Mimicking Production Behavior with Generated Mocks

Deepika Tiwari, Martin Monperrus, and Benoit Baudry
KTH Royal Institute of Technology, Stockholm, Sweden

{deepikat, monperrus, baudry}@kth.se

Abstract—Mocking allows testing program units in isolation. A developer who writes tests with mocks faces two challenges: design realistic interactions between a unit and its environment; and understand the expected impact of these interactions on the behavior of the unit. In this paper, we propose to monitor an application in production to generate tests that mimic realistic execution scenarios through mocks. Our approach operates in three phases. First, we instrument a set of target methods for which we want to generate tests, as well as the methods that they invoke, which we refer to as mockable method calls. Second, in production, we collect data about the context in which target methods are invoked, as well as the parameters and the returned value for each mockable method call. Third, offline, we analyze the production data to generate test cases with realistic inputs and mock interactions. The approach is automated and implemented in an open-source tool called RICK. We evaluate our approach with three real-world, open-source Java applications. RICK monitors the invocation of 128 methods in production across the three applications and captures their behavior. Based on this captured data, RICK generates test cases that include realistic initial states and test inputs, as well as mocks and stubs. All the generated test cases are executable, and 52.4% of them successfully mimic the complete execution context of the target methods observed in production. The mock-based oracles are also effective at detecting regressions within the target methods, complementing each other in their fault-finding ability. We interview 5 developers from the industry who confirm the relevance of using production observations to design mocks and stubs. Our experimental findings clearly demonstrate the feasibility and added value of generating mocks from production interactions.

Index Terms—Mocks, production monitoring, mock-based oracles, test generation

1 Introduction

Software testing is an expensive, yet indispensable, activity. It is done to verify that the system as a whole, as well as the individual modules that compose it, behave as expected. The latter is achieved through unit testing. When a unit interacts with others, or with external components, such as the file system, database, or the network, it becomes challenging to test it in isolation. As a solution to this problem, developers rely on a technique called mocking [1]. Mocking allows a unit to be tested on its own, by replacing dependent objects that are irrelevant to its functionality, with "fake" implementations. There are several advantages of mocking, such as faster test executions, fewer side-effects, and quicker fault localization [2].

Despite their advantages, using mocks within tests requires considerable engineering effort. Developers must first identify the components that may be replaced with mocks [3]. They must also determine how these mocks behave when triggered with a certain input, i.e., how they are stubbed [4], [5]. In order to address these challenges, several techniques have been developed to automatically generate mocks. For example, mocks have been generated through search-based algorithms by including contextual information, such as the state of the environment, in the search space for input data generation [6], [7]. Dynamic symbolic execution has been used to define the behavior of mocks through generated mock classes [8]. System test executions can be monitored to derive unit tests that use mocks [9]. This preliminary research has validated the concept of mock generation. However, the test inputs and the mocks produced by these techniques are either synthetic or manually written by developers per their assumptions of how the system behaves. These approaches do not guarantee that the generated mocks reflect realistic behaviors as observed in production contexts.

The fundamental premise of mocking is to replace a real object with a fake one that mimics it [10], [11]. This implies that, for it to be useful, the behavior of the mock should resemble, as closely as possible, that of a real object [12]. Our key insight is to derive realistic behavior from real behavior, i.e., to generate mocks from production usage. This builds upon the fact that 1) the behavior of an application in production is the reference one [13], and 2) this production behavior can be used for test generation [14], [15]. Given a set of methods of interest for test generation, we monitor them in production. As a result of this monitoring, we capture realistic execution contexts to generate tests for each target method, where external objects are replaced with mocks, and stubbed based on production states.

In this paper, we propose RICK, an approach for mimicking production behavior with generated mocks. RICK monitors an application executing in production with the goal of generating valuable tests. The intention of the generated tests is to verify the behavior of the methods, where the reference behavior captured in the oracle is the one from production. The interactions of this method with other external objects are mocked. Within each generated test, the data captured from production is expressed as rich serialized test inputs. Each generated test has a mock-based oracle, which verifies distinct aspects of the invocation of the target method and its interactions with mock methods, such as the object returned from the target method, the parameters with which the mock methods are called, as well as the frequency and sequence of these mock method calls.

Our approach for the synthesis of mocks is based on two fundamentally novel concepts. First, the mocks are stubbed with data and interactions observed in production. Second, our three mock-based oracles enable powerful behavior verification. A key benefit of this latter point is that, in addition to checking the output of a method directly with a straightforward assertion, we also verify the actions that should occur within the method.

We evaluate the capabilities of RICK with three open-source Java applications: a map-based routing application called GRAPHHOPPER, a feature-rich graph analysis and visualization tool called GEPHI, and a utility library for working with PDF documents called PDFBOX. We target 212 methods across the three applications, which get invoked as the applications execute in typical production scenarios. RICK generates 294 tests for 128 of these methods. Within each generated test, RICK recreates execution states that mimic production ones with objects that range in size from 37 bytes to 39 megabytes. When we execute the tests, we find that 68 of the 128 methods (53.1%) have at least one passing test that recreates real usage conditions, and 154 of the 294 (52.4%) tests successfully recreate the complete production execution context. These results indicate that RICK is capable of monitoring applications in production, capturing realistic behavior for target methods, and transforming it into tests that mimic the behavior of the methods, while isolating it from its interactions with external objects. Furthermore, through mutation analysis, we determine that the generated tests are effective at detecting regressions within the target methods. The mock-based oracles contained in the generated tests complement each other with respect to their ability to detect bugs. To assess the quality of the RICK-generated tests, we interview 5 software developers from different sectors of the IT industry. All of them find the collection of production values to be relevant to generate realistic mocks. Moreover, they appreciate the structure and the understandability of the tests generated by RICK.

To sum up, the key contributions of this paper are:

- The novel concept of the automated generation of mocks, stubs, and oracles using data collected from production.
- A comprehensive methodology for generating tests that mimic complex production interactions through mocks, by capturing receiving objects, parameters, returned values, and method invocations, for a method under test, while an application executes.
- An evaluation of the approach on 3 widely-used, large, open-source Java applications, demonstrating the feasibility and benefits of generating mocks.
- A publicly available tool called RICK implementing the approach for Java, and a replication package for future research on this novel topic[1].

1. https://zenodo.org/doi/10.5281/zenodo.6914463

The rest of the paper is organized as follows. section 2 presents the background on mocking and mock objects. We describe how RICK generates tests and mocks in section 3. Next, section 4 discusses the methodology we follow for our experiments, applying RICK to real-world Java projects, followed by the results of these experiments in section 5. section 7 presents closely related work, and section 8 concludes the paper.

2 BACKGROUND

This section summarizes the key concepts of mock objects, and how they are used in practice. We also discuss the challenges of using mocks within tests.

2.1 Mock Objects

Software comprises of individual modules or units. These units interact with each other, as well as with external libraries, for example, to send emails, transfer data over the network, or perform database operations. This facilitates modular development, as different teams can work in parallel on implementing distinct functionalities of the system. The modules are then composed together, in order to achieve use cases. Yet, a disadvantage of this coupling is that testing each unit in isolation from others is not straightforward. *Mocking* was proposed as a solution to this problem [1]. It is a mechanism to replace real objects with skeletal implementations that mimic them [2]. Mocking allows individual functionalities to be tested independently. The process of unit testing with mocks is faster and more focused [3]. Since the test intention is to verify the behavior of one individual unit, mocking can facilitate fault localization. External objects, with interactions that are complex to set up within a test, can be replaced with mocks [16]. Furthermore, a test can be made more reliable by using mocks to replace external, potentially unreliable or non-deterministic, components [6], [7]. Mock objects typically behave in specific, pre-determined ways through a process called *stubbing* [5], [10]. For example a method called `getAnswer` invoked on a mock object can be stubbed to return a value of `42`, without the actual invocation of `getAnswer`. Stubbing can be very useful for inducing behavior that may be hard to produce locally within the test, such as error- or corner-cases. Mocking can also be used for *verifying* object protocols [17]. For example, consider a method called `subscribeToNewsletter`, which should call another method `sendWelcomeEmail` on an object of type `EmailService`. Developers can mock the `EmailService` object within the test for `subscribeToNewsletter`, to verify that the method `sendWelcomeEmail` is indeed invoked on `EmailService` exactly once, with a parameter of type `UserID`. This interaction is therefore verifiable without side-effects, i.e., without an actual email being sent.

In short, the three key concepts of testing with mocks are *Mocking*, *Stubbing*, and *Verifying*. Real objects can be replaced with fake implementations called mocks. These mocks can be stubbed to define tailor-made behaviors, i.e., produce a certain output for a given input. Moreover, the interactions made with the mocks can be verified, such as the number of times they were triggered with a given input, and in a specific sequence.

```
1 public class ReservationCentre {
2   ...
3   // Target method
4   public void purchaseTickets(int quantity, PaymentService
        paymentService) {
5     ...
6     double amount = basePrice * quantity;
7     ...
8     // Mockable method call #1
9     if (paymentService.checkActiveConnections() > 0) {
10      ...
11      // Mockable method call #2
12      boolean isPaymentSuccessful =
          paymentService.processPayment(amount);
13      ...
14    }
15    ...
16    return ...;
17  }
18  ...
19 }
```

Listing 1: Target method `purchaseTickets` has mockable method calls on the `PaymentService` object

```
1 @Test
2 public void testTicketPurchasing() {
3   ReservationCentre resCentre = new ReservationCentre();

5   // Mock external types
6   PaymentService mockPayService = mock(PaymentService.class);

8   // Stub behavior
9   when(mockPayService.checkActiveConnections()).thenReturn(1);
10  when(mockPayService.processPayment(42.24)).thenReturn(true);

12  resCentre.purchaseTickets(2, mockPayService);

14  // Verify invocations on mocks
15  verify(mockPayService, times(1)).checkActiveConnections();
16  verify(mockPayService, times(1)).processPayment(anyDouble());
17 }
```

Listing 2: A test for the `purchaseTickets` method which mocks `PaymentService`

2.2 The Practice of Testing with Mocks

Mocks can be implemented in several ways. For example, developers may manually write classes that are intended as replacements for real implementations [18]. However, a more common way of defining and using mocks is through the use of mocking libraries, which are available for most programming languages. Mockito[2] is one of the most popular mocking frameworks for Java, both in the industry and in software engineering research [19], [20]. It can be integrated with testing frameworks such as JUnit and TestNG, allowing developers to write tests that use mocks.

Let us consider the example of the method `purchaseTickets` presented in Listing 1. This method handles the purchase of tickets, including the interaction with the payment gateway, `PaymentService`. It is defined in the `ReservationCentre` class, and takes two parameters. The first parameter is an integer value for the `quantity` of tickets, and the second parameter is the object `paymentService` of the external type `PaymentService`. Two methods are called on the `paymentService` object: `checkActiveConnections` on line 9, and `processPayment` on line 12 which accepts a parameter of type `double`. We illustrate the use of mocks, stubs, and verification through the unit test for `purchaseTickets` presented in Listing 2. The intention of this test, `testTicketPurchasing`, is to verify the behavior of `purchaseTickets`, while mocking its interactions with `PaymentService`. First, the receiving object `resCentre` of type `ReservationCentre` is set up (line 3). Next, `PaymentService` is mocked, through the `mockPayService` object (line 6). Lines 9 and 10 stub the two methods called on this mock: `checkActiveConnections` is stubbed to return a value of `1`, and `processPayment` is stubbed to return `true` when invoked with the `double` value `42.24`. Finally, on line 12, `purchaseTickets` is called with the `quantity` of 2, and the mocked parameter `mockPayService`. The statements on lines 15 and 16 verify that this invocation of `purchaseTickets` calls `checkActiveConnections` exactly once, and `processPayment` exactly once, with a double value (specified using `anyDouble()`). Thus, this test verifies the behavior of the target method, `purchaseTickets`, isolating it from the interactions with a real `PaymentService` object. Moreover, method calls on `PaymentService` are stubbed so that `purchaseTickets` gets executed as it normally would, without the side-effect of an actual payment being made. This allows for more focus on the method under test, `purchaseTickets`.

2. https://site.mockito.org/

2.3 The Challenges of Mocking

Despite the benefits of mocking that we highlight, it is not trivial to incorporate mocks in practice. Deciding what to mock, and how the mocks should behave, is hard [11]. For example, developers would first identify that `PaymentService` may be mocked within the test for `purchaseTickets` in Listing 2. Next, they must also manually define concrete values for the parameters and returned values for stubbing the calls made on this mock, in order trigger a specific path through `purchaseTickets`. Additionally, they would have to determine which interactions made on this mock are verifiable. It is also challenging to decide between conventional object-based testing and mock-based testing. Because of these challenges, developers are hesitant to incorporate mocks within their testing practice, as highlighted by a study by Spadini *et al.* [19], who found that mocks are most likely to be introduced at the time a test class is first written. This suggests the potential opportunity and benefits of automated mock generation throughout the development lifecycle. The results from our developer study in RQ5 address this aspect.

To address the challenges of manually implementing mocks, several studies propose methodologies for their automated generation, such as through search-based algorithms [21] or symbolic execution [22]. However, none of these studies use data from production executions to do so. In this work, we propose to monitor an application in production, with the goal of generating tests with mocks. These tests use mocks to 1) isolate a target method from external units, and 2) verify distinct aspects of the behavior with oracles specific to mocks.

3 Mock Generation with RICK

We introduce RICK, a novel approach for automatically generating tests with mock objects, using data collected during the execution of an application. In production, RICK collects realistic data for recreating the program states for the method under test, as well as the parameters and values returned by methods called on external objects. In subsection 3.1, we present an overview of the RICK pipeline. This is followed in subsection 3.2 by a discussion of the kinds of oracles produced by RICK. Next, subsection 3.3 motivates the design decisions of RICK and highlights its key features. We discuss in subsection 3.4 how RICK can be useful in the software development lifecycle. Finally, subsection 3.5 presents technical details of its implementation.

3.1 Overview

RICK operates in three phases. We illustrate them in Figure 1. In the first phase (1), RICK identifies test generation targets within an application. These targets are called methods under test, and they have mockable method calls. We define them as follows:

Methods under Test (MUTs): The target methods for test generation by RICK are called methods under test (MUTs). A method is considered as being an MUT if it invokes methods on objects of other types. The identification of MUTs forms the basis of the test generation effort, since the intention of each test generated by RICK is to verify the behavior of one MUT after isolating it from such external interactions.

Mockable method calls: We define a mockable method call as a method call nested within an MUT, that is made on a field or a parameter object whose type is different from the declaring type of the MUT. RICK will mock objects of types that are declared within the project, and not types from the standard library or dependencies. When RICK generates a test for the MUT, a mockable method call becomes a *mock method call*, i.e., the external object is replaced with a mock object, and the mockable method call occurs on this mock object.

Figure 1 illustrates the identification of an MUT (highlighted in yellow), together with its mockable method calls (shown here in green circles). In subsection 3.3 we detail how a nested method call qualifies as being mockable. As an example, consider the class `ClassUnderTest` presented in Listing 3. RICK identifies the method `methodUnderTestOne` (line 5) as an MUT. Moreover, the nested call to `mockableMethodOne` on line 8 within `methodUnderTestOne` is identified as a mockable method call because it is made on the field `extField` of `ClassUnderTest` (line 2), which is of an external type. Similarly, the method `methodUnderTestTwo` is also considered as an MUT by RICK, because it has two mockable methods called within it. The first is `mockableMethodTwo` called 42 times inside a loop (line 20), and the second one is `mockableMethodThree` (line 22). Both of these methods are called on `extParam`, which is an external parameter of `methodUnderTestTwo`.

The second phase (2) of RICK occurs when the application is deployed and running. During this phase, RICK monitors the invocation of the MUTs identified in the previous phase, and collects data corresponding to these invocations. By construction, the monitoring data reflects real interactions by end-users. Moreover, for inadequately-tested applications, it may represent usage scenarios that are not well tested by developer-written tests [13], [15].

RICK collects data for an invocation in production with the end goal of recreating the same invocation within a generated test. This data includes the parameters and returned values for each MUT and its corresponding mockable method calls, as well as the object on which the MUT is invoked, which we refer to as the *receiving object*. Figure 1 depicts the second phase, where the monitor defined within RICK is attached to both an MUT as well as its mockable method calls, in order to collect data about their invocations. subsection 3.3 presents more details on this monitoring.

