Can ChatGPT Repair Non-Order-Dependent Flaky Tests?

Yang Chen
University of Illinois Urbana-Champaign
yangc9@illinois.edu

Reyhaneh Jabbarvand
University of Illinois Urbana-Champaign
reyhaneh@illinois.edu

ABSTRACT

Regression testing helps developers check whether the latest code changes break software functionality. Flaky tests, which can non-deterministically pass or fail on the same code version, may mislead developers' concerns, resulting in missing some bugs or spending time pinpointing bugs that do not exist. Existing flakiness detection and mitigation techniques have primarily focused on general order-dependent (OD) and implementation-dependent (ID) flaky tests. There is also a dearth of research on *repairing* test flakiness, out of which, mostly have focused on repairing OD flaky tests, and a few have explored repairing a subcategory of non-order-dependent (NOD) flaky tests that are caused by asynchronous waits. As a result, there is a demand for devising techniques to reproduce, detect, and repair NOD flaky tests. Large language models (LLMs) have shown great effectiveness in several programming tasks. To explore the potential of LLMs in addressing NOD flakiness, this paper investigates the possibility of using ChatGPT to repair different categories of NOD flaky tests. Our comprehensive study on 118 from the IDoFT dataset shows that ChatGPT, despite as a leading LLM with notable success in multiple code generation tasks, is ineffective in repairing NOD test flakiness, even by following the best practices for prompt crafting. We investigated the reasons behind the failure of using ChatGPT in repairing NOD tests, which provided us valuable insights about the next step to advance the field of NOD test flakiness repair.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Software Testing, Test Flakiness, Large Language Models

1 INTRODUCTION

Flaky tests can non-deterministically pass or fail when running on the same code version. Typically, developers rely on regression testing results to debug if the code changes bring problems into the code base. However, when test failures are, in fact, due to test flakiness, not a bug in the code, developers waste their time pinpointing a bug that does not exist [21]. Flaky tests can also drastically degrade the quality of regression testing and cause negative impacts on software quality [27, 38].

Previous research characterizes test flakiness into different categories [17, 19, 28, 29, 31, 32]. Specifically, based on whether the test results depend on the test orders in which they are run, they can be categorized into Order-Dependent (OD) and Non-Order-Dependent (NOD) tests [30]. Implementation-dependent (ID) tests are a specific type of NOD tests [4], which are caused by using wrong assumptions of unordered collections to implement specific APIs. There are also several techniques in detecting test flakiness [14, 26, 30, 37, 44, 45, 48, 53, 57, 59], but limited attempts have been made in repairing test flakiness [20, 33, 43, 46, 49]. iFixFlakies [46], iPFlakies [49], and ODRepair [33] are designed to repair OD tests. DexFix [58] is designed to repair ID tests, and TRaF [43] and FaTB [29] repairs one sub-category of NOD tests that are caused by asynchronous waits.

Existing test flakiness repair techniques are all based on program analysis, i.e., implementing domain-specific repair rules. Such techniques can be limited if the test suite has new programming features or various development styles. More importantly, unlike OD flaky tests—caused by polluted shared status between tests—and ID flaky tests—caused by specific API implementations, the root causes of NOD tests are various, such as concurrency issues, platform dependencies, runtime environments, and more. This motivates an approach to look for solutions beyond solely relying on rule-based and program analysis to repair NOD tests.

LLMs have shown great effectiveness in generative tasks related to code, as code synthesis [12, 15, 22–25, 39, 41, 50–52], code translation [41], and program repair [56]. Given their abilities in code synthesis and the fact that NOD tests have several sub-categories under which the instances may have similar patterns, we investigate using GPT-4 [2] to repair NOD tests. Our proposed technique, NODoctor, follows best practices in prompting to provide proper context [41, 47, 55], minimizing the amount of context needed to account for limited context windows [35, 40, 41], and prompting iteratively to make the model learn from the textual execution and compilation error feedback [50]. Our extensive evaluation of NODoctor that involved 118 NOD flaky tests from 11 real-world Java projects shows that the NODoctor can generate plausible patches for only eight tests. Among them, six were false positives, and only two were confirmed to be a correct repair. To the best of our knowledge, we are the first to explore using LLM (GPT-4) for repairing *NOD* test flakiness. We observe that GPT-4 is unable to repair NOD test flakiness, and performed a deep analysis of the cases where GPT-4 were or were not successful in repairing them. We believe that our findings offer valuable insights for future research directions in this domain. Our artifact is available at [1].

