infsof.2015.07.003

Hollweck, T., & Yin, R. K. (2014). Case study research design and methods (5th ed.), *The Canadian Journal of Program Evaluation*, Thousand Oaks, CA: Sage, p. 282. https://doi.org/10.3138/cjpe.30.1.108

Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering, 37*(5), 649–678. https://doi.org/10.1109/TSE.2010.62

Kochhar, P. S., Thung, F., & Lo, D. (2015). Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Proceedings of 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER 2015)*, IEEE, pp. 560–564. https://doi.org/10.1109/SANER.2015.7081877

Laurent, T., Papadakis, M., Kintis, M., Henard, C., Le Traon, Y., & Ventresque, A. (2017). Assessing and improving the mutation testing practice of pit. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST),* IEEE, pp. 430–435.

Leotta, M., Clerissi, D., Ricca, F., Tonella, & P.: Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of 20th Working Conference on Reverse Engineering (WCRE 2013)*, IEEE, pp. 272–281. https://doi.org/10.1109/WCRE.2013.6671302

Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2014). Visual vs. DOM-based web locators: An empirical study. In: Casteleyn, S., Gustavo Rossi, M. W. (eds.), *Proceedings of 14th International Conference on Web Engineering (ICWE 2014)* (vol. 8541), LNCS, Springer, pp. 322–340. https://doi.org/10.1007/978-3-319-08245-5_19

Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2016). Approaches and tools for automated end-to-end web testing. *Advances in Computers, 101*, 193–237. https://doi.org/10.1016/bs.adcom.2015.11.007

Leotta, M., Paparella, D., & Ricca, F. (2022). Comparing the effectiveness of assertions with differential testing in the context of web testing. In: Vallecillo, A., Visser, J., Pérez-Castillo, R. (eds.), *Proceedings of 15th International Conference on the Quality of Information and Communications Technology (QUATIC 2022)* (vol. 1621), CCIS, Springer, pp. 108–124. https://doi.org/10.1007/978-3-031-14179-9_8

Leotta, M., Ricca, F., Stoppa, S., & Marchetto, A. (2022). Is NLP-based test automation cheaper than programmable and capture & replay? In: Vallecillo, A., Visser, J., Pérez-Castillo, R. (eds.), *Proceedings of 15th International Conference on the Quality of Information and Communications Technology (QUATIC 2022)* (vol. 1621), CCIS, Springer, pp. 77–92. https://doi.org/10.1007/978-3-031-14179-9_6

McKeeman, W. M. (1998). Differential testing for software. *Digital Technical Journal, 10*(1), 100–107.

Mirshokraie, S., Mesbah, A., & Pattabiraman, K. (2015). Guided mutation testing for Javascript web applications. *IEEE Transactions on Software Engineering*, *41*(05), 429–444. https://doi.org/10.1109/TSE.2014.2371458

Moran, K., Tufano, M., Bernal-Cárdenas, C., Linares-Vásquez, M., Bavota, G., Vendome, C., Di Penta, M., & Poshyvanyk, D. (2018). Mdroid+: A mutation testing framework for android. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 33–36.

Offutt, A. J., & Untch, R. H. (2001). Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century, Advances in Database Systems* (vol. 24), Springer, pp. 34–44. https://doi.org/10.1007/978-1-4757-5939-6_7

Olianas, D., Leotta, M., & Ricca, F. (2022). SleepReplacer: A novel tool-based approach for replacing thread sleeps in selenium webdriver test code. *Software Quality Journal (SQJ), 30*, 1089–1121. https://doi.org/10.1007/s11219-022-09596-z

Olianas, D., Leotta, M., Ricca, F., Biagiola, M., & Tonella, P. (2021). STILE: A tool for parallel execution of E2E webtest scripts. In *Proceedings of 14th IEEE International Conference on Software Testing, Verification and Validation (ICST 2021)*, IEEE, pp. 460–465. https://doi.org/10.1109/ICST49551.2021.00060

Papadakis, M., Shin, D., Yoo, S., & Bae, D. H. (2018). Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, pp. 537–548.

Petrović, G., Ivanković, M., Fraser, G., & Just, R. (2022). Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering, 48*(10), 3900–3912. https://doi.org/10.1109/TSE.2021.3107634

Praphamontripong, U., & Offutt, J. (2010). Applying mutation testing to web applications. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 132–141. https://doi.org/10.1109/ICSTW.2010.38

Praphamontripong, U., Offutt, J., Deng, L., & Gu, J. (2016). An experimental evaluation of web mutation operators. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 102–111. https://doi.org/10.1109/ICSTW.2016.17

Project, S. (2021). Selenium webdriver documentation. https://www.selenium.dev/documentation/webdriver/

Rodríguez-Baquero, D., & Linares-Vásquez, M. (2018). Mutode: Generic javascript and node.js mutation testing tool. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018, Association for Computing Machinery, New York, NY, USA, pp. 372–375. https://doi.org/10.1145/3213846.3229504

Runeson, P., Host, M., Rainer, A., & Regnell, B. (2012). *Case study research in software engineering: Guidelines and examples (1st edn.)*. Wiley Publishing.

Unadkat, J. (2021). Selenium webdriver tutorial: Getting started with test automation. https://www.browserstack.com/guide/selenium-webdriver-tutorial

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., & Wessln, A. (2012). *Experimentation in Software Engineering*. Incorporated: Springer Publishing Company.

Maurizio Leotta is a Researcher at the University of Genova, Italy. He received his PhD degree in Computer Science from the same university, in 2015, with the thesis "Automated Web Testing: Analysis and Maintenance Effort Reduction". He is author or coauthor of more than 90 research papers published in international journals and conferences/ workshops. His current research interests are in software engineering, with a particular focus on the following themes: Web, Mobile, and IoT application testing, functional test automation, empirical software engineering, business process modelling and model-driven software engineering.

Davide Paparella is a Software Engineer. He received the Master's degree in Computer Science in 2022 with the thesis "Experimental assessment of differential Testing in the context of Web apps" at the University of Genova, Italy. He is already coauthor of a paper published in an international conference.

Filippo Ricca is an Associate Professor at the University of Genova, Italy. He received his PhD degree in Computer Science from the same University, in 2003, with the thesis Analysis, Testing and Re-structuring of Web Applications. In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications". He is author or coauthor of more than 100 research papers published in international journals and conferences/ workshops. Filippo Ricca was Program Chair of CSMR/WCRE 2014, CSMR 2013, ICPC 2011, and WSE 2008. His current research interests include: Software modeling, Reverse engineering, Empirical studies in Software Engineering, Web applications and Software Testing.