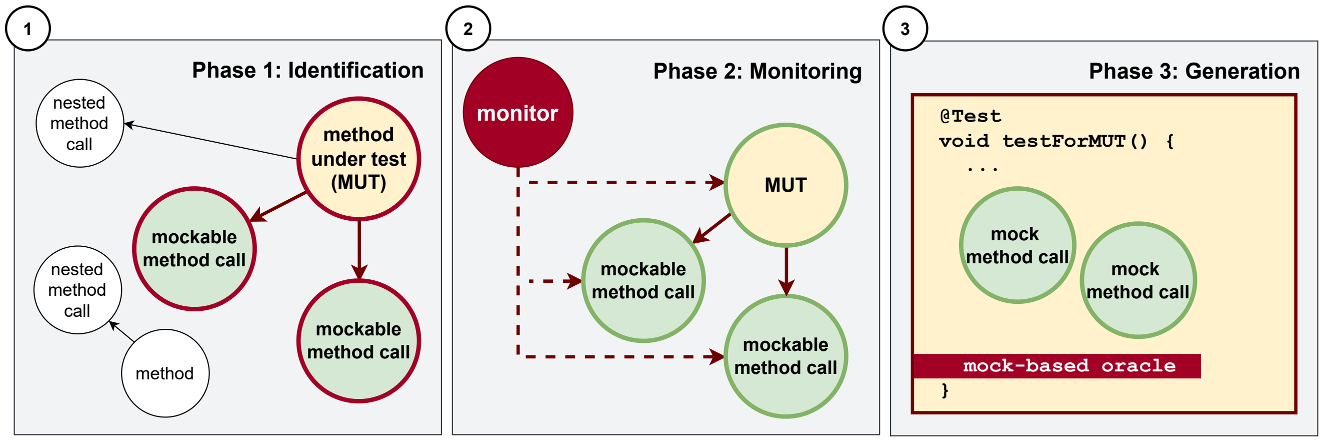
Finally, in the third phase (3), RICK uses the data collected in the second phase as inputs to generate tests with mocks, as illustrated in Figure 1. These tests are designed to recreate the invocation of the MUTs, and verify their behavior as was observed in production, while simulating the interactions of the MUTs with external objects using mocks and stubs. They can serve as regression tests, and can also potentially lead to faster fault localization because they isolate the invocation of the MUT from the mockable method calls. For example, Listing 4 presents one test generated for the MUT `methodUnderTestOne`. This test verifies the observed behavior of `methodUnderTestOne` through the assertion on line 17, while mocking the `ExternalTypeOne` object (line 8). Within the test, the mockable method `mockableMethodOne` becomes the mock method, and is stubbed on line 11 using its parameter and returned value captured from production. We present more information on how RICK processes production data to generate tests in subsection 3.3. The generated tests are the final output of the RICK pipeline. Each generated test falls under one of three categories, determined by the kind of oracle within it. We discuss these categories in subsection 3.2.

3.2 Mock-based Oracles

The oracle in a unit test specifies a behavior that is expected as a consequence of running the MUT with a specific test input [4]. In the context of the tests generated by RICK, the oracles in the generated test verify the behavior of the MUT, while isolating it from method calls to external objects made within the MUT, i.e., mockable method calls. This facilitates the decoupling of the MUT from its environment, and allows the focus of the testing to be on the MUT itself. Moreover, the behavior being verified in the generated tests, both for the MUT and the mockable method calls, is sourced from production. This means that through these generated tests, developers can verify how the system behaves for actual users.

There is no systematized knowledge on oracles for tests with mocks. To overcome this, we now define three categories of oracles, all implemented by RICK.

Output Oracle, OO: The first category of tests generated by RICK have an output oracle. This oracle verifies that the behavior of the MUT is the same as the one observed in production, despite the introduction of mock objects. Even though this oracle relies on regular assertions, we still consider it as a mock-based oracle, as it assesses the behavior of the MUT in the presence of mock objects. A failure in a test with an output oracle indicates a regression in the MUT, which may be caused by its interaction with the mockable method call. Listing 4 presents an example



```
@Test
void testForMUT() {
  ...
}
```

Fig. 1: The RICK test generation pipeline: offline, identify methods under test and mockable method calls; in production, monitor state and arguments for methods under test and mockable method calls; offline again, generate tests with mocks for the methods under test

```java
class ClassUnderTest {
  ExternalTypeOne extField;
  ...

  public int methodUnderTestOne(int param) {
    ...
    // mockable method call on field
    int x = param + extField.mockableMethodOne(booleanVal);
    // nested, non-mockable method call
    int y = x + methodFour();
    ...
    return ...;
  }

  public int methodUnderTestTwo(double param,
                              ExternalTypeTwo extParam) {
    ...
    // mockable method calls on parameter
    for (int i = 0; i < 42; i++) {
      listOfInts.add(extParam.mockableMethodTwo(floatVals[i]));
    }
    int z = extParam.mockableMethodThree(intVal);
    ...
    return ...;
  }

  private void methodThree() { ... }

  public int methodFour() {
    // nested, non-mockable method call
    methodThree();
    ...
    return ...;
  }
}
```

Listing 3: The class, `ClassUnderTest`, has four methods. RICK identifies two of these methods, `methodUnderTestOne` and `methodUnderTestTwo`, as MUTs as they have mockable method calls.

```java
1 @Test
2 public void testMethodUnderTestOne_OO() {
3 // Arrange
4   // Create test fixture from serialized production data
5   ClassUnderTest productionObj = deserialize(new File(
        "receiving1.xml"));

7   // Inject the mock
8   ExternalTypeOne mockExternalTypeOne =
        injectMockField_extField_InClassUnderTest();

10  // Stub the behavior
11  when(mockExternalTypeOne.mockableMethodOne(true))
        .thenReturn(27);

13 // Act
14  int actual = productionObj.methodUnderTestOne(17);

16 // Assert
17  assertEquals(42, actual);
18 }
```

Listing 4: A RICK test with an Output Oracle, **OO**, for the MUT `methodUnderTestOne`

of a generated test with an output oracle. The MUT is `methodUnderTestOne`, and the mockable method call is `mockableMethodOne`. This test corresponds to one invocation of `methodUnderTestOne` observed by RICK in production. The test recreates the receiving object, `productionObj`, as it was observed in production, by deserializing it (line 5). Next, it mocks the field `extField` and injects it into `productionObj` (line 8). This is followed in line 11 by stubbing `mockableMethodOne`, to return a value of `27` when invoked with the parameter `true`, in accordance with the observed production behavior of `mockableMethodOne`. On line 14, `methodUnderTestOne` is invoked on `productionObj` with the production parameter `17`. Finally, the output oracle is the assertion on line 17, which verifies that the output from this invocation of `methodUnderTestOne`, with the stubbed call to `mockableMethodOne`, is `42`, which is the value observed for this invocation in production.

Parameter Oracle, PO: The second category of generated tests have an oracle to verify that the mockable method calls occur with specific parameter(s), the same as production, within the invocation of the MUT. A test with a parameter oracle may fail due to regressions in the MUT which cause a mockable method call to be made with unexpected parameters. An example of this oracle is presented in Listing 5. This test recreates the receiving object `productionObj`, for the MUT `methodUnderTestTwo` (line 5). Next, it prepares a mock object for `ExternalTypeTwo` called `mockExternalTypeTwo` (line 8), and stubs the 42 invocations of the mockable method call, `mockableMethodTwo`. For brevity, we include only two of these stubs on lines 11 and 12. The single invocation of `mockableMethodThree`

```
1 @Test
2 public void testMethodUnderTestTwo_PO() {
3 // Arrange
4   // Create test fixture from serialized production data
5   ClassUnderTest productionObj = deserialize(new File(
          "receiving2.xml"));

7   // Create the mock
8   ExternalTypeTwo mockExternalTypeTwo = mock(
          ExternalTypeTwo.class);

10  // Stub the behavior
11  when(mockExternalTypeTwo.mockableMethodTwo(4.2F))
          .thenReturn(89);
12  when(mockExternalTypeTwo.mockableMethodTwo(9.8F))
          .thenReturn(92);
13  ...
14  when(mockExternalTypeTwo.mockableMethodThree(15))
          .thenReturn(48);

16 // Act
17  productionObj.methodUnderTestTwo(6.2, mockExternalTypeTwo);

19 // Assert
20  verify(mockExternalTypeTwo, atLeastOnce())
          .mockableMethodTwo(4.2F);
21  verify(mockExternalTypeTwo, atLeastOnce())
          .mockableMethodTwo(9.8F);
22  ...
23  verify(mockExternalTypeTwo, atLeastOnce())
          .mockableMethodThree(15);
24 }
```

Listing 5: A RICK test with a Parameter Oracle, **PO**, for the MUT `methodUnderTestTwo`

(line 14) is also stubbed, with the parameter and returned value observed in production. This is followed by a call to `methodUnderTestTwo` on `productionObj` (line 17), passing it `mockExternalTypeTwo` as parameter. Finally, the statements on lines 20 to 23 are unique to this category of tests, and serve as the parameter oracle. The statements on lines 21 and 22 verify that `mockableMethodTwo` is called at least once on `mockExternalTypeTwo` with the concrete production parameter `4.2F`, as well as with `9.8F`. We omit the verification of the other 40 invocations of `mockExternalTypeTwo` from this code snippet. Next, the parameter oracle verifies on line 23 that `mockableMethodThree` is called at least once with the parameter `15`, within this invocation of `methodUnderTestTwo`.

Call Oracle, CO: Oracles in the generated tests for the third category verify the sequence and frequency of mockable method calls within the invocation of the MUT. Any deviation from the expected sequence and frequency of mockable method calls within an MUT will cause a test with a call oracle to fail. This can be helpful for developers when localizing a regression related to object protocols within the MUT. An example of this oracle is presented in Listing 6. This test first recreates the receiving object, `productionObj`, for the MUT `methodUnderTestTwo` (line 5), stubs the mockable method calls to `mockableMethodTwo` and `mockableMethodThree` (lines 11 to 14), and invokes `methodUnderTestTwo` with the mocked parameter (line 17). Next, the call oracle in this test verifies the number of times the mockable method calls occur within this invocation of `methodUnderTestTwo`, as well as the order in which these calls occur. This is achieved with the order verifier defined on line 20. The statements on lines 21 and 22 verify that `mockableMethodTwo` is invoked exactly 42 times with a float

```
1 @Test
2 public void testMethodUnderTestTwo_CO() {
3 // Arrange
4   // Create test fixture from serialized production data
5   ClassUnderTest productionObj = deserialize(new File(
          "receiving2.xml"));

7   // Create the mock
8   ExternalTypeTwo mockExternalTypeTwo = mock(
          ExternalTypeTwo.class);

10  // Stub the behavior
11  when(mockExternalTypeTwo.mockableMethodTwo(4.2F))
          .thenReturn(89);
12  when(mockExternalTypeTwo.mockableMethodTwo(9.8F))
          .thenReturn(92);
13  ...
14  when(mockExternalTypeTwo.mockableMethodThree(15))
          .thenReturn(48);

16 // Act
17  productionObj.methodUnderTestTwo(6.2, mockExternalTypeTwo);

19 // Assert
20  InOrder orderVerifier = inOrder(mockExternalTypeTwo);
21  orderVerifier.verify(mockExternalTypeTwo, times(42))
          .mockableMethodTwo(anyFloat());
22  orderVerifier.verify(mockExternalTypeTwo, times(1))
          .mockableMethodTwo(anyInt());

24 }
```

Listing 6: A RICK test with a Call Oracle, **CO**, for the MUT `methodUnderTestTwo`

parameter, and that these invocations are followed by one call to `mockableMethodThree` with an integer parameter, as was observed in production.

3.3 Key Phases

As outlined in subsection 3.1, RICK operates in three phases. We now discuss these three phases in more detail.

3.3.1 *Identification of Test Generation Targets*

It is not possible to generate test cases with mocks for all methods with nested method calls. For example, a static method invoked within another method is typically not mocked [3]. Also, it is not feasible to replace an object created within the body of a method, and subsequently mock the interactions made with it. Therefore, RICK includes a set of rules to determine the methods that can be valid targets for the generation of test cases and mocks. It is also possible for developers to provide an initial set of methods of interest, which RICK can consider.

3.3.1.1 Identifying MUTs: First, RICK finds methods that are part of the API of the application, i.e., methods that are public, non-abstract, non-deprecated, and non-empty [23], [24]. These criteria have also been used previously to generate differential unit tests for open-source Java projects [15]. Of these methods, RICK selects as MUTs the ones that invoke other methods on objects of external types.

3.3.1.2 Identifying Mockable Method Calls: Second, RICK identifies the nested method calls within each of the selected MUTs which could be mocked. An MUT may have several nested method calls, not all of which are suitable for mocking. For it to be mocked, a nested method call must be invoked on a parameter or a field, such that a mock can be injected to substitute it in the generated test. Next, the declaring type of this parameter or field should be different from the type of the MUT, in order to represent an interaction of the MUT with an external type, per the theory of mocking external resources. RICK stubs methods that return a primitive or `String` value. Mocks are never returned from stubbed methods. Nested method calls that meet all these criteria, are marked as a *mockable method calls*.

We illustrate the rules for target selection with the help of the excerpt of the class `ClassUnderTest` in Listing 3. This excerpt includes a field as well as four methods defined in `ClassUnderTest`. The first method, `methodUnderTestOne` (lines 5 to 13), accepts an integer parameter, and returns an integer value. The body of `methodUnderTestOne` includes a call to the method `mockableMethodOne(boolean)`, on the field `extField` (line 8). This field is declared as being of type `ExternalTypeOne` in `ClassUnderTest` (line 2). There is another call on line 10 to a method defined in `ClassUnderTest` called `methodFour`. The second method, `methodUnderTestTwo` (lines 15 to 25), returns an integer value, and accepts two parameters. The first parameter is a double, and the second parameter called `extParam` is of type `ExternalTypeTwo`. Within the loop on lines 19 to 21, `methodUnderTestTwo` calls the method `mockableMethodTwo(float)` on the parameter `extParam` (line 20). There is another call on `extParam` to `mockableMethodThree(int)` (line 22). The third method defined in `ClassUnderTest` is a private method called `methodThree` (line 27). It does not call any other method. Finally, the fourth method in this excerpt is `methodFour` (lines 29 to 34), which has a call to `methodThree` (line 31).

As a consequence of the aforementioned criteria, RICK identifies `methodUnderTestOne` and `methodUnderTestTwo` as MUTs. Moreover, the nested method calls, `mockableMethodOne`, and `mockableMethodTwo` and `mockableMethodThree`, in these MUTs respectively, are recognized as mockable method calls by RICK. However, the call to `methodFour` within `methodUnderTestOne` is not mockable within `methodUnderTestOne`, and `methodThree` and `methodFour` are not MUTs since they do not fulfill these criteria.

3.3.2 *Monitoring Test Generation Targets*

Once it finds a set of MUTs and their corresponding mockable method calls, RICK instruments them in order to monitor their execution as the application runs in production. The goal of this instrumentation is to collect data as the application executes. RICK collects data about each invocation of an MUT: the receiving object, which is the object on which it is invoked, the parameters passed to the invocation, as well as the object returned from the invocation. At the same time, RICK collects data about the mockable method calls within this MUT. This includes the parameters and the returned value for each mockable method call. The data collected from this monitoring is serialized and saved to disk. For example, the sequence diagram in Figure 2 illustrates the monitoring of the MUT `methodUnderTestTwo` in `ClassUnderTest` presented in Listing 3. For one invocation of `methodUnderTestTwo`, RICK collects its receiving object, parameters, and returned value, as well as the parameters and returned values corresponding to the invocations of mockable method calls to `mockableMethodTwo` and `mockableMethodThree` within `methodUnderTestTwo`.

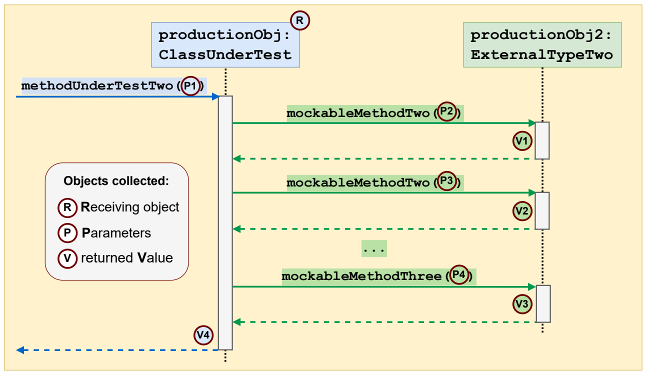


Fig. 2: Monitoring method invocations in production: RICK observes the invocation of the target method, `methodUnderTestTwo`, and captures the receiving object and parameters for this invocation, as well as the object returned from it. RICK also observes the method calls to `mockableMethodTwo` and `mockableMethodThree`, collecting their parameters and returned values.