FTW '24, April 14, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0558-8/24/04…$15.00
https://doi.org/10.1145/3643656.3643900

2 BACKGROUND

Given that this paper focuses on NOD flaky tests, we dedicate a section to providing a detailed background on NOD tests. NOD tests can fail due to various reasons instead of only depending on the test orders in which they are run. Common reasons include concurrency, asynchronous wait, platform dependency, I/O operations, timing issues, race conditions, resource leak, and network issues [17, 31]. Several research studies have been proposed to study characterizing and detecting [13, 14, 17, 29, 31] and repairing NOD flaky tests [29, 43].

Characterizing and Detecting Test Flakiness. Previous research has shown various reasons contributing to NOD flakiness. Unlike OD tests, which can be observed by controlling test orders in the test suite, NOD tests occur in isolation. They may fail very infrequently. Some NOD tests only show failure under certain circumstances. Detecting NOD flakiness is challenging, particularly when the likelihood of observing flaky behavior is minimal. Researchers previously detected NOD tests by rerunning test suites more than a thousand times [32], which shows that rerunning test suites to detect NOD flakiness consumes much computing resources.

Repairing NOD tests. The first step for repairing NOD tests is to pinpoint the root cause in the test implementation. Previous approaches [29, 43] utilize domain-specific rules to automatically repair specific categories of NOD flakiness. However, these methods may struggle with generalization as NOD tests can have diverse root causes, confirmed by the fact that only a subset of these causes has been explored [17, 32]. Furthermore, NOD tests fail non-deterministically and infrequently, making the localization of test flakiness time-consuming, with an average time of about a day per new flaky test [31].

To illustrate the issue's significance, Figure 1 shows a NOD test previously fixed by researchers [6] through manual inspection. The `count` test is flaky due to self-pollution [53]. Specifically, it checks if there is exactly one user in a local file with an email address `bob@example.com` and if the count is not equal to 1, i.e., more users with the same email address exist, the test will fail (Line 30). In the first execution of the test suite, the test passes since the `setUp` adds the user to the database (Line 22), and that user appears in the database only once. However, if we re-execute the test suite, this test fails, as the `setUp` adds the user again to the database. The developer, in fact, has tried to avoid such a situation by deleting the local files before the test execution in the `setUpClass` (Line 4). However, a wrong file is given as an argument, and executing that code does not remove the intended local file.

This flakiness can only be detected through manual inspection of the local file systems and understanding that the file name should be corrected (Line 5), which is hard and time-consuming. In fact, inspecting the files and observing that they have been updated multiple times across multiple test suite executions is the key to the root cause of the test flakiness. On the other hand, looking at the test code and test execution results, developers may not understand the issue.

```
1  @BeforeClass //In abstract class
2  public void setUpClass() throws IOException {
3     FileUtils.delete(new File("target/derbydb"));
4  -  FileUtils.delete(new File("target/lucene"));
5  +  FileUtils.delete(new File("target/lucene3"));
6     AnnotationConfiguration cfg = new AnnotationConfiguration();
7     cfg.addAnnotatedClass(User.class);
8     Properties props = new Properties();
9     InputStream is = SearchQueryTest.class.getResourceAsStream("/
        derby.properties");
10    try {
11        props.load(is);
12     } finally {
13        is.close();
14     }
15    cfg.setProperties(props);
16    sessionFactory = cfg.buildSessionFactory();
17 }
18
19 @Override //In the same test class which includes the flaky test
20 public void setUp() {
21     super.setUp();
22     createUser("Bob", "Stewart", "Smith", "bob@example.com");
23     ...
24 }
25
26 @Test
27 public void count() {
28    BooleanExpression filter =
29    user.emailAddress.eq("bob@example.com");
30    assertEquals(1, query().where(filter).fetchCount());
31 }
```

Figure 1: A NOD flaky test fixed by human in project querydsl [10]

3 APPROACH

Figure 2 shows the overview of NODoctor, which contains *four* components: *Inspector*, *Prompt Generator*, *Repair*, and *Validator*. *Inspector* component takes the original flaky test suite as input to start the repair process. The inspection results will be sent to *Prompt Generator* to further generate prompts based on specific failure location, then the prompts will be sent to *Repair* component. After receiving the response from LLMs, the patch will be sent to *Validator* component to check if the flakiness is indeed resolved. In the remainder of this section, we will explain the details of each component and how they contribute to repairing NOD flakiness.