We systematically consider special cases. First, a mockable method may be invoked from different MUTs. Also, an MUT may itself be a mockable method for another MUT. Moreover, an MUT may be invoked without its corresponding mockable method call(s), if the latter is invoked within a branch, for example. It is therefore important to ensure that the data collected for a mockable method call is associated with a specific invocation of an MUT. RICK implements this association by assigning a unique identifier to each MUT invocation, and the same identifier to each mockable method call within it. This information is required for the generation of all of the three kinds of oracles, i.e., the output oracle, the parameter oracle, as well as the call oracle. Second, one invocation of an MUT may have multiple mockable method calls, which may or may not have the same signature. Furthermore, these invocations occur in a specific order within the MUT. In order to account for this, RICK collects the timestamps for each mockable method call. This is done to synthesize statements corresponding to the call oracle, which verify the sequence and frequency of mockable method calls in the generated tests.

3.3.3 *Generation of Tests with Mocks*

Once RICK has collected data about invocations of MUTs and corresponding mockable method calls, the final phase can begin, triggered by the developer. RICK connects an MUT invocation with mockable method calls by utilizing the unique identifiers assigned to each invocation observed in production. It then generates code to produce the three categories of tests for each invocation, as detailed in subsection 3.2. The final output from the test generation phase is a set of test classes, which include tests from the three categories, for each invocation of an MUT that was observed by RICK in production.

RICK generates tests by bringing together all the data it has observed, collected, and linked to the respective invocation of an MUT and its mockable method calls. Within each test generated by RICK for an MUT:

- The serialized receiving object and parameter(s) for the MUT are deserialized to recreate their production state. For example, the receiving object of the respective MUT is recreated from its serialized XML state on line 5 of Listing 4, Listing 5, and Listing 6.
- External objects, on which mockable method calls occur, are substituted with mock objects. For example, a mock object substitutes the external field `extField` on line 8 of Listing 4. A mock object is prepared for the parameter `ExternalTypeTwo` on line 8 of Listing 5 and Listing 6.
- Mockable method calls become mock method calls: they are stubbed, with production parameter(s) and returned value. The mock method call within `methodUnderTestOne` is stubbed on line 11 of Listing 4. The mock method calls within `methodUnderTestTwo` are stubbed from lines 11 to 14 in Listing 5 and Listing 6.
- The generated test case calls the method under test, once. This makes the test intention very clear: the behavior of the MUT is the one which will be assessed by the oracle. Note that multiple methods may be called on the mock objects, all stubbed. The number of mock objects and stubbed methods is what creates a large testing space.
- The oracle verifies a unique aspect about the invocation of the MUT and its interactions with the external object(s): the **OO** on line 17 of Listing 4 verifies the output of `methodUnderTestOne`, the **PO** on lines 20 to 23 of Listing 5 verify that the mock method calls occur with the same parameters as they did in production, and the **CO** on lines 21 and 22 of Listing 6 verify that the mock method calls happen in the same sequence and the same frequency as they did in production.

3.4 RICK in the Software Development Lifecycle

There are two main use cases for RICK in the software development pipeline. First, for a project that has few automated tests, or uses only manual testing [25], [26], RICK can be used to bootstrap the creation of a test suite. The setup would be as follows: human QA testers are employed to evaluate and manually test the system as a whole. Meanwhile, RICK would capture the realistic interactions that the testers trigger, and would generate automated unit test cases, which can be frequently run when the developers evolve the application.

Second, for projects that already contain automated tests, RICK can contribute with unit tests that reflect realistic behavior, as observed in production. Field executions can be a rich source of data, and are likely to include usage scenarios not envisioned by developers [13]. Leveraging the monitoring and automated test generation capabilities of RICK would allow these behaviors to be incorporated into the test suite. New tests based on production observations have been shown to complement developer-written tests and improve the effectiveness of the test suite [15], [27].

A key phase in both use cases is the curation of a set of essential methods of interest, which will be the targets for test generation with RICK. Though RICK ships with good default filters for identifying target methods, developers can use their domain expertise to define the most valuable target methods within their project. Ideal candidates for test generation include methods that have recently been added or modified, methods at the public interface of the application, or methods that do not meet a specified test adequacy criterion.

The test cases generated by RICK fully depend on the production usages that were monitored. To that extent, RICK is not good at generating tests for corner cases that rarely occur. On the other hand, RICK is excellent at generating tests for mission-critical functionalities that reflect typical usage scenarios for a target application.

Finally, the tests generated by RICK can be fully integrated in a code review process. We envision that the lead test engineer handle the test generation and open a pull-request to add the new tests. Then, fellow developers would review the tests generated for each target method before merging them into the test suite for the project. This process can be repeated multiple times, one target method at a time, throughout the lifecycle of the project.

3.5 Implementation

RICK is implemented in Java. MUTs and their corresponding mockable method calls are identified through static analysis with Spoon [28]. Once identified, they are instrumented and monitored in production with Glowroot, an open-source Application Production Monitoring agent[3]. Glowroot is a well-documented, industry-grade tool. It has a low overhead and is stable. This makes it the best fit for monitoring production executions for mock generation. Data collection from production is handled through serialization by XStream[4]. RICK relies on the code generation capabilities of Spoon[5] to produce JUnit tests[6]. These tests define and use Mockito[7] mocks, specifically the `mockito-inline` flavour, which allows mocking final classes. By default, RICK generates three separate tests that contain the parameter oracle the call oracle, and the output oracle. If the MUT does not return a primitive or a `String`, RICK generates only two tests with the parameter and the call oracles. RICK uses some capabilities provided by the PANKTI framework [15].

4 Experimental Methodology

This section introduces our experimental methodology. We describe the open-source projects we use as case studies to evaluate mock generation with RICK. Then, we describe the production conditions that we use to collect data for test generation. Next, we present our research questions and define the protocol that we use to answer them.

4.1 Case Studies

As detailed in section 3, RICK uses data collected from an application in production, in order to generate tests with mock-based oracles. For this evaluation, we therefore target applications that we can build, deploy, and for which we

3. https://glowroot.org/
4. https://x-stream.github.io/
5. https://spoon.gforge.inria.fr/
6. https://junit.org/
7. https://site.mockito.org/

TABLE 1: Case studies for the evaluation of RICK

METRIC	GRAPHHOPPER	GEPHI	PDFBOX
VERSION	5.3	0.9.6	2.0.24
SHA	af5ac0b	ea3b28f	8876e8e
STARS	3.7K	4.9K	1.7K
COMMITS	5.8K	6.5K	8.7K
LOC	89K	35K	165K
METHODS	4,104	2,117	9,102
CANDIDATE_MUTS	356	115	319

can define a production-grade usage scenario. We manually search for three notable, open-source Java projects that satisfy these criteria. We also make sure that the case studies represent different categories of software: a library with a command-line interface, a desktop application, and a backend server application.

Table 1 summarizes the details of the projects we use as case studies to evaluate the capabilities of RICK. For each case study, we provide the exact version as well as the SHA of the commit used for our experiments. This information will facilitate further replication. We also provide the number of lines of code, the number of commits, and the number of methods as indicators of the scale of the project, as well as the number of stars in the project repository, as an indicator of its visibility. The last row in Table 1 indicates the number of candidate MUTs for each case study, which is the set of MUTs with mockable method calls identified by RICK.

Our first case study is the web-based routing application based on OpenStreetMap called GRAPHHOPPER[8]. It allows users to find the route between locations on a map, considering diverse means of transport and other routing information such as elevation. We use version 5.3 of GRAPHHOPPER, with 89K lines of code (LOC), $5,844$ commits, and over 4K methods. The project's repository on GitHub has 3.7K stars.

The second case study is GEPHI, an application for working with graph data[9]. With 4.9K stargazers on GitHub, GEPHI is very popular, and has been adopted by both the industry and by researchers [29]. It allows users to import graph files, manipulate them, and export them in different file formats. We use version 0.9.6 of GEPHI, which includes $6,548$ commits and 126K LOC. For our evaluation, we exclude the GUI modules, as the generation of GUI tests has its own challenges [30] that are outside the scope of RICK. The 8 modules of GEPHI we consider are implemented in 35K LOC and contain $2,117$ methods in total.

The last case study is PDFBOX, a PDF manipulation command-line tool developed and maintained by the Apache Software Foundation[10] [31]. It can extract text and images from PDF documents, convert between text files and PDF documents, encrypt and decrypt, and split and merge PDF documents. As highlighted in Table 1, we use version 2.0.24 of PDFBOX, which has 165K LOC, over 9K methods, $8,797$ commits, and has been starred by 1.7K GitHub users.

8. https://www.graphhopper.com/open-source/
9. https://gephi.org/
10. https://pdfbox.apache.org/

TABLE 2: Characteristics of the workloads for the case studies in production: The four metrics are defined in subsection 4.2.

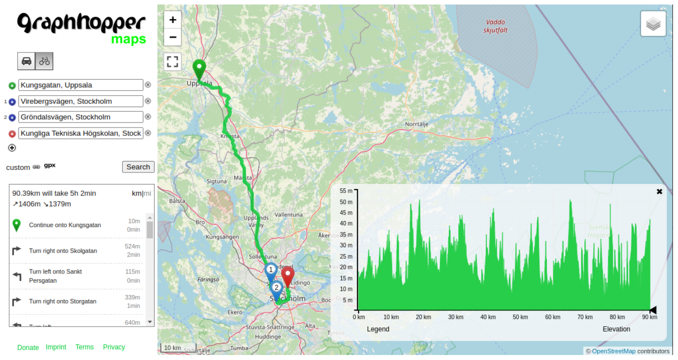
METRIC	GRAPHHOPPER	GEPHI	PDFBOX	TOTAL
MUT_ INVOKED	72	68	72	212
MOCKABLE_ INVOKED	81	63	55	199
MUT_ INVOCATIONS	73,025	21,548	7,429,800	7,524,525
MOCKABLE_ INVOCATIONS	246,822	202,630	5,144,790	5,594,242

To generate tests with RICK, the first step consists of identifying candidate MUTs, which RICK will instrument so their invocations can be monitored as the project executes in production. According to the criteria introduced in subsection 3.3, RICK identifies and instruments a total of 790 CANDIDATE_MUTs across the three applications: 356 in GRAPHHOPPER, 115 in GEPHI, and 319 in PDFBOX. These methods have interactions with objects of external types, where these objects are either the parameters of the MUT, or a field defined within the declaring type of the MUT.

4.2 Production Usage

Once the candidate MUTs for an application are identified, the instrumented application is deployed and run under a certain workload. As RICK aims at consolidating test suites for mission-critical functionalities that everybody relies on, we design workloads that exercise common features. We manually analyze the case studies' codebase and refer to their documentation, in order to design workloads that capture commonly used features and operations.

Table 2 summarizes the key characteristics of the workloads. Rows 2 and 3 capture the scope for test generation that we consider for our experiments. The number of candidate MUTs actually invoked in production is indicated by MUT_INVOKED, while MOCKABLE_INVOKED represents the number of distinct mockable methods invoked within these invoked MUTs. We also report the number of times the MUTs and their mockable methods are invoked in the rows MUT_INVOCATIONS and MOCKABLE_INVOCATIONS, respectively. The number of invocations of MUTs and mockable methods demonstrate the relevance and comprehensiveness of the production scenarios we design. They represent the extent to which we exercise the three applications in production, and reflect actual usage of their functionalities. In total, RICK focuses on 212 target methods, that invoke 199 mockable method calls. Our experiments trigger more than 7.5 million invocations of these 212 methods.

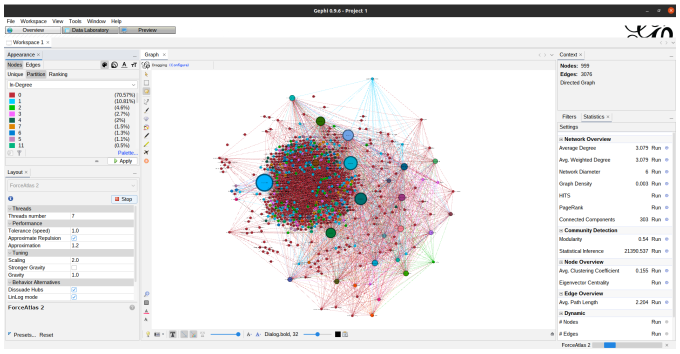


Kungsgatan, Uppsala
Virebergsvägen, Stockholm
Gröndalsvägen, Stockholm
Kungliga Tekniska Högskolan, Stock

90.39km will take 5h 2min
↗1406m ↘1379m

Continue onto Kungsgatan 10m 0min
Turn right onto Skolgatan 524m 2min
Turn left onto Sankt Persgatan 115m 0min
Turn right onto Storgatan 339m 1min

Fig. 3: Snapshot of GRAPHHOPPER in production. We query for the route between 4 locations in Sweden, as RICK monitors target method invocations.



Nodes: 999
Edges: 3076
Directed Graph

Network Overview
Average Degree 3.079
Avg. Weighted Degree 3.079
Network Diameter 6
Graph Density 0.003
HITS
PageRank
Connected Components 303
Community Detection
Modularity 0.54
Statistical Inference 21390.537
Node Overview
Avg. Clustering Coefficient 0.155
Eigenvector Centrality
Edge Overview
Avg. Path Length 2.204
Dynamic
Nodes
Edges

Fig. 4: We use GEPHI to import a data file with details on 999 Java artifacts on Maven Central. We interact with the features of GEPHI to manipulate the resulting graph. RICK monitors method invocations corresponding to these interactions in production.

GRAPHHOPPER: To experiment with GRAPHHOPPER, we deploy its server and use its website to search for the car and bike route between four points in Sweden, specifically, from the residence of each author to their common workplace in Stockholm[11]. Figure 3 is a snapshot of this experiment. Recall from subsection 4.1 that RICK monitors the invocation of 356 CANDIDATE_MUTs as GRAPHHOPPER executes. As presented in Table 2, 72 of the candidate MUTs are invoked (MUT_INVOKED), which become test generation targets for RICK. Within these MUTs, 81 distinct mockable methods are also called. With this production scenario, the 72 MUTs are invoked a total of $73,025$ times, while the 81 mockable methods are invoked $246,822$ times within the MUTs.

GEPHI: As production usage for GEPHI, we deploy the application and import a graph data file. This file has details about the top 999 Java artifacts published on Maven Central, as well as the dependencies between them. We retrieve this data file from previous work [32], [33]. We use GEPHI to produce a graph from this data, and to manipulate its layout, as illustrated in Figure 4. Finally, we export the resulting graph in PDF, PNG, and SVG formats, before exiting the application. These interactions with GEPHI lead to the invocation of 68 of the 115 candidate MUTs, and 63 distinct mockable methods called by these MUTs. Moreover, these MUTs are invoked $21,548$ times, while there are $202,630$ mockable method calls.

PDFBOX: We use the command-line utilities provided by PDFBOX to perform 10 typical PDF manipulation operations on 5 PDF documents. These documents are sourced from [34], and the operations performed on them include text and image extraction, conversion into a text file, and vice-versa, etc. This methodology has been adopted from previous work [15]. Of the 319 candidate MUTs, this workload leads to the invocation of 72 MUTs and 55 different mockable methods. The MUTs are called over 7 million times, and the mockable methods are called over 5 million times. The magnitude of these invocation counts is due to the processing of a myriad of media content from the real-world PDF documents we select.

11. https://bit.ly/3LG2zSQ

4.3 Research Questions

For each application described in subsection 4.1 and exercised in production per the workload specified in subsection 4.2, RICK generates tests with mocks using the captured production data. Through these experiments, we aim to answer the following research questions.

- ***RQ1 [methods under test]***: To what extent can RICK generate tests for MUTs invoked in production?
- ***RQ2 [production-based mocks]***: How rich is the production context reflected in the tests, mocks, and oracles generated by RICK?
- ***RQ3 [mimicking production]***: To what extent can the execution of generated tests and mocks mimic realistic production behavior?
- ***RQ4 [effectiveness]***: How effective are the generated tests at detecting regressions?
- ***RQ5 [quality]***: What is the opinion of developers about the tests generated by RICK?

Each of the research questions presented above highlights a unique aspect of the capabilities of RICK, with respect to the automated generation of tests with mocks, using data sourced from production executions.

4.4 Protocols for Evaluation

The MUTs instrumented by RICK are invoked thousands of times (Table 2). For experimental purposes, and to allow for a thorough, qualitative analysis of the results, we collect data about the first invocation of the MUT in production and use it to generate tests with RICK. We analyze these generated tests according to the following protocols in order to answer the research questions presented in subsection 4.3.

Protocol for RQ1: This first research question aims to characterize the target MUTs for which RICK transforms observations made in production into concrete test cases. We describe the MUTs by reporting their number of lines of code (LOC) as well as the number of parameters. We also report the number of tests generated by RICK for these target MUTs. This includes the tests with the three kind of mock-based oracles, **OO** for MUTs that return primitive values, **PO**, and **CO**, for one invocation of the MUT observed in production.

Protocol for RQ2: With RQ2, we analyse the ability of RICK to capture rich production contexts and turn them into test inputs and oracles that verify distinct aspects of their behavior. We answer RQ2 by dissecting the data captured by RICK as the three applications execute, as well as the tests generated by RICK using this data. First, to characterize the receiving object and parameters captured for the MUTs from production, we discuss the size of the serialized production state on disk. Second, we analyse the three kinds of oracles in the generated tests, specifically the assertion statement in the OO tests, and the number of verification statements in PO and CO tests. Furthermore, we report the number of external objects (fields and/or parameters) mocked within the tests, the stubs produced by RICK based on production observations, as well as the mock method calls.

Protocol for RQ3: With RQ3 we highlight the feasibility and complexity of automatically generating tests and mocks that successfully execute in order to mimic actual production behavior, in an isolated manner. In order to answer RQ3, we execute the generated tests, and we analyse the outcome. There are three possible outcomes of the execution of a generated test. First, a test is successfully executed if the oracles pass, implying that the test mimics the behavior of both the MUT and the mock method calls(s) observed by RICK in production. Second, a generated oracle may fail, meaning that the test and its mocks do not replicate the production observations. For example, objects recreated within the generated test through deserialization may not be identical to those observed in production [15]. Third, during the execution of the test, a runtime exception may happen before the oracles are evaluated, rendering them useless. We report on those cases as well.

Protocol for RQ4: The goal of RQ4 is to determine how effectively the tests with mock-based oracles generated by RICK can detect regressions within the MUTs. In order to do so, we inject realistic bugs within each MUT that has at least one passing RICK test [35]. We rely on the LittleDarwin mutation testing tool [36] to generate a set of first-order mutants for these MUTs. LittleDarwin is ideal for our analysis as it generates mutated source files on disk, which allows for automated, configurable, and reproducible experiments. The 14 mutation operators provided by LittleDarwin include the standard arithmetic, relational, and nullifying mutants, as well as one extreme mutation operator that replaces the whole body of an MUT with a default return statement [37]. Next, we substitute each MUT with a version that contains a mutant reachable by the test input [38], and run each test generated for the MUT by RICK. A test failure indicates that a mutant was covered, detected, and killed by the corresponding **OO**, **PO**, or **CO** within the test. We also analyze whether mock-based oracles differ in their ability to find faults [39], [40].

Protocol for RQ5: RQ5 is a qualitative assessment of the tests generated by RICK. It serves as a proxy for the readiness of the RICK tests to be integrated into the test suite of projects. To assess the quality of the generated tests, we conduct a developer survey, presenting a set of 6 tests generated by RICK for 2 MUTs in GRAPHHOPPER, to 5 software testers from the industry. We carefully select the tests to present to the survey participants in order to have a representation of the three kinds of oracles as well as diverse mocking contexts i.e., external field or parameter objects. Developer surveys have previously been conducted to assess mocking practices [3], [12]. The key novelty of our survey consists in assessing mocks that have been automatically generated.

We conduct each survey online for one hour, and follow this systematic structure: introduce mocking, the RICK test generation pipeline, and the GRAPHHOPPER case study; next, we give the participant access to a fork of GRAPHHOPPER on GitHub, with the RICK tests added, inviting the participant to clone this repository, or browse through it online; finally, we ask them questions about the generated tests. We select GRAPHHOPPER for the survey because its workload, fetching a route on a map, is intuitive and does not add to the complexity of the interview. We select the MUTs for our discussions based on the following criteria: the MUTs have at least 10 lines of code, and the tests generated for them have at least one stub. From these, we select two MUTs in GRAPHHOPPER for which RICK has generated all three mock-based oracles, and for which the tests contain a mocked field and a mocked parameter.

The goal of this survey is to gauge the opinion of developers about the quality of the 6 tests with respect to three criteria: mocking effectiveness, structure, and understandability. Our replication package includes all the details about this survey.

5 EXPERIMENTAL RESULTS

This section presents the results from our evaluation of RICK with GRAPHHOPPER, GEPHI, and PDFBOX. In subsection 5.1, subsection 5.2 and subsection 5.3, we answer RQ1, RQ2 and RQ3 based on the metrics summarized in Table 3, Table 4, and Table 5. The results for RQ4 are presented in subsection 5.4. In subsection 5.5 we answer RQ5 based on the surveys conducted with testers from the industry.

5.1 Results for RQ1 [Methods Under Test]

As presented in the first four columns of Table 3, Table 4, and Table 5, RICK generates tests for 23 MUTs in GRAPHHOPPER, 57 in GEPHI, and 48 in PDFBOX. In total, RICK generates tests for 128 MUTs which have at least one mockable method call. The median number of LOCs in these 128 target methods is 18, while the largest method is MUT#26 in PDFBOX with 328 lines of code. The median number of parameters for the MUTs is 1, while several MUTs (such as MUT#38 and MUT#39 in GEPHI) take as many as 6 parameters. In general, RICK handles a wide variety of MUTs in the case studies, with successful identification and instrumentation of these methods, detailed monitoring in production, as well as the generation of tests that compile and run.

These results validate that mock generation from production can indeed be fully automated, and is robust with respect to the complexity of real world methods. RICK handles the diversity of methods, data types, and interactions observed in real software and production usage scenarios.

In Table 2, we see that the workloads trigger the execution of 72 MUTs in GRAPHHOPPER, 68 in GEPHI, and 72 in PDFBOX. A subset of them are actually used as targets for

TABLE 3: Experimental results for GRAPHHOPPER

RQ1: METHOD UNDER TEST				RQ2: PRODUCTION-BASED MOCKS							RQ3: MIMICKING PRODUCTION		
MUT_ID	#LOC	#PARAMS	#TESTS	CAPTURED_OBJ_SIZE	#MOCK_OBJECTS	#MOCK_METHODS	#STUBS	#OO_STMNTS	#PO_STMNTS	#CO_STMNTS	#SUCCESSFULLY_MIMIC	#INCOMPLETELY_MIMIC	#UNHANDLED_MUT_BEHAVIOR
MUT # 1	7	1	2	41 B	1	1	0	0	1	1	0	0	2
MUT # 2	19	1	3	47 B	1	3	6	1	6	9	0	3	0
MUT # 3	8	1	2	53 B	1	2	0	0	2	3	2	0	0
MUT # 4	5	1	2	284 B	1	1	1	0	1	1	2	0	0
MUT # 5	15	1	3	477 B	1	1	0	1	1	1	3	0	0
MUT # 6	13	0	3	982 B	1	3	2	1	3	3	3	0	0
MUT # 7	7	1	2	1.1 KB	1	1	1	1	1	1	2	0	0
MUT # 8	15	2	3	1.5 KB	1	2	4	1	4	2	3	0	0
MUT # 9	25	0	3	2.7 KB	2	5	9	1	10	8	0	3	0
MUT # 10	15	0	3	2.7 KB	2	3	11	1	7	6	0	3	0
MUT # 11	26	0	3	3 KB	2	2	11	1	11	2	0	3	0
MUT # 12	5	1	3	3.1 KB	1	1	1	1	1	1	3	0	0
MUT # 13	5	1	3	3.1 KB	1	1	1	1	1	1	3	0	0
MUT # 14	25	0	3	56.7 KB	2	4	5	1	5	5	0	3	0
MUT # 15	5	1	3	156.8 KB	1	1	1	1	1	1	3	0	0
MUT # 16	58	1	3	156.8 KB	1	1	1	1	1	1	3	0	0
MUT # 17	18	1	3	156.9 KB	1	1	1	1	1	1	3	0	0
MUT # 18	5	1	3	383 KB	1	1	1	1	1	1	3	0	0
MUT # 19	74	1	2	590 KB	2	2	0	0	2	2	2	0	0
MUT # 20	8	0	2	729.2 KB	1	2	20	0	20	20	0	2	0
MUT # 21	57	0	3	2.2 MB	3	3	2	1	3	3	0	0	3
MUT # 22	37	2	3	9.8 MB	1	3	3	1	4	4	2	1	0
MUT # 23	4	0	2	9.8 MB	2	1	0	0	1	1	0	2	0
TOTAL: 23	**MEDIAN: 15**	**MEDIAN: 1**	**62**	**MEDIAN: 3.1 KB**	**31**	**45**	**81**	**16**	**88**	**78**	**37**	**20**	**5**

test generation: 23, 57, and 48 MUTs in the respective case studies. This happens due to two reasons.

First, when statically identifying targets for test generation, RICK finds MUTs with mockable method calls. However, as highlighted in subsubsection 3.3.2, MUTs may be invoked, without their corresponding mockable methods being called, because of the flow of control through the program. For example, a mockable method call may happen only within a certain path through the MUT, which is not observed in production. In this case, a test with mocks is not generated. Second, the receiving objects for some MUTs are sometimes too large and complex to be captured through serialization, which is the case for some MUTs invoked within GRAPHHOPPER. We have observed serialized object snapshots beyond tens of megabytes, which reaches the scalability limits of state of the art serialization techniques.

The difference between the number of invoked MUTs and the number of MUTs for which RICK generates tests is most significant for GRAPHHOPPER. We notice that many receiving objects for GRAPHHOPPER do not get successfully serialized owing to their large size. For example, the receiving object for an invoked MUT was as large as 454 MB, before even being serialized as XML. To counter this, we allocated more heap space, increasing it up to 9 GB, while deploying the server, yet were unsuccessful in serializing it. Our research on using serialization for automated mocking clearly touches the frontier of serialization for handling arbitrarily large and complex data from production.

The total number of tests generated by RICK for the 128 target MUTs is 294. Recall from subsection 4.2 that we only monitor and generate tests for a single invocation – the first one – of each of the target MUTs. Note also that, as signified by the column #TESTS in Table 3, Table 4, and Table 5, RICK generates either 2 or 3 tests for each target MUT. The number of tests generated for an MUT depends on its return type. For MUTs that return a non-void, primitive value, such as MUT#2 in GRAPHHOPPER, RICK generates an **OO** test to assert on the output of the MUT invocation, in addition to a **PO** and **CO** test. Across the three case studies, RICK generates **OO** tests for 38 MUTs, and **PO** and **CO** tests for all 128 MUTs.

Answer to RQ1

RICK captures the production behavior for a set of 128 out of the 212 MUTs invoked in production. RICK transforms the data collected from these production invocations into 294 concrete tests with different oracles. The key result of RQ1 is that RICK handles a large variety of real world methods in an end to end manner, from monitoring in production to the generation of tests with mocks.

5.2 Results for RQ2 [Production-based Mocks]

RICK generates mocks from real data observed in production. While RQ1 has demonstrated feasibility, RQ2 explores how production data is reflected in the generated test cases.

Columns 5 through 11 in Table 3, Table 4, and Table 5 present the results for RQ2. Each row highlights the data that RICK collects for one MUT and its mock method call(s) from production. The column CAPTURED_OBJ_SIZE presents the size on disk (in B / KB / MB) of the serialized object states for the receiving object and, if present, the parameter objects of the MUT. We also present the number of external objects mocked within the test (#MOCK_OBJECTS), the number of methods called on these mock objects (#MOCK_METHODS), the number of stubs (#STUBS) defined for these mock methods, and the number of statements corresponding to the **OO**, **PO**, and **CO** oracles in the gener-

TABLE 4: Experimental results for GEPHI

RQ1: METHOD UNDER TEST				RQ2: PRODUCTION-BASED MOCKS							RQ3: MIMICKING PRODUCTION		
MUT_ID	#LOC	#PARAMS	#TESTS	CAPTURED_ OBJ_SIZE	#MOCK_ OBJECTS	#MOCK_ METHODS	#STUBS	#OO_ STMNTS	#PO_ STMNTS	#CO_ STMNTS	#SUCCESSFULLY_ MIMIC	#INCOMPLETELY_ MIMIC	#UNHANDLED_ MUT_BEHAVIOR
MUT # 1	16	1	2	48 B	1	2	2	0	2	2	0	0	2
MUT # 2	14	1	2	48 B	1	1	1	0	1	1	0	0	2
MUT # 3	9	3	2	56 B	1	1	1	0	1	1	2	0	0
MUT # 4	6	1	3	56 B	1	1	1	1	1	1	0	0	3
MUT # 5	16	1	3	56 B	1	1	1	1	1	1	3	0	0
MUT # 6	31	3	3	56 B	1	1	0	1	1	1	3	0	0
MUT # 7	12	2	2	118 B	1	4	2	0	4	4	2	0	0
MUT # 8	12	2	2	134 B	1	2	0	0	2	2	1	1	0
MUT # 9	11	2	3	143 B	1	2	2	1	2	2	3	0	0
MUT # 10	8	1	2	186 B	1	1	0	0	1	1	0	0	2
MUT # 11	18	3	2	206 B	1	4	4	0	4	20	1	1	0
MUT # 12	13	1	2	237 B	1	2	1	0	2	2	0	0	2
MUT # 13	14	1	2	238 B	1	1	1	0	1	1	0	0	2
MUT # 14	12	1	2	255 B	1	1	1	0	1	1	0	0	2
MUT # 15	17	1	2	291 B	1	1	1	0	1	1	0	0	2
MUT # 16	24	1	2	299 B	1	1	1	0	1	1	0	0	2
MUT # 17	43	3	3	307 B	1	1	1	1	1	1	0	0	3
MUT # 18	18	1	2	340 B	1	1	1	0	1	1	0	0	2
MUT # 19	7	1	2	344 B	1	1	1	0	1	1	2	0	0
MUT # 20	29	1	2	425 B	1	1	1	0	1	1	0	0	2
MUT # 21	11	1	3	472 B	1	2	2	1	2	2	3	0	0
MUT # 22	40	3	2	1.7 KB	3	3	1	0	3	3	0	2	0
MUT # 23	50	2	2	2.2 KB	1	2	1	0	2	2	0	2	0
MUT # 24	23	3	2	4.4 KB	1	1	1	0	1	1	2	0	0
MUT # 25	13	1	2	58.2 KB	3	3	3	0	3	3	0	2	0
MUT # 26	4	1	3	58.2 KB	1	1	1	1	1	1	3	0	0
MUT # 27	23	3	2	59.2 KB	1	1	1	0	1	1	2	0	0
MUT # 28	53	1	3	60.1 KB	1	4	10	1	4	27	0	3	0
MUT # 29	21	2	2	79 KB	1	1	1	0	1	1	2	0	0
MUT # 30	85	4	2	100 KB	1	1	1	0	1	1	0	0	2
MUT # 31	85	1	2	112.5 KB	4	4	5	0	4	6	0	0	2
MUT # 32	25	1	2	112.5 KB	1	1	1	0	1	1	2	0	0
MUT # 33	10	1	3	149.5 KB	1	1	1	1	1	1	2	1	0
MUT # 34	90	0	3	177.4 KB	1	1	1	1	1	1	0	3	0
MUT # 35	15	3	2	196.3 KB	1	3	0	0	3	3	2	0	0
MUT # 36	28	2	2	276 KB	1	1	1	0	1	1	0	0	2
MUT # 37	40	2	2	300 KB	1	1	1	0	1	1	0	0	2
MUT # 38	18	6	2	370 KB	1	1	1	0	1	1	0	0	2
MUT # 39	16	6	3	600 KB	1	1	1	1	1	1	0	0	3
MUT # 40	5	2	2	3.6 MB	1	1	1	0	1	1	0	0	2
MUT # 41	62	6	2	3.6 MB	2	8	8	0	8	8	0	0	2
MUT # 42	15	6	2	3.8 MB	1	1	1	0	1	1	0	0	2
MUT # 43	26	0	2	3.9 MB	1	1	6	0	1	1	0	2	0
MUT # 44	185	0	2	3.9 MB	1	1	6	0	1	1	0	0	2
MUT # 45	11	0	2	3.9 MB	1	1	0	0	1	1	2	0	0
MUT # 46	123	0	2	3.9 MB	2	3	1	0	3	3	0	2	0
MUT # 47	59	5	2	4 MB	1	1	3	0	1	1	0	0	2
MUT # 48	45	6	2	4.1 MB	2	2	3	0	2	2	0	0	2
MUT # 49	20	0	2	4.2 MB	1	1	0	0	1	1	2	0	0
MUT # 50	18	2	3	4.5 MB	1	1	0	1	1	1	0	0	3
MUT # 51	126	0	2	10.8 MB	1	1	6	0	1	1	0	2	0
MUT # 52	24	0	2	10.8 MB	1	1	1	0	1	1	0	2	0
MUT # 53	24	0	2	10.8 MB	1	1	1	0	1	1	0	2	0
MUT # 54	15	2	2	11.2 MB	1	1	1	0	1	1	2	0	0
MUT # 55	11	2	2	11.6 MB	1	2	4	0	2	2	1	1	0
MUT # 56	7	3	2	11.6 MB	1	1	1	0	1	1	2	0	0
MUT # 57	32	3	2	11.6 MB	1	2	3	0	3	3	0	0	2
TOTAL: 57	**MEDIAN: 18**	**MEDIAN: 1**	**126**	**MEDIAN: 60.1 KB**	**67**	**93**	**103**	**12**	**94**	**135**	**44**	**26**	**56**