Inspector. After receiving the original flaky test suite with known flakiness identified by detection tools, the *Inspector* component attempts to reproduce the test flakiness. We utilize NonDex [4] to re-execute the test *five times* for reproducing NOD tests. It generates two outputs for the *Prompt Generator* component: (1) test execution results, consisting of specific error messages and the category into *Test Pass*, *Test Failure*, and *Compilation error*; and (2) pertinent code related to test failures. The code includes the flaky test method, helper methods (e.g., `setUp` and `tearDown`), and other custom-defined methods within the same test class that are invoked in the test body. These two outputs proceed to the next stage, the *Prompt Generator*, where key messages are extracted and combined into prompts, as shown in *Related Code* and *Failure Location* sections of the prompt in Figure 3.

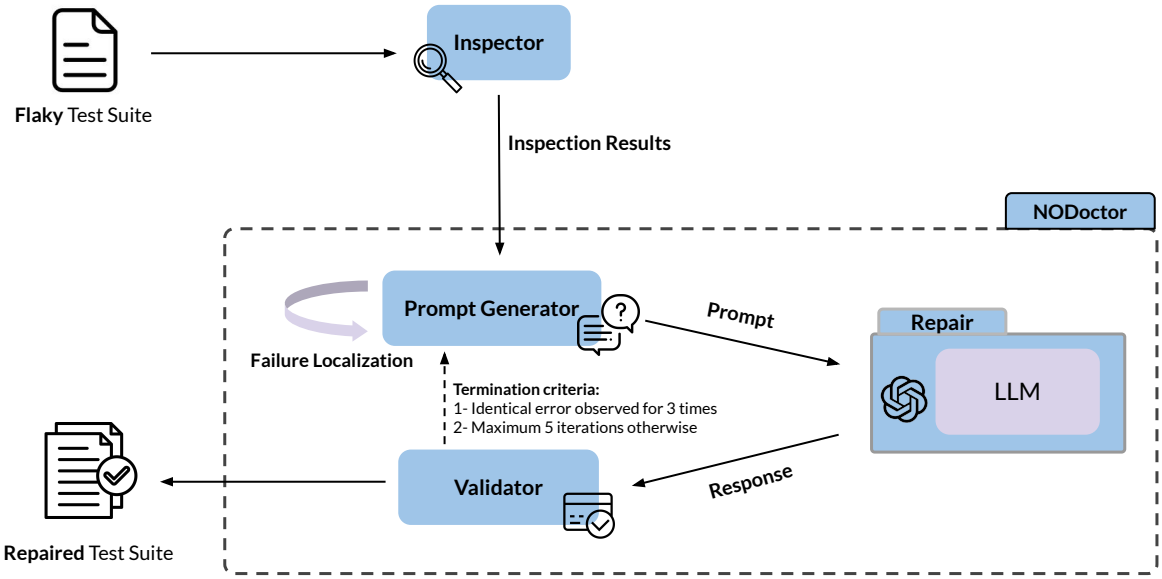

Figure 2: Overview of NODoctor for repairing NOD test flakiness

Prompt Generator. A prompt consists of *five* sections: *Instuction*, *Problem Definition*, *Related Code*, *Failure Location* and *Rules*. Figure 3 shows a prompt template filled with information to repair a NOD test. The prompt initiates with a natural language directive, instructing the LLM to address test flakiness. In cases where the LLM is instruction-tuned, the prompt guides it to assume the role of a software testing expert, thereby enhancing the likelihood of a higher-quality response [7]. Tailored to the specific type of flakiness to be addressed, i.e., fixing NOD flaky tests, the instructions furnish additional details about the flakiness type and offer general advice on fixing it.

Following this, the prompt presents the problem to be resolved: repairing flakiness by listing the names of the involved tests and presenting their corresponding source code. To provide enough context for LLMs to understand the problem, we will provide *test method code, global variables in test class,custom-defined methods called in the test body within the same test class, other helper methods (such as* `setUp` *and* `tearDown`*)* as *Related Code* to LLMs. This comprehensive information is provided to LLMs as we have observed that root causes can sometimes extend beyond the test method itself to other methods like `setUp` within the same test class.