TABLE 5: Experimental results for PDFBox

RQ1: Method Under Test				RQ2: Production-based Mocks							RQ3: Mimicking Production		
MUT_ID	#LOC	#PARAMS	#TESTS	CAPTURED_ OBJ_SIZE	MOCK_ OBJECTS	MOCK_ METHODS	#STUBS	#OO_ STMNTS	#PO_ STMNTS	#CO_ STMNTS	#SUCCESSFULLY_ MIMIC	#INCOMPLETELY_ MIMIC	#UNHANDLED_ MUT_BEHAVIOR
MUT # 1	84	5	2	37 B	1	1	0	0	1	1	2	0	0
MUT # 2	6	1	2	352 B	1	2	2	0	2	4	0	0	2
MUT # 3	11	1	2	3.3 KB	1	1	0	0	1	1	0	0	2
MUT # 4	5	1	2	4.3 KB	1	1	0	0	1	1	2	0	0
MUT # 5	5	1	2	4.7 KB	1	1	0	0	1	1	2	0	0
MUT # 6	101	3	2	6.5 KB	1	1	1	0	1	1	0	0	2
MUT # 7	28	1	2	11.7 KB	1	1	0	0	5	1	2	0	0
MUT # 8	34	1	2	267 KB	1	1	1	0	3	2	2	0	0
MUT # 9	42	2	3	1.6 MB	1	1	1	1	1	1	3	0	0
MUT # 10	33	1	2	2 MB	1	1	0	0	1	1	2	0	0
MUT # 11	16	1	2	2.2 MB	1	1	1	0	1	1	0	0	2
MUT # 12	10	0	3	2.3 MB	1	2	2	1	2	2	3	0	0
MUT # 13	19	3	3	2.3 MB	1	2	2	1	2	2	3	0	0
MUT # 14	19	0	2	2.4 MB	1	1	1	0	1	1	2	0	0
MUT # 15	18	0	3	3 MB	1	1	1	1	1	1	3	0	0
MUT # 16	18	2	2	3 MB	1	1	1	0	1	1	2	0	0
MUT # 17	8	0	2	3.2 MB	1	1	1	0	1	1	2	0	0
MUT # 18	9	1	3	3.6 MB	1	1	1	1	1	1	3	0	0
MUT # 19	8	0	3	3.8 MB	1	1	1	1	1	1	3	0	0
MUT # 20	18	1	2	3.8 MB	1	2	2	0	2	2	2	0	0
MUT # 21	5	1	2	4.4 MB	1	1	1	0	1	1	2	0	0
MUT # 22	19	0	3	4.6 MB	1	2	2	1	2	2	3	0	0
MUT # 23	39	1	2	4.6 MB	1	1	0	0	1	1	2	0	0
MUT # 24	85	1	3	5.3 MB	3	1	1	1	1	1	0	3	0
MUT # 25	40	1	2	5.3 MB	1	1	1	0	1	1	0	0	2
MUT # 26	328	2	2	5.4 MB	1	1	1	0	1	1	0	0	2
MUT # 27	11	1	3	5.6 MB	1	2	2	1	2	2	3	0	0
MUT # 28	29	2	3	5.8 MB	1	2	3	1	2	4	2	1	0
MUT # 29	79	4	2	6.9 MB	1	1	0	0	1	1	2	0	0
MUT # 30	19	1	2	6.4 MB	1	1	0	0	1	1	2	0	0
MUT # 31	31	1	2	6.6 MB	1	1	0	0	1	1	2	0	0
MUT # 32	62	4	2	6.8 MB	1	1	1	0	1	1	2	0	0
MUT # 33	23	1	2	7.2 MB	1	1	0	0	1	1	2	0	0
MUT # 34	27	2	2	7.4 MB	1	1	0	0	1	1	0	0	2
MUT # 35	40	1	2	7.6 MB	1	1	1	0	1	1	0	0	2
MUT # 36	20	1	2	8.3 MB	1	1	1	0	1	1	0	2	0
MUT # 37	12	1	2	9.5 MB	1	1	0	0	1	1	2	0	0
MUT # 38	12	1	2	9.5 MB	1	1	0	0	1	1	2	0	0
MUT # 39	11	1	2	9.7 MB	1	1	0	0	1	1	2	0	0
MUT # 40	23	1	2	10.1 MB	1	1	0	0	1	0	2	0	0
MUT # 41	11	0	2	11.1 MB	1	1	0	0	1	1	1	1	0
MUT # 42	5	1	2	21.4 MB	1	1	0	0	1	1	2	0	0
MUT # 43	8	4	2	30.8 MB	3	5	4	0	7	13	0	2	0
MUT # 44	14	0	2	31 MB	1	2	0	0	2	2	0	0	2
MUT # 45	17	2	2	32 MB	1	2	0	0	3	3	0	2	0
MUT # 46	36	2	2	32 MB	1	1	0	0	1	1	0	0	2
MUT # 47	12	0	2	34 MB	2	5	0	0	5	5	2	0	0
MUT # 48	117	1	2	39 MB	1	2	2	0	2	3	0	0	2
TOTAL: 48	**MEDIAN: 19**	**MEDIAN: 1**	**106**	**MEDIAN: 5.3 MB**	**53**	**66**	**38**	**10**	**75**	**80**	**73**	**11**	**22**

ated tests (signified through #OO_STMNTS, #PO_STMNTS, and #CO_STMNTS, respectively).

We first discuss the serialized production state, defined as a receiving object on which the MUT is invoked, as well as the parameters with which this invocation is made. Recall that the receiving object is serialized. For example, consider MUT#32 in PDFBox (Table 5). RICK captures the production state of the receiving object on which MUT#32 gets invoked, and the 4 parameters for this invocation. The size of these captured objects amounts to a total of 6.8 MB. These objects serve as a snapshot for reproducing the production state within the 2 tests generated by RICK for MUT#32. For our experiments with the three applications, the maximum size of the captured objects is 39 MB for MUT#48 in PDFBox. The median size of these captured objects is 486 bytes of serialized data. Table 5 characterizes the realistic nature of the objects that RICK captures over 128 real-world MUTs, as they are invoked during the end-to-end execution of the applications.

The main feature of RICK is to monitor and collect data about mockable method calls that occur within the MUT, with the receiving objects and passed parameters. Specifically, as detailed in subsection 3.3, RICK mocks the parameters and fields of external types, on which mockable method calls occur in production. For example, there are 2 mock objects within the 2 tests generated for MUT#19 in GraphHopper (Table 3) for two fields in the declaring type of MUT#19. Moreover, 2 mock method calls are made within

```
1 @Test
2 @DisplayName("moveNode with parameter oracle,
3 mocking Node.x(), Node.y(), Node.setX(float), Node.setY(float)")
4 public void testMoveNode_PO() throws Exception {
5   // Arrange
6   StepDisplacement receivingObject =
        deserialize("receiving.xml");
7   Object[] paramObjects = deserialize("params.xml");
8   ForceVector paramObject2 = (ForceVector) paramObjects[1];
9   Node mockNode = Mockito.mock(Node.class);
10  when(mockNode.x()).thenReturn(-423.78378F);
11  when(mockNode.y()).thenReturn(107.523186F);

13  // Act
14  receivingObject.moveNode(mockNode, paramObject2);

16  // Assert
17  verify(mockNode, atLeastOnce()).x();
18  verify(mockNode, atLeastOnce()).y();
19  verify(mockNode, atLeastOnce()).setX(-403.92587F);
20  verify(mockNode, atLeastOnce()).setY(105.14341F);
21 }
```

Listing 7: The **PO** test generated by RICK for MUT#7 in GEPHI, which is a method called `moveNode`. The test mocks a parameter on which four mocked methods are invoked. The behavior of two mocked methods is stubbed within the test.

the tests, one on each of the 2 mock objects. In comparison, there is 1 mock object within each of the 3 tests generated for MUT#21 in GEPHI (Table 4). This mock object replaces the parameter of MUT#21, on which 2 mockable method calls are observed by RICK in production. In total, RICK uses 151 mock objects as parameter or field across the generated tests. 204 mock methods are invoked on these mock objects, reflecting the production interactions with external objects within the MUTs. These generated mock objects recreate actual interactions of the MUT with its environment. The data serialized from production provide developers with realistic test data.

Within the generated tests, the behavior of the mock objects is defined through method stubs. A stub provides the canned response that should be returned from a non-void method called on a mock object, given a set of parameters, i.e., it defines the behavior of a mock method within the test for an MUT. RICK sources the parameters and the primitive returned value from production observations, and represents them through stubs within the generated tests. For example, RICK generates 2 tests for MUT#7 in GEPHI (Table 4). This MUT is the method `moveNode(Node,ForceVector)`. Within each of the generated tests, 1 of the parameters of `moveNode` is mocked, and 4 methods are invoked on this mock object. We present the **PO** test generated for `moveNode` in Listing 7. On line 9 the `Node` object is mocked. Within the invocation of `moveNode` in production, RICK recorded the invocation of the methods `x()` and `y()` on the `Node` object, including their returned values. Consequently, RICK expresses their behavior through the 2 stubs in the generated test (lines 10 and 11) using this production data. The invocations of the two other mockable methods `setX(float)` and `setY(float)` are not stubbed because they are void methods. In total, RICK generates a total of 222 stubs to mimic the production behavior of mockable method calls that occur within MUTs. These generated stubs guide the behavior of the MUT within the generated test, per the observations made for it in production.

We observe from the column #OO_STMNTS in Table 3, Table 4, and Table 5, that the number of **OO** statements for any MUT is either 0 or 1. This is because the oracle in **OO** tests is expressed as a single assertion statement for the output of an MUT that returns a primitive value. However, the number of **PO** and **CO** statements within each test varies depending on the observations made for the corresponding MUT in production. For instance, three tests are generated for MUT#8 in GRAPHHOPPER (Table 3). In each of the three tests, there is 1 mock object, and 2 different mock methods are invoked on this mock object. The behavior of the 2 mock methods is defined through the 4 generated stubs. The **OO** test has an assertion to verify the output of MUT#8. The **PO** test has 4 verification statements to verify the parameters with which the mock methods are called. The **CO** test has 2 verification statements which correspond to the observations made by RICK about the sequence and frequency of these mock method calls within the invocation of MUT#8 in production.

In total, across the 294 tests generated for the 128 MUTs, RICK generates 38 assertion statements (one in each **OO**), 257 statements that verify the parameters with which mock methods are called, and 293 statements to verify the sequence and frequency of mock method calls. Furthermore, RICK uses the parameters passed to, and the value returned from the mockable method calls observed in production, to generate a total of 222 stubs across the 294 tests.

Answer to RQ2

RICK captures a wide range of production data for test generation. We have demonstrated with static insights that it can handle well the two dimensions of mock-based testing: capturing the production state in mock objects, and stubbing real-world methods. The analysis of the generated tests shows that RICK can generate various types of oracles that verify different aspects of the MUT interacting with its environment.

5.3 Results for RQ3 [Mimicking Production]

The results for RQ3 are presented in the last three columns in Table 3, Table 4, and Table 5. All the tests generated by RICK for the 128 MUTs are self-contained and compile correctly. We run each generated test ten times and verify that it is not flaky. Next, for each test, we report the status of its execution. In each row, the column #SUCCESSFULLY_MIMIC highlights the number of tests that completely recreate the observed production context with mocks and oracle(s) that pass, while #INCOMPLETELY_MIMIC signifies the number of tests for which at least one oracle fails. The last column, #UNHANDLED_MUT_BEHAVIOR, represents those test executions where we observe a runtime exception thrown by the MUT. We now discuss the implications of each of these scenarios, and why they occur.

An **OO** test successfully mimics the production context if the MUT returns the same output as it did in production, when invoked with mocks replacing the external objects, and stubs for the behavior of the mock method calls. If the test does not completely mimic the production behavior of the MUT, the invocation of the MUT returns a different output, which is unequal to the one returned by the MUT in production. Consequently, the assertion statement within the **OO** test fails.

For a **PO** test to be successful, all the verification statements must pass, indicating that mock methods are called by the MUT within the generated test with the same arguments as the ones observed in production. In comparison, a failure in any of the verification statements implies that a mock method call occurs with different parameters than the ones observed for its invocation in production. This implies that the test does not faithfully recreate the observed interactions of the MUT and the mock method.

A passing **CO** test verifies that the mock method calls occur in the same order and the same number of times within the MUT, as they did in production. On the contrary, if the test does not completely mimic the order and/or frequency of mock method calls, a verification statement will fail.

The proportion of tests that successfully mimic production behavior differs across case studies (column #SUCCESSFULLY_MIMIC). Of the tests generated for GRAPHHOPPER, 59.7% are successful. In GEPHI, the successful tests are 35% of the total generated tests, while in PDFBOX 68.9% tests are successful. Overall, the 154 successful tests account for 52.4% of all the tests generated. This is arguably a high ratio given the multi-stage pipeline of RICK, where each stage can fail in some conditions. In 52.4% of cases, all stages of RICK succeed: All the captured objects are correctly serialized and deserialized before the MUT is invoked, recreating an appropriate and realistic execution state. Also, the mock methods are successfully stubbed: they mimic production behavior without impacting the behavior of the MUT.

In the column #INCOMPLETELY_MIMIC, we observe that 57 generated tests, which account for 19.4% of the total, have a failing oracle, implying that they do not completely mimic production behavior. This includes 32.2% of the GRAPHHOPPER tests, 20.6% GEPHI tests, and 10.4% of the tests generated for PDFBOX. These failures can occur due to the following reasons.

Unfaithful recreation of production state: The captured objects may be inaccurately deserialized within the generated test, implying that production states are not completely recreated. Deserialization of complex objects captured from production is a known problem and a key challenge for recreating real production conditions in generated tests [15], [27]. A test may also fail because a production resource, such as a file, is not available during test execution, resulting in an exception. Moreover, since mocks are skeletal objects that substitute a concrete object within the test, they can induce a change in the path taken through the MUT, which renders the oracle unsuccessful. For example, the tests generated for MUT#2 in GRAPHHOPPER (Table 3) fail due to failing **OO**, **PO**, and **CO** oracles. The tests mock an object of type `com.graphhopper.util.PointList`, and MUT#2 tries to invoke a loop over the size of this mock list of points. Since the loop is not exercised, a different path is traversed through the MUT than the one observed in production.

Type-based stubbing: We observe that some tests fail because of the granularity of stubbing. Glowroot, the current infrastructure for monitoring within RICK, identifies a target type based on its fully qualified name. Consequently, RICK stubs mockable methods called on the type of an object, but not based on specific instances of the object. A failure can occur if an MUT calls the same mockable method on multiple parameters or fields of the same type. For instance, we present an excerpt of MUT#24 in PDFBOX in Listing 8. This method `codeToGID(int)` (line 6), has three calls to the same mockable method, `getGlyphId(int)` (lines 9, 11, and 13). These calls are made on three different fields of type `CmapSubtable` called `cmapWinUnicode`, `cmapWinSymbol`, and `cmapMacRoman` defined in the `PDTrueTypeFont` class (lines 2 to 4). RICK records the mockable method call in production, and mocks the three fields. However, the information on which of these mock fields actually calls the mock method is not available. One solution would be to do object-based stubbing, but we are not aware of any work on this and consider this sophistication as future work.

```
 1 class PDTrueTypeFont {
 2   CmapSubtable cmapWinUnicode;
 3   CmapSubtable cmapWinSymbol;
 4   CmapSubtable cmapMacRoman;

 6   public int codeToGID(int code) {
 7     ...
 8     if (...) {
 9       gid = cmapWinUnicode.getGlyphId(...);
10     } else if (...) {
11       gid = cmapMacRoman.getGlyphId(...);
12     } else {
13       gid = cmapWinSymbol.getGlyphId(...);
14     }
15     ...
16     return gid;
17   }
18 }
```

Listing 8: The same mockable method, `getGlyphId`, is called on three different fields of the same type within MUT#24 of PDFBOX, `codeToGID`

We now discuss the cases of #UNHANDLED_MUT_BEHAVIOR. The execution of 83 of the 294 generated tests (28.2%) causes exceptions to be thrown by the MUT. For example, the MUT may have multiple non-mockable interactions with the mock object, before the mock method is invoked on it. Any of these other interactions can behave unexpectedly, resulting in exceptions to be raised before the oracle is even evaluated within the test. Examining the test execution logs, we see such unhandled behaviors as exceptions. We observe these cases in all three applications: 8.1% in GRAPHHOPPER, 44.5% in GEPHI, 20.7% in PDFBOX. For example, a null pointer exception is thrown from MUT#1 in GRAPHHOPPER, when it calls other methods on the mock object before the mockable method is called. Across the 83 unhandled cases, 74 arise from null pointer exceptions, of which 54 are in GEPHI, 18 in PDFBOX, and 2 in GRAPHHOPPER. We find the other two unhandled cases for GEPHI in the tests generated for MUT#31, `addEdge(EdgeDraft)`, where the parameter `EdgeDraft` is one of the 4 mock objects. This MUT calls another method, which has been designed by the developers of GEPHI to throw a `ClassCastException` if `EdgeDraft` is not an instance of `ElementDraftImpl`, which fully explains the failure to execute with a mock. In GRAPHHOPPER, the three tests generated for `load` (MUT#21) mock the field `EncodingManager` within the type `GraphHopper`. This MUT invokes method `checkProfilesConsistency`, which is designed to return an `IllegalArgumentException` if the `EncodingManager` does not have an encoder for the vehicle set in the profile. In PDFBox, the four tests generated for `prepareForDecryption` (MUT#6) and `getColorSpace` (MUT#34) throw an `IOException` because methods called by these MUTs find an unexpected value when accessing a field within the mock object. Across all these cases, the unhandled behavior occurs within a non-mockable method which is called by the MUT, and is thus indirectly called by the generated test. Our results demonstrate that automatic mocking is full of caveats and handling all corner cases is an important direction for future work on automated mock generation.

Answer to RQ3

RICK succeeds in generating 294 tests, of which 154 (52.4%) fully mimic production observations with fully passing oracles. For 19.4% of the test cases, at least one oracle statement fails, showing that the oracles can indeed differentiate between successfully mimicked and incompletely mimicked contexts. At runtime, the majority of the tests generated by RICK completely mimic production behavior in the sense that the state asserted by the oracle is equal to the one observed in production. The cases where the generated tests fail at runtime reveal promising research directions for sophisticated production monitoring tools, such as effective deserialization and efficient resource snapshotting.

5.4 Results for RQ4 [Effectiveness]

As described in subsection 4.4, we want to determine the effectiveness of RICK tests at determining regressions [35]. We generate a set of first-order mutants for each MUT with LittleDarwin [36]. We focus the generation of mutants for the 68 MUTs across the three projects that have at least one passing RICK test. Furthermore, we consider the mutants that are covered by the test input in the generated tests, i.e., mutants that lie on the path of the MUTs exercised by the tests. This is because a mutant that lies on an uncovered path will be undetectable by design [38]. In total, we consider 449 mutants: 69 mutants for the 14 GraphHopper MUTs that have at least one passing test, 107 mutants for the 21 MUTs in Gephi, and 273 mutants for the 33 PDFBox MUTs. Our replication package[12] contains the automated mutation analysis pipeline, as well as the generated mutants and test execution logs. We also include detailed reports on the set of mutants detected by each mock-based oracle of each MUT.

Our findings from the execution of the generated tests against the mutants are summarized in Figure 5. The Venn diagrams represent the distribution of the 210 mutants killed by **OO**, **PO**, and **CO** for the three case studies. We note from the Venn diagrams that 19 mutants in GraphHopper, 8 mutants in Gephi, and 20 in PDFBox are killed by all three mock-based oracles. Meanwhile, for all three projects, the three mock-based oracles differ in their ability to detect mutants. For example, per Figure 5a, 2 and 4 mutants in

12. https://github.com/ASSERT-KTH/rick-experiments

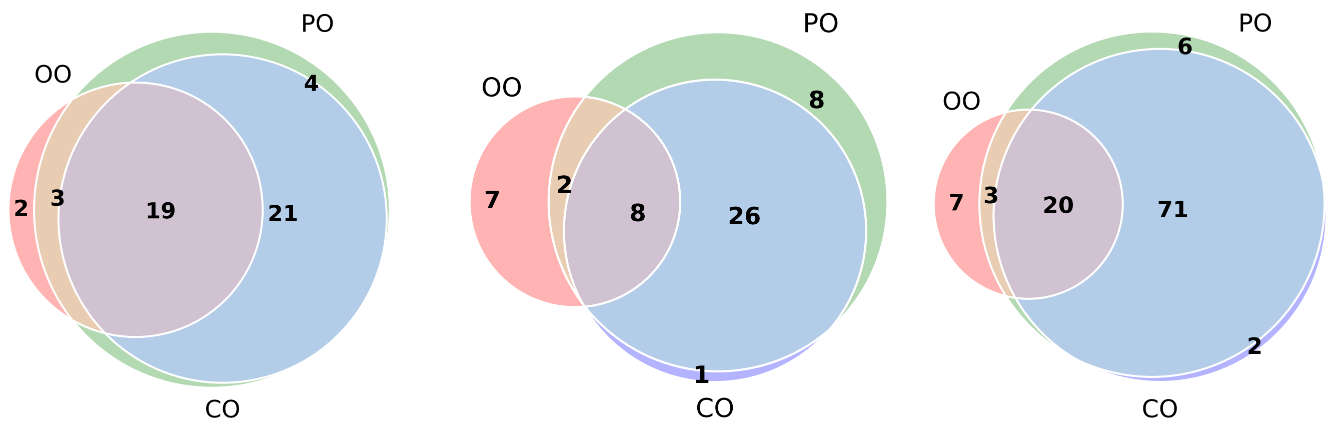
```java
public class LineIntIndex {
 ...
 public boolean loadExisting() {
  ...
  if (!dataAccess.loadExisting())
   return false;
  ...
  GHUtility.checkDAVersion(..., dataAccess.getHeader(0));
  checksum = dataAccess.getHeader(1 * 4);
  minResolutionInMeter = dataAccess.getHeader(2 * 4);
  ...
  return true;
 }
}
```

Listing 9: The original MUT#8 in GraphHopper, `loadExisting`

GraphHopper are detected only by **OO** and **PO**, respectively. Likewise, in Figure 5c, 2 mutants in PDFBox result in extra invocations of a mocked method that are only detected by **CO**. Moreover, in Figure 5b, 2 mutants in Gephi are killed by **OO** and **PO**, but not **CO**. Across the three projects, **OO** kills 16 mutants, **PO** kills 18 mutants, and **CO** kills 2 mutants that are undetected by other oracles. The lower number of mutants killed only by **CO** can be attributed to the fact that LittleDarwin does not include a mutation operator that directly removes method calls. Also, mocked methods may be invoked in the expected order and frequency, but with different, mutated parameters. This will not be detected by **CO** but will be detected by **PO**. The set of mutants killed by **OO** is always smaller than for the other oracles. This is because, as highlighted in subsection 5.1, we generate **OO** tests only for MUTs that return primitive values. Yet, when present, **OO** kills mutants in all three projects, i.e., 24 mutants in GraphHopper, 17 in Gephi, and 30 mutants in PDFBox.

Overall, all three types of oracles are effective at detecting regressions. We have also observed that the generated tests kill at least one mutant for each MUT. These observations are evidence that the RICK tests, with inputs sourced from production, indeed specify the behavior of the MUTs. Moreover, **OO**, **PO**, and **CO** can detect different bugs. The RICK tests with mock-based oracles can therefore complement each other, even given the same test input. This aligns with the findings of Staats *et al.* [39] and Gay *et al.* [40] that multiple oracles specified for a test input may perform differently with respect to their fault-finding ability.

We illustrate this phenomenon using the example of MUT#8 in GraphHopper, which is the `loadExisting` method presented in Listing 9. Listing 11 presents the common *Arrange* and *Act* phases of the three tests generated by RICK for `loadExisting`. Listing 12, Listing 13, and Listing 14 contain the *Assert* phase of the **OO**, **PO**, and **CO** test, respectively. The 5 mutants produced by LittleDarwin for `loadExisting`, are shown in Listing 10. The generated **OO** test detects 2 of these mutants (#2 and #3), as the assertion (Listing 12) fails on an output that differs from the expected `boolean` value. The **CO** (Listing 14) kills 3 mutants (#1, #2, and #3), as the invocations to the methods `loadExisting` and `getHeader` are expected on the mocked `DataAccess` object, but do not occur due to the mutation. The verification statements in the **PO** (Listing 13) kill all 5 mutants. This is because the expected mock method invocations within the MUT either do not occur entirely, or occur with unexpected



(a) GRAPHHOPPER (b) GEPHI (c) PDFBOX

Fig. 5: The RICK tests kill 49 mutants in GRAPHHOPPER, 52 mutants in GEPHI, and 109 mutants in PDFBOX. The three types of mock-based oracles complement each other to detect regressions in all three projects.

```
// Mutant #1: Extreme mutation - Lines 4 to 12
public boolean loadExisting() {
- if (initialized)
- ...
- return true;
+ return true;
}

// Mutant #2: Extreme mutation - Lines 4 to 12
public boolean loadExisting() {
- if (initialized)
- ...
- return true;
+ return false;
}

// Mutant #3: Line 5
- if (!dataAccess.loadExisting())
+ if (dataAccess.loadExisting())

// Mutant #4: Line 9
- checksum = dataAccess.getHeader(1 * 4);
+ checksum = dataAccess.getHeader(1 / 4);

// Mutant #5: Line 10
- minResolutionInMeter = dataAccess.getHeader(2 * 4);
+ minResolutionInMeter = dataAccess.getHeader(2 / 4);
```

Listing 10: Five first-order mutants are introduced in `loadExisting`. The line numbers correspond to the line numbers in Listing 9.

```
@Test
@DisplayName("Test for loadExisting, mocking
DataAccess.loadExisting(), DataAccess.getHeader(int)")
public void testLoadExisting() {
 // Arrange
 LineIntIndex receivingObject = deserialize(
      "receiving.xml");
 DataAccess mockDataAccess =
      insertMockField_DataAccess_InLineIntIndex(
      receivingObject);
 when(mockDataAccess.loadExisting()).thenReturn(true);
 when(mockDataAccess.getHeader(0)).thenReturn(5);
 when(mockDataAccess.getHeader(4)).thenReturn(1813699);
 when(mockDataAccess.getHeader(8)).thenReturn(300);

 // Act
 boolean actual = receivingObject.loadExisting();

 // Assert
 ...
```

Listing 11: The *Arrange* and *Act* phases of the RICK tests for `loadExisting`

```
 assertEquals(true, actual);
}
```

Listing 12: The generated **OO** for `loadExisting`

```
 verify(mockDataAccess, atLeastOnce()).loadExisting();
 verify(mockDataAccess, atLeastOnce()).getHeader(0);
 verify(mockDataAccess, atLeastOnce()).getHeader(4);
 verify(mockDataAccess, atLeastOnce()).getHeader(8);
}
```

Listing 13: The generated **PO** for `loadExisting`

```
 InOrder orderVerifier = inOrder(mockDataAccess);
 orderVerifier.verify(mockDataAccess, times(1)).loadExisting();
 orderVerifier.verify(mockDataAccess, times(3))
      .getHeader(anyInt());
}
```

Listing 14: The generated **CO** for `loadExisting`

parameters, causing the **PO** test to fail.

We now discuss cases where mutants are not killed by RICK tests. First, a mutant will be undetected if it results in a behavior that is equivalent to that of the original MUT. This is a well-known limitation of mutation analysis [41]. Second, a mutant will be undetected if the production input cannot infect the program's state, or if a mock-based oracle does not capture its side-effects. The challenges of producing test cases that can effectively infect, propagate and observe the effects of a mutant are well known in the literature [42], [43]. An interesting prospect for future work is to trigger the detection of alive covered mutants through stubbing. For example, we observe this for mutant #1 in `loadExisting` (Listing 10), which is undetected by the **OO** in Listing 12. The mutation causes the MUT to directly return `true`, which is indeed the expected value specified by the **OO**. However, this mutant is killed by **PO** and **CO**, as the mock method calls they specify no longer occur within the MUT. Similarly, the **CO** in Listing 14 does not kill mutants #4 and #5 in Listing 10. The method calls specified by the **CO** still occur with the same frequency and in the expected order, but with different parameters. Parameter verification is not the focus of the **CO**, but the **PO** in Listing 13 detects the mutants and fails.

Answer to RQ4

All three types of oracles are effective at detecting regressions introduced within MUTs. Moreover, 16, 18, and 2 mutants are only killed by **OO**, **PO**, and **CO**, respectively. The RICK tests with mock-based oracles complement each other for regression testing, making them a useful addition to the test suite.

5.5 Results for RQ5 [Quality]

Per the protocol described in subsection 4.4, we have interviewed 5 developers between June and July, 2022, with the goal of assessing their opinion on the tests generated by RICK. Table 6 presents the details of the participants of the survey. The five developers work in different sectors of the IT industry, and have between 6 and 30 years of experience with software development. Notably, participant P4 is a core contributor to GRAPHHOPPER, with the highest number of commits to its GitHub repository in the last 5 years. From Table 6, we see that all participants write tests sometimes or everyday, and most of them also define and use mocks. The developers also work with diverse programming environments. P1, P2, and P4 work with Java testing and mocking frameworks, specifically JUnit and Mockito. P3 is a Python developer, while P5, who works in a game development company, works mostly with C, C#, and .NET framework.

We begin each meeting by introducing the concepts and terminology of mock-based testing, the RICK pipeline, and our experiments with GRAPHHOPPER. We demonstrate the features of the tests generated by RICK by selecting a total of 6 generated tests, one for each of the three mock-based oracles for 2 MUTs defined in GRAPHHOPPER. We select the first two MUTs from Table 3 that meet the selection criteria mentioned in subsection 4.4. The two selected MUTs, MUT#16 and MUT#6, have 58 and 13 LOC, respectively. The tests generated for MUT#16 have two stubs, and a method call on an external parameter mock object. The tests for MUT#6 have one stub and three method calls on a mocked field. Excluding comments, the number of lines of code across the six generated tests is 39 (median 6 lines of code for each test). We introduce the two MUTs, MUT#6 and MUT#16, to the participant, also presenting their source code. We invite them to clone a fork of GRAPHHOPPER which includes the generated tests. During the meeting, we browse through the generated tests with them via screen sharing. Finally, we ask the participant three sets of questions about the generated tests while documenting their responses. These questions relate to *mocking effectiveness*, i.e., how mocks are used within the tests, as well as the *structure* and *understandability* of the generated tests.

Mocking effectiveness: The first set of questions relates to how the mocks are used in the generated tests. Per their answers, all five participants agree that the generated tests for the 2 MUTs represent realistic behavior of GRAPHHOPPER in production, which would be useful for developers. P2 observed that this can *"save the time spent on deciding the combination of inputs and finding corner cases, especially for methods with branches."* P5 added that, according to them, collecting data from production is *"where RICK shines most, since it abstracts away [for developers] the tricky exercise of deciding test inputs and internal states."* Furthermore, we had detailed discussions with the participants about the verification statements in **PO** and **CO** tests. P1, P3, and P5 noted that they contribute differently to the verification of the behavior of the MUT. P5 remarked that while some verification statements may be redundant, they *"can be manually customized by developers"* when generated tests are presented to them. However, P3, who works primarily with Python, a dynamically-typed language, commented that for them *"the verification of the frequency of the mock method calls in the CO tests is useful, but not the `anyInt()` or `anyString()` wildcards to match argument types."* Additionally, P3 and P4 also discussed the stability of these tests with respect to code refactoring, with P4 saying that *"the tests might break in case a developer refactors legacy code. But because they are so detailed, a regression can also be figured out at a very low level."* P5 highlighted an interesting aspect about the human element in software development by sharing that *"while RICK fits exactly an actual problem in the industry, a potential disadvantage is that it can spoil developers who may become incentivized to design without testability in mind. The tests can be automatically produced later when the application is production-ready."*

Structure: Next, we assess the opinion of developers about the structure of the generated tests. All 5 developers appreciated the "Arrange-Act-Assert" pattern [44] that is systematically followed in the generated tests. They note that this pattern makes the structure of the tests clear. P2, who has experimented with other test generation tools, noted that clear structure and intention is important to improve the adoption of automated tools. All the developers mentioned that they typically use the pattern when writing tests, including P5 who added that the structure was *"spot on."* Moreover, P1, P2, and P4 were appreciative of the description generated by RICK for each test, using the `@DisplayName` JUnit annotation, with P2 noting that *"it is rare and useful."*

Understandability: Finally, we question each participant about the understandability of the generated tests. P1 and P2 mentioned that the comments demarcating each phase in the generated tests contribute to their intuitiveness and make their intention clearer, with P5 exclaiming that they make the tests *"super easy to visualize."* Additionally, P2, P4, and P5 noted that the comments are useful, especially since they help understand what is happening within tests generated automatically. However, P4 observed that while the comments *"do not hurt"*, they would not add them while writing the test manually. P3 was also of the opinion that the comments may be removed without impacting the understandability of the tests.