Next, we need to provide the *Failure Location* information. The localization of the specific line that failed is extracted by the *Prompt Generator* through the analysis of test failure results from the *Inspector* component. Recognizing the challenge of repairing without bug localization information, the prompt emphasizes method-level localization and removing irrelevant methods from the context. This assists the LLMs in focusing on modifying the code in the correct place. Such statement-level localization guides the LLMs, indicating where to change and offering hints about potential solutions. In contrast to other methods that provide the entire stack trace or failure report to the model [16, 41, 50], NODoctor meticulously analyzes the stack trace to pinpoint the lines responsible for the errors precisely. For instance, from a relatively big test failure report with usually around 1,000 lines, we extract information about which assertions in the test class have failed and include only such relevant details (with only one or two lines). While LLMs are expected to extract such information independently, this approach minimizes the contextual load on them, producing more accurate responses.

Finally, the prompt concludes by outlining a set of rules for the LLMs to follow while attempting to repair the flakiness. These rules encompass thinking step by step to enable an implicit chain of thoughts (CoT), updating the import list if necessary, and generating syntactically correct code. Additionally, formatting rules, such as enclosing the response between <code></code> tags, are included to facilitate response processing. Once the Prompt Generator component finalizes the prompts, they will be sent to LLMs for patch generation.

Repair. Once the prompts are generated, they are forwarded to the LLMs within the *Repair* component. The current implementation of NODoctor employs GPT-4 as the LLM for two main reasons: (1) GPT-4 has demonstrated superior performance over other commercial or open-source models in various research studies [18, 35, 41]. Since our goal is to advance the state of NOD flakiness repair, choosing the best model is the most reasonable option; and (2) GPT-4 offers greater accessibility with the longest context window compared to other models. Loading LLMs such as Codegen-16GB [39] and StarCoder [34] requires the availability of high-end GPUs that not everyone can afford. Besides, GPT-4 has the longest context window compared to other models (8192 tokens of GPT-4 compared to 2048 tokens of CodeGen-16B and StarCoder). Our Tasks related to repairing flaky tests need long prompts.

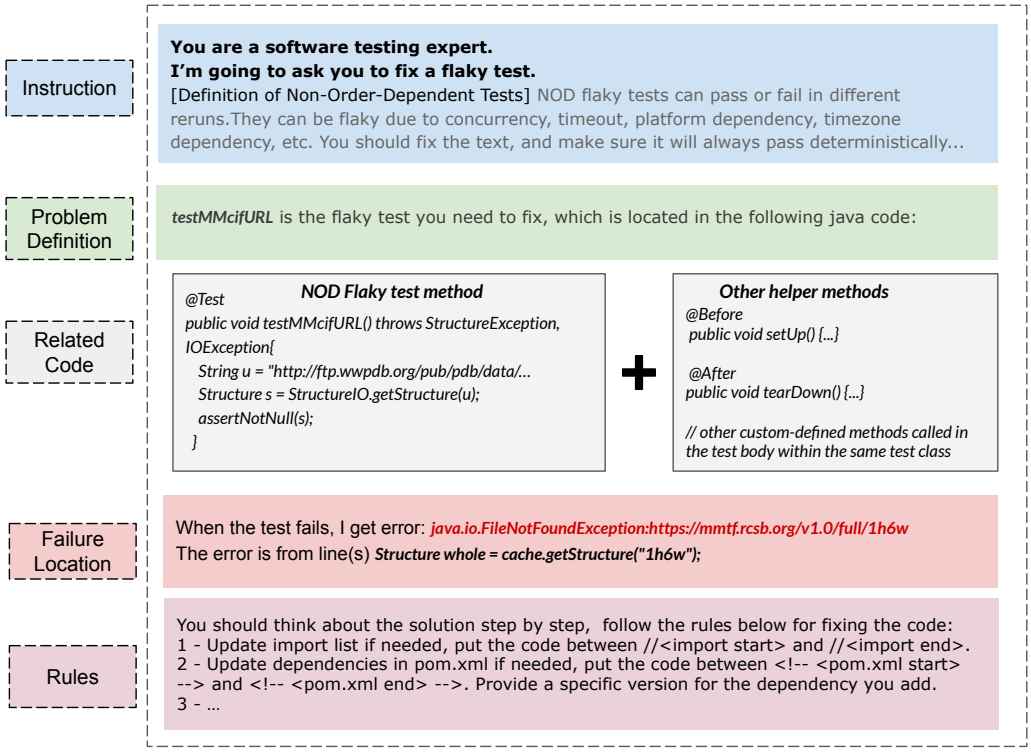

Figure 3: NOD Prompt template

Validator. The patch from *Repair* component is then passed to the *Validator* component, which examines whether it effectively addresses the flakiness. If validated by rerunning NonDex, the patch is recognized as a successful resolution. In cases where the patch fails validation, the patches generated from the current response will undergo further processing in the *Prompt Generator* component, incorporating the latest compilation or test execution outputs. A new prompt with updated information is generated for another iteration of the repair process.