The perception of developers of automatically generated tests and mocks, using data collected from production, is

TABLE 6: Profiles of the developers who participated in the survey to assess the quality of tests generated by RICK

PARTICIPANT	EXPERIENCE (YEARS)	SECTOR	WRITES TESTS	USES MOCKS	PROGRAMMING ENVIRONMENT
P1	6	Consultancy	*"everyday, testing is [my] life"*	sometimes	`JUnit + Mockito`
P2	30	Consultancy	often	sometimes	`JUnit + Mockito`
P3	7	Telecom	TDD practitioner	sometimes	`Python (pytest)`
P4	10	Product (GRAPHHOPPER)	*"all the time"*	sometimes	`JUnit + Mockito`
P5	14	Game development	sometimes	rarely, *"want to mock more"*	`C, C#, .NET`

valuable qualitative feedback about the relevance of RICK. The 5 experienced developers and testers confirm that the data collected in production is realistic and useful to generate tests and mocks. They also appreciate the systematic structure of the test cases, as well as the explicit intention documented in comments.

This qualitative study also suggests the need for further work in the area of automated mock generation. For example, P3 expressed interest in analyzing the influence of the architecture of the system under test on the stability of the mocks. P4 observed that it would be useful to evaluate the overall testability of an application in terms of how many MUTs have mockable method calls. We also discussed with P5 about adding more context and business-awareness to test names and comments to further help developers troubleshoot test executions. We identify these as excellent directions for further work, with much potential for impact on the industry.

Furthermore, from our discussions with the developers, we find that they all agree that mocking is advantageous. However, developers often rely on metrics such as coverage as a proxy for the strength of their test suite. This leads to a methodological mismatch where mocks are desirable, yet do not contribute directly to the strength of the tests, i.e., do not have an impact on test coverage. We note that an implicit prerequisite for developers to be more open to the benefits of using mocks is to consider test quality beyond coverage, to embrace the value of mocking, as well as the effort required to include mocks in their pipeline. Mocking is not trivial and comes with the challenges highlighted in subsection 2.3. RICK can help with realistic mocks, directly available in readable tests, that can capture regressions.

Answer to RQ5

Five experienced developers confirm that the concept of data collection in production is relevant for the generation of tests with mocks. They all appreciate the systematic "Arrange-Act-Assert" template for the test which contributes to the overall good understandability of the tests generated by RICK.

6 DISCUSSION

We now discuss the performance of RICK, the limitations of our approach for mock generation, and the threats to the validity of our findings.

6.1 Performance

Runtime data capturing is a key process within RICK, which requires some additional computation and memory resources. We adapt the methodology used in previous work [15] to measure the performance implications of RICK during the execution of our three case studies. We exercise each application with the workload described in subsection 4.2 in three distinct ways. First, we determine the baseline performance of the application by running it without any agent attached. Next, we run it with the default monitoring agent, i.e., Glowroot, attached (recall that Glowroot is standard monitoring technology used in the industry). Finally, we attach RICK to the application as a Glowroot plugin, which means the complete monitoring plus data collection machinery for the MUT and mockable method invocations. The experiments are performed on a machine running Ubuntu 22.04, with an 8-core Intel i5 processor and 16GB memory. We use the Linux `top` command, filtered on the application name, to obtain its CPU and memory usage.

For GRAPHHOPPER, the average CPU consumption for the baseline execution is 35.4% while the memory usage is 824.7 MB. Attaching Glowroot as a monitoring agent to GRAPHHOPPER increases the CPU and memory consumption to 66% and 983.4 MB. Attaching the complete monitoring and data capturing abilities of RICK results in the CPU and memory usage of 109.2% and 1570.2 MB. This means that the execution with RICK and all MUTs and mockable methods monitored, consumes thrice the CPU compared to baseline, and twice the memory. Next, normal execution of GEPHI consumes 117.6% CPU and 856.5 MB memory on average. Monitoring with Glowroot increases these usages to 142.2% and 1157.8 MB, respectively. Attaching RICK with GEPHI results in 249.1% CPU and 1601.9 MB memory usage. The resource consumption during execution of GEPHI with RICK is about twice the baseline amount. Finally, averaging across 10 executions of PDFBOX, we find that its CPU consumption is 92%, while its memory usage is 63.4 MB. Attaching Glowroot increases CPU and memory consumption to 335.4% and 190.3 MB, respectively. Attaching RICK does not contribute to additional CPU consumption, and leads to an increased memory usage at 428.2 MB (about 6.7 times the baseline). Overall, we note that monitoring contributes to additional resource consumption for all cases, with respect to the baseline. This increases more with RICK as a consequence of dynamic instrumentation and serialization.

Monitoring is an essential component of modern observability, for ensuring smooth operation and diagnosis [45]. RICK leverages monitoring to generate unit tests that reflect production behaviors and detect regressions. However, monitoring comes at a cost, impacting the scalability of RICK, as well as similar approaches for observation-based test generation [15], [27]. We have taken measures to mitigate this issue, designing RICK to be configurable. First, developers configure the target methods to monitor and capture data for, and only those are instrumented. Additionally, they also specify the number of invocations of these methods that must be monitored in production. Being a research prototype, it is clear that the monitoring agent has ample room for performance optimization when productized. Furthermore, as we highlight in subsection 3.4, RICK is only meant to be periodically employed, for every testing campaign when the development team focuses on improving unit tests with production data.

6.2 Subsequent Test Failures

As we demonstrate with RQ4, the potential future failures of a test generated by RICK are meant to be indicative of regressions in the method under test, compared to the current behavior. However, there are two cases when a RICK test would fail in the absence of a regression.

First, for mocked methods that perform non-deterministic actions, such as network calls, the stubs and mock-based oracles generated by RICK reflect the responses observed in the field, by design. For instance, calls that result in valid responses (2XX) or 4XX errors, such as a 404 for a non-existent resource, will lead to the generation of useful tests with RICK. However, RICK will generate a test that may overfit a situation with a 5XX error. Such a response might not be reproduced within the generated test, causing it to fail.

Second, the failure of a RICK-generated test may be the result of a refactoring [46], and not a regression. For example, for a method under test *m*, the call oracle will fail if the order of the mock method calls within *m* is changed, while the semantics of *m* is preserved. Also, the parameter oracle may not hold if the arguments received by at least one mock method call within *m* deviate from their expected values. This can indicate a behavioral change within *m*, but also a refactoring of the stubbed method call. The parameter and call oracles are sensitive to refactoring changes. However, a failing output oracle implies that the output from *m* is unequal to the expected output, signifying a behavioral change in the public API.

Consider lines 1 to 9 of Listing 15, which present the method `getWidthFromFont` of PDFBox (MUT#27 in Table 5). This MUT calls two mock methods on the field `ttf` of type `TrueTypeFont`, `getAdvanceWidth` (line 4) and `getUnitsPerEm` (line 5). The mock-based oracles generated by RICK for `getWidthFromFont` are presented on line 15 (**OO**), lines 22-23 (**PO**), and lines 30-32 (**CO**). Listing 16 shows a refactored version of `getWidthFromFont`, with two semantically-preserving changes: 1) the mock method calls are reordered, i.e., `getUnitsPerEm` occurs first (line 4), and 2) `getAdvanceWidth` is renamed to `calculateAdvanceWidth` (line 5). These changes cause a failure of the **CO** and **PO**, but the **OO** still holds.

```
1 public float getWidthFromFont(int code) {
2   int gid = codeToGID(code);
3   float width = ttf.getAdvanceWidth(gid);
4   float unitsPerEM = ttf.getUnitsPerEm();
5   if (unitsPerEM != 1000) {
6     width *= 1000f / unitsPerEM;
7   }
8   return width;
9 }
10 ........................................................................
11 @Test
12 public void testGetWidthFromFont_OO() {
13   ...
14   // Assert
15   assertEquals(750.0, actual, 0.0);
16 }
17 ........................................................................
18 @Test
19 public void testGetWidthFromFont_PO() {
20   ...
21   // Assert
22   verify(mockTTF, atLeastOnce()).getAdvanceWidth(0);
23   verify(mockTTF, atLeastOnce()).getUnitsPerEm();
24 }
25 ........................................................................
26 @Test
27 public void testGetWidthFromFont_CO() {
28   ...
29   // Assert
30   InOrder ordVerifier = inOrder(mockTTF);
31   ordVerifier.verify(mockTTF,
         times(1)).getAdvanceWidth(anyInt());
32   ordVerifier.verify(mockTTF, times(1)).getUnitsPerEm();
33 }
```

Listing 15: RICK generates tests with the three mock-based oracles for MUT#27 of PDFBox, `getWidthFromFont`. Refactoring `getWidthFromFont` can impact the validity of these oracles.

```
1 // Refactored getWidthFromFont
2 public float getWidthFromFont(int code) {
3   int gid = codeToGID(code);
4   float unitsPerEM = ttf.getUnitsPerEm(); // reordered call
5   float width = ttf.calculateAdvanceWidth(gid); // renamed
        method
6   if (unitsPerEM != 1000) {
7     width *= 1000f / unitsPerEM;
8   }
9   return width;
10 }
```

Listing 16: MUT#27 of PDFBox, `getWidthFromFont`, after refactoring. Relative to Listing 15, its mockable method calls `getAdvanceWidth` and `getUnitsPerEm` are reordered, and the former is renamed to `calculateAdvanceWidth`.

Despite their varying degree of sensitivity, all three mock-based oracles alert the developer of behavioral changes that are introduced in their code, which is also the goal of testing. The impact of code refactoring on mock-based oracles is an important direction for future work.

6.3 Threats & Limitations

Serialization The internal validity of RICK is impacted by technical limitations. For example, instrumentation of some methods may fail [47]. One of the biggest technical challenge is the serialization and deserialization of large and complex objects. This may result in incomplete or unfaithful program states within the generated tests, which do not reflect the ones observed in production.

Application Vs Library A source of threat to the external validity of our findings arises from the software projects we consider for the evaluation of RICK. We make sure that the three projects are 1) complex, and 2) from diverse domains. Indeed, we consider a library, a desktop application, and a backend application. We do not guarantee that our findings hold for applications and libraries in other languages such as TypeScript, or nim. Still, we believe that RICK would not let them down.

Program Evolution Tests and programs co-evolve within software projects [48]–[50]. The tests generated by RICK are no different, they may be impacted by changes in the application code. A refactoring in the source code, such as a modification in the order of method invocations, or a change in the parameters of an existing method, may require changes in the generated tests. However, if these modifications are not semantically relevant, this is tedious, low value work for developers. This limitation is shared by all regression test generation techniques. In this case, developers can always regenerate new tests when significant changes are made to the methods under tests.

7 Related Work

This section presents the literature on mock objects, as well as their automated generation. We also discuss studies about the use of information collected from production for the generation of tests.

7.1 Studies on Mocking

Since mocks were first proposed [1], they have been widely studied [2], [51]. Their use has been analyzed for major platforms, such as Java [19], Python [52], C [53], Scala [54], Android [55], [56], PHP, and JavaScript [57]. These studies highlight the prevalence and practices of defining and using mocks. Some empirical studies analyze more specific aspects about the usage of mocks, such as their definition through developer-written mock classes [18], or mocking frameworks [20], or their use in the replacement of calls to the file system [58]. Xiao *et al.* [31] find that mocking is practiced in 66% of the 264 projects of the Apache Software Foundation. Spadini *et al.* discuss the criteria developers consider when deciding what to mock [3], as well as how these mocks evolve [19]. MockSniffer by Zhu *et al.* [59] uses machine learning models to recommend mocks. Mockingbird by Lockwood *et al.* [60] uses mocks to isolate the code under dynamic analysis from its dependencies. The use of mocks for Test Driven Development [61], modeling [62], and as an educational tool for object-oriented design [63], [64] has also been investigated. These works have been inspirational for us in many respects. Moreover, RICK is designed to generate tests that use Mockito, which is the most popular mocking framework [20], [31], [57], and is itself a subject of study [65]–[68]. However, none of these related works touch upon mocking in the context of production monitoring, specifically generating mocks and mock-based oracles. These are the two key contributions of our work.

7.2 Mock Generation

Several studies propose approaches to automatically generate mocks, albeit not from production executions. For example, search-based test generation can be extended to include mock objects [21], to mock calls to the file system [6] and the network [7]. Symbolic execution may also be used to generate mocks [8], [22], [69], to mock the file system [70], or a database for use in tests [71]. Honfi and Micskei [72] generate mocks to replace external interactions with calls to a parameterized sandbox. This sandbox receives inputs from the white-box test generator, Pex [23]. Moles by Halleux and Tillmann [73] also works with Pex to isolate the unit under test from interactions with dependencies by delegating them to alternative implementations. Salva and Blot [74] propose a model-based approach for mock generation. Bhagya *et al.* [75] use machine learning models to mock HTTP services using network traffic data. GenUTest [76] generates JUnit tests and mock aspects by capturing the interactions that occur between objects during the execution of medium-sized Java programs. StubCoder [5] by Zhu *et al.* uses an evolutionary algorithm to generate new stubs and repair incorrect stubs within existing JUnit tests. On the other hand, ARUS [77] utilizes information from the execution of the test suite to detect and remove unnecessary stubbing. Abdi and Demeyer [78] leverage mocking within their proposed test transplantation technique that ports client tests into library test suites. ARTISAN by Gambi *et al.* [79] instruments end-to-end GUI tests of Android applications, in order to carve unit tests that mock classes of the Android framework.

To the best of our knowledge, RICK is the only tool that generates mock-based oracles to verify the behavior of the system under test, per the the production executions, with real usages and real data.

The definition and behavior of mocks can also be extracted from other artifacts. For instance, the design contract of the type being mocked can be used to define the behavior of a mock [80]. Samimi *et al.* [81] propose declarative mocking, an approach that uses constraint solving with executable specifications of the mock method calls. Solms and Marshall [82] extract the behavior of mock objects from interfaces that specify their contract. Wang *et al.* [83] propose an approach to refactor out developer-written subclasses that are used for the purpose of mocking, and replace them with Mockito mocks. Mocks have also been generated in the context of cloud computing [84], such as for the emulation of infrastructure by MockFog [16]. Jacinto *et al.* [85] propose a mock-testing mode for drag-and-drop application development platforms. Contrary to these approaches, RICK monitors applications in production in order to generate mocks. Consequently, the generated tests reflect the behavior of an application with respect to actual user interactions.

The executions of system tests in the existing test suite can also be leveraged to generate mocks. This approach has been used by Saff *et al.* [9] through system test executions, and Fazzini *et al.* [55] to generate mocks for mobile applications. Bragg *et al.* [86] use the test suite of Sketch programs to generate mocks in order to modularize program synthesis. However, system tests are artifacts written by developers, and can therefore suffer from biases that developers have about how the system should behave. In contrast, production executions, where RICK sources its test inputs, are free from these assumptions, and reflect how the system actually behaves under real workloads.

7.3 Capture and Replay

Many studies propose techniques to capture a sequence of events that occur within an executing system, with the goal of replaying it [87]. The premise of these techniques is to replicate the state of the system as it was at a certain point in time. Capture and replay has been successfully applied for the reproduction of crashes [88] and failures [89] that occur in the field. The captured sequence of events leading up to a crash or failure allows for more efficient debugging when replayed offline by developers [90], [91], as well as the evaluation of candidate patches for bugs [92]. Capture and replay can also be used to exercise the same sequence of interactions with an application GUI as was done by end-users [93]. This can be used to analyze the performance of interactive applications [94]. All these existing techniques do not generate an explicit oracle. They instead rely on an implicit oracle, such as the reproduction of a failure. Saff *et al.* [9] carve focused unit tests from system tests. Their technique cannot be applied to production environments without major challenges. In particular, a key challenge that we address with RICK is the serialization of production objects with reasonable overhead. This challenge is also noted by Meta, who propose TestGen for generating observation-based tests for Instagram [27]. This aspect, together with our specific oracles, are fundamentally novel compared to the technique of Saff *et al.* [9]. In addition, their evaluation considers one program, while our evaluation considers three real-world programs exercised with representative field workloads.

RICK is fundamentally different from capture and replay techniques since it generates full-fledged test cases, which include an explicit oracle in an assertion. This essential difference allows us to assess the effectiveness of the generated tests with mutation analysis, which none of the capture and replay techniques do.

7.4 Production-based Oracles

Monitoring an executing application with the goal of generating tests is an effective means of bridging the gap between the developers' understanding of their system, and how it is actually exercised by users [13]. To this end, several studies propose tools that capture runtime information. Thummalapenta *et al.* [95] use execution traces for the generation of parameterized unit tests. Wang and Orso [14] capture the sequence of method executions in the field, and apply symbolic execution to generate tests for untested behavior. Jaygarl *et al.* [96] capture objects from program executions, which can then be used as inputs by other tools for the generation of method sequences. Tiwari *et al.* [15] generate tests for inadequately tested methods using production object states. PRODJ by Wachter *et al.* [47] focus on the readability of the unit tests generated from production data, incorporating the objects captured at runtime as plain Java code. Incoming production requests have also been utilized to produce tests for databases [97] and web applications [26]. RICK leverages this methodology with the novel and specific goal of generating tests with mock-based oracles that verify the interactions between a method and objects of external types, as they occur in production.

8 Conclusion

In this paper, we present RICK, a novel approach for generating tests with mocks, using data captured from the observation of applications executing in production. RICK instruments a set of methods under test, monitors their invocations in production, and captures data about the methods and the mockable method calls. Finally, RICK generates tests using the captured data. The mock-based oracles within the generated tests verify distinct aspects of the interactions of the method under test with the external object, such as the output of the method (**OO**), the parameters with which invocations are made on the external object (**PO**), and the sequence of these invocations (**CO**). Our evaluation with three open-source applications demonstrates that RICK never gives up: It monitors and transforms observed production behavior into concrete tests (RQ1). The data collected from production is expressed within these generated tests as complex receiving objects and parameters for the methods, as well stubs and mock-based oracles (RQ2). When executed, 52.4% of the generated tests successfully mimic the observed production behavior. This means that they recreate the execution context for the method under test, the stubbed behavior is appropriate, and the oracle verifies that the method under test behaves the same way as it did in production (RQ3). The three mock-based oracles can detect regressions within the methods under test, and **OO**, **PO**, and **CO** can complement each other in finding bugs (RQ4). Furthermore, our qualitative survey with professional software developers reveals that the data and oracle extracted from production by RICK are relevant, and that the systematic structure of RICK tests is understandable (RQ5).

Overall, we are the first to demonstrate the feasibility of creating tests with mocks directly from production, in other terms to capture production behavior in isolated tests. Since the generated tests reflect the actual behavior of an application in terms of concrete inputs and oracles, they are valuable for developers to augment manually crafted inputs with ones that are relevant in production.

Our findings open up several opportunities for more research. It would be useful to handle more kinds of interactions of a method under test with its environment, such as all method calls made on an external object within the method under test, in order to achieve further isolation within the generated tests. Future work should also consider different choices of mockable method calls, to support mocking types within dependencies. Additionally, the impact of code refactoring on automatically generated mock-based oracles warrants a detailed analysis.

Acknowledgements

This work has been partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, as well as by the Chains project funded by the Swedish Foundation for Strategic Research (SSF).

References

[1] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: unit testing with mock objects," *Extreme programming examined*, pp. 287–301, 2000.

[2] D. Thomas and A. Hunt, "Mock objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[3] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock? an empirical study on mocking practices," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 402–412.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[5] H. Zhu, L. Wei, V. Terragni, Y. Liu, S.-C. Cheung, J. Wu, Q. Sheng, B. Zhang, and L. Song, "Stubcoder: Automated generation and repair of stub code for mock objects," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–31, 2023.

[6] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 79–90.

[7] ——, "Generating tcp/udp network data for automated unit test generation," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 155–165.

[8] M. Islam and C. Csallner, "Dsc+ mock: A test case+ mock class generator in support of coding against interfaces," in *Proceedings of the Eighth International Workshop on Dynamic Analysis*, 2010, pp. 26–31.

[9] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 114–123.

[10] M. Fowler, "Mocks aren't stubs," https://martinfowler.com/articles/mocksArentStubs.html, date accessed September 1, 2024.

[11] M. Christakis, P. Emmisberger, P. Godefroid, and P. Müller, "A general framework for dynamic stub injection," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 586–596.

[12] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 201–211.

[13] Q. Wang, Y. Brun, and A. Orso, "Behavioral execution comparison: Are tests representative of field behavior?" in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 321–332.

[14] Q. Wang and A. Orso, "Improving testing by mimicking user behavior," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 488–498.

[15] D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, "Production monitoring to improve test suites," *IEEE Transactions on Reliability*, pp. 1–17, 2021.

[16] J. Hasenburg, M. Grambow, and D. Bermbach, "Mockfog 2.0: Automated execution of fog application experiments in the cloud," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2021.

[17] N. E. Beckman, D. Kim, and J. Aldrich, "An empirical study of object protocols in the wild," in *European Conference on Object-Oriented Programming*. Springer, 2011, pp. 2–26.

[18] G. Pereira and A. Hora, "Assessing mock classes: An empirical study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 453–463.

[19] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock objects for testing java systems," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1461–1498, 2019.

[20] S. Mostafa and X. Wang, "An empirical study on the usage of mocking frameworks in software testing," in *2014 14th International Conference on Quality Software*, 2014, pp. 127–132.

[21] A. Arcuri, G. Fraser, and R. Just, "Private API access and functional mocking in automated unit test generation," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST*, 2017, pp. 126–137.

[22] N. Tillmann and W. Schulte, "Mock-object generation with behavior," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 365–368.

[23] N. Tillmann and J. d. Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*. Springer, 2008, pp. 134–153.

[24] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Mseqgen: Object-oriented unit-test generation via mining source code," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 193–202.

[25] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel, "How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1281–1291.

[26] L. Zetterlund, D. Tiwari, M. Monperrus, and B. Baudry, "Harvesting production GraphQL queries to detect schema faults," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 365–376.

[27] N. Alshahwan, M. Harman, A. Marginean, R. Tal, and E. Wang, "Observation-based unit test generation at meta," in *Proceedings of ESEC/FSE*, 2024.

[28] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[29] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks," in *Proceedings of the international AAAI conference on web and social media*, vol. 3, no. 1, 2009, pp. 361–362.

[30] A. M. Memon and M. B. Cohen, "Automated testing of gui applications: models, tools, and controlling flakiness," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1479–1480.

[31] L. Xiao, K. Li, E. Lim, X. Wang, C. Wei, T. Yu, and X. Wang, "An empirical study on the usage of mocking frameworks in apache software foundation," *Available at SSRN 4100265*.

[32] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: A temporal graph-based representation of maven central," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR'19. IEEE Press, 2019, p. 344–348.

[33] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The emergence of software diversity in maven central," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR'19. IEEE Press, 2019, p. 333–343.

[34] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *digital investigation*, vol. 6, pp. S2–S11, 2009.

[35] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[36] A. Parsai, A. Murgia, and S. Demeyer, "LittleDarwin: a feature-rich and extensible mutation testing framework for large and complex java systems," in *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers 7*. Springer, 2017, pp. 148–163.

[37] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "A comprehensive study of pseudo-tested methods," *Empirical Software Engineering*, vol. 24, pp. 1195–1225, 2019.

[38] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Practical mutation testing at scale: A view from google," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900–3912, 2021.

[39] M. Staats, G. Gay, and M. P. E. Heimdahl, "Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the International Conference on Software Engineering*, 2012, pp. 870–880.

[40] G. Gay, M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Automated oracle data selection support," *IEEE Trans. Software Eng.*, vol. 41, no. 11, pp. 1119–1137, 2015.

[41] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, 2013.

[42] H. Du, V. K. Palepu, and J. A. Jones, "Ripples of a mutation—an empirical study of propagation effects in mutation testing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[43] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "Suggestions on test suite improvements with automatic infection and propagation analysis," 2019. [Online]. Available: https://arxiv.org/abs/1909.04770

[44] C. Wei, L. Xiao, T. Yu, X. Chen, X. Wang, S. Wong, and A. Clune, "Automatically Tagging the "AAA" Pattern in Unit Test Cases Using Machine Learning Models," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–3.

[45] L. Maguire, "Automation doesn't work the way we think it does," *IEEE Software*, vol. 41, no. 01, pp. 138–141, 2024.

[46] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[47] J. Wachter, D. Tiwari, M. Monperrus, and B. Baudry, "Serializing java objects in plain code," *arXiv preprint arXiv:2405.11294*, 2024.

[48] S. Shimmi and M. Rahimi, "Leveraging code-test co-evolution patterns for automated test case recommendation," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 65–76.

[49] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 195–204.

[50] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "Untangling spaghetti of evolutions in software histories to identify code and test co-evolutions," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 206–216.

[51] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2004, pp. 236–246.

[52] F. Trautsch and J. Grabowski, "Are there any unit tests? an empirical study on unit testing in open source python projects," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 207–218.

[53] S. Mudduluru, "Investigation of test-driven development based on mock objects for non-oo languages," Master's thesis, Linköping University, Department of Computer and Information Science, The Institute of Technology, 2012.

[54] K. Laufer, J. O'Sullivan, and G. K. Thiruvathukal, "Tests as maintainable assets via auto-generated spies: A case study involving the scala collections library's iterator trait," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, 2019, pp. 17–21.

[55] M. Fazzini, A. Gorla, and A. Orso, "A framework for automated test mocking of mobile apps," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1204–1208.

[56] M. Fazzini, C. Choi, J. M. Copia, G. Lee, Y. Kakehi, A. Gorla, and A. Orso, "Use of test doubles in android testing: An in-depth investigation," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2266–2278.

[57] R. De Almeida, R. M. Da Silva, L. S. Serrano, H. De Souza Campos Junior, and V. De Oliveira Neves, "Mock objects in software testing: An analysis of usage in open-source projects," in *Proceedings of the XXII Brazilian Symposium on Software Quality*, 2023, pp. 72–79.

[58] M. R. Marri, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "An empirical study of testing file-system-dependent software with mock objects," in *2009 ICSE Workshop on Automation of Software Test*. IEEE, 2009, pp. 149–153.

[59] H. Zhu, L. Wei, M. Wen, Y. Liu, S.-C. Cheung, Q. Sheng, and C. Zhou, "MockSniffer: Characterizing and recommending mocking decisions for unit tests," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 436–447.

[60] D. Lockwood, B. Holland, and S. Kothari, "Mockingbird: a framework for enabling targeted dynamic analysis of java programs," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 39–42.

[61] T. Kim, C. Park, and C. Wu, "Mock object models for test driven development," in *Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06)*. IEEE, 2006, pp. 221–228.

[62] J. Stoel, T. van der Storm, and J. Vinju, "Modeling with mocking," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 59–70.

[63] J. Nandigam, V. N. Gudivada, A. Hamou-Lhadj, and Y. Tao, "Interface-based object-oriented design with mock objects," in *2009 Sixth International Conference on Information Technology: New Generations*. IEEE, 2009, pp. 713–718.

[64] J. Nandigam, Y. Tao, V. N. Gudivada, and A. Hamou-Lhadj, "Using mock object frameworks to teach object-oriented design principles," *The Journal of Computing Sciences in Colleges*, vol. 26, no. 1, pp. 40–48, 2010.

[65] G. Gay, "Challenges in using search-based test generation to identify real faults in mockito," in *International Symposium on Search Based Software Engineering*. Springer, 2016, pp. 231–237.

[66] A. J. Turner, D. R. White, and J. H. Drake, "Multi-objective regression test suite minimisation for mockito," in *International Symposium on Search Based Software Engineering*. Springer, 2016, pp. 244–249.

[67] S. Wang, M. Wen, X. Mao, and D. Yang, "Attention please: Consider mockito when evaluating newly proposed automated program repair techniques," in *Proceedings of the Evaluation and Assessment on Software Engineering*, 2019, pp. 260–266.

[68] D. J. Kim, N. Tsantalis, T.-H. P. Chen, and J. Yang, "Studying test annotation maintenance in the wild," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 62–73.

[69] N. Alshahwan, Y. Jia, K. Lakhotia, G. Fraser, D. Shuler, and P. Tonella, "AUTOMOCK: Automated Synthesis of a Mock Environment for Test Case Generation," in *Practical Software Testing : Tool Automation and Human Factors*, ser. Dagstuhl Seminar Proceedings (DagSemProc), vol. 10111, 2010, pp. 1–4.

[70] S. Kong, N. Tillmann, and J. de Halleux, "Automated testing of environment-dependent programs - a case study of modeling the file system for pex," in *2009 Sixth International Conference on Information Technology: New Generations*, 2009, pp. 758–762.

[71] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 289–292.

[72] D. Honfi and Z. Micskei, "Automated isolation for white-box test generation," *Information and Software Technology*, vol. 125, p. 106319, 2020.

[73] J. d. Halleux and N. Tillmann, "Moles: tool-assisted environment isolation with closures," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2010, pp. 253–270.

[74] S. Salva and E. Blot, "Using model learning for the generation of mock components," in *IFIP International Conference on Testing Software and Systems*. Springer, 2020, pp. 3–19.

[75] T. Bhagya, J. Dietrich, and H. Guesgen, "Generating mock skeletons for lightweight web-service testing," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 181–188.

[76] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "GenUTest: a unit test and mock aspect generation tool," *International journal on software tools for technology transfer*, vol. 11, no. 4, pp. 273–290, 2009.

[77] M. Li and M. Fazzini, "Automatically removing unnecessary stubbings from test suites," *arXiv preprint arXiv:2407.20924*, 2024.

[78] M. Abdi and S. Demeyer, "Test transplantation through dynamic test slicing," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2022, pp. 35–39.

[79] A. Gambi, H. Gouni, D. Berreiter, V. Tymofyeyev, and M. Fazzini, "Action-based test carving for android apps," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023, pp. 107–116.

[80] S. J. Galler, A. Maller, and F. Wotawa, "Automatically extracting mock object behavior from design by contract™ specification for test data generation," in *Proceedings of the 5th Workshop on Automation of Software Test*, 2010, pp. 43–50.

[81] H. Samimi, R. Hicks, A. Fogel, and T. Millstein, "Declarative mocking," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 246–256.

[82] F. Solms and L. Marshall, "Contract-based mocking for services-oriented development," in *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, 2016, pp. 1–8.

[83] X. Wang, L. Xiao, T. Yu, A. Woepse, and S. Wong, "An automatic refactoring framework for replacing test-production inheritance by mocking mechanism," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 540–552.

[84] L. Zhang, X. Ma, J. Lu, T. Xie, N. Tillmann, and P. De Halleux, "Environmental modeling for automated cloud application testing," *IEEE software*, vol. 29, no. 2, pp. 30–35, 2011.

[85] A. Jacinto, M. Lourenço, and C. Ferreira, "Test mocks for low-code applications built with outsystems," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2020, pp. 1–5.

[86] N. F. Bragg, J. S. Foster, C. Roux, and A. Solar-Lezama, "Program sketching by automatically generating mocks from tests," in *International Conference on Computer Aided Verification*. Springer, 2021, pp. 808–831.

[87] S. Joshi and A. Orso, "Scarpe: A technique and tool for selective capture and replay of program executions," in *2007 IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 234–243.

[88] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej, "Monitoring user interactions for supporting failure reproduction," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 73–82.

[89] J. Bell, N. Sarda, and G. Kaiser, "Chronicler: Lightweight recording to reproduce field failures," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 362–371.

[90] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 474–484.

[91] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive record/replay for web application debugging," in *Proceedings of the 26th annual ACM symposium on User interface software and technology*, 2013, pp. 473–484.

[92] A. Saieva and G. Kaiser, "Update with care: Testing candidate bug fixes and integrating selective updates through binary rewriting," *Journal of Systems and Software*, vol. 191, p. 111381, 2022.

[93] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jrapture: A capture/replay tool for observation-based testing," in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, 2000, pp. 158–167.

[94] A. Adamoli, D. Zaparanuks, M. Jovic, and M. Hauswirth, "Automated gui performance testing," *Software Quality Journal*, vol. 19, no. 4, pp. 801–839, 2011.

[95] S. Thummalapenta, J. d. Halleux, N. Tillmann, and S. Wadsworth, "Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *International Conference on Tests and Proofs*. Springer, 2010, pp. 77–93.

[96] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "Ocat: object capture-based automated testing," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 159–170.

[97] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee, "Snowtrail: Testing with production queries on a cloud database," in *Proceedings of the Workshop on Testing Database Systems*, 2018, pp. 1–6.