Feedback Loop. LLMs may not effectively address flakiness with just one round of prompting, inspiring us to re-issue requests for improved results. The process of iteratively repairing a flaky test is repeated up to five times. NODoctor streamlines the feedback loop by terminating it sooner if it detects identical compilation or test failure errors in three consecutive prompting rounds or successfully generates a valid patch. At the end of each iteration, the *Prompt Generator* component takes compilation errors or test failures as inputs, enhances the previous prompt with this information, and prompts the LLM once again.

4 EVALUATION

4.1 Experimental Setup

We collected experimental NOD flaky tests from IDoFT [3], which contains more than 5,490 flaky tests of multiple categories detected in real-world Java projects. Starting from 662 NOD tests from 45 projects, we exclude projects and tests that: (1) were deleted, (2) we were unable to compile in a reasonable time due to non-trivial issues such as deprecated dependencies, and (3) we were unable to reproduce NOD flakiness even after running NonDex in *five* times. Consequently, we narrowed down our selection to 118 NOD flaky tests from 11 projects, which served as the subjects for our experiments.

4.2 Effectiveness in Repairing NOD Test Flakiness

Table 1 presents the subjects used in our experiments along with the evaluation results. The sub-columns PF and PU represent the number of *previously fixed* and *previously unfixed* tests, respectively. The sub-column FP displays the number of *false positives*, indicating patches that passed the validation phase but were later found, upon manual inspection, to be semantically incorrect fixes. From the results, we can see that NODoctor was only able to repair two out of a total of 118 NOD tests. Excluding six tests identified as false positives, the remaining 110 tests could not be repaired. In this section, we will discuss detailed examples of tests in each category.

4.2.1 NOD tests that are successfully repaired. Figure 4 shows a successful repair of `testMMcifURL` by NODoctor. In Line 3, the string `u` refers to the file resource `4nwr-assembly1.cif.gz`, accessed in Line 11. Due to the unavailability of the `4nwr-assembly1.cif.gz` resource, the original test encountered a *FileNotFoundException* error when querying it in Line 11. Consequently, errors occur when attempting to assert its non-null status in Line 12. NODoctor generated a patch to address this issue by encapsulating the assertion within `try-catch` blocks. These blocks first verify the file's existence through `http` requests, send a `GET` to establish the connection,

Table 1: Effectiveness of NODoctor in repairing NOD flakiness. PF: Previously Fixed NOD tests (excluding False Positives); PU: Previously Unfixed NOD tests (excluding False Positives); FP: False Positive.

Project	# Tests		# Repaired		
	PF	PU	PF	PU	FP
mercury	1	0	0	0	1
alibabacloud-tairjedis-sdk	0	71	0	0	0
wasp	0	34	0	0	0
biojava	0	2	0	**1**	1
one	0	1	0	0	0
querydsl	2	0	0	0	2
tyrus	0	1	0	0	1
admiral	0	1	0	0	1
wildfly-maven-plugin	0	2	0	0	0
fastjson	0	1	0	**1**	0
secor	2	0	0	0	0
Total	5	113	0	**2**	6

and proceed with the assertion only if the file is confirmed to exist, thereby resolving the test failure.

```
@Test
public void testMMcifURL() throws StructureException, IOException {
   String u = "http://ftp.wwpdb.org/pub/pdb/data/biounit/mmCIF/
      divided/nw/4nwr-assembly1.cif.gz";
+  try {
+     URL url = new URL(u);
+     HttpURLConnection connection =
+       (HttpURLConnection) url.openConnection();
+     connection.setRequestMethod("GET");
+     int responseCode = connection.getResponseCode();
+     if(responseCode == HttpURLConnection.HTTP_OK) {
         Structure s = StructureIO.getStructure(u);
         assertNotNull(s);
+      }
+    } catch (IOException e) {
+       fail("The URL is not accessible");
+    }
 }
```

Figure 4: A NOD test successfully repaired by NODoctor in project biojava [8]

Figure 5 shows another NOD test, `test_groovy`, which was successfully repaired by NODoctor. In Line 14 of the original test, an instance (`GroovyObject b`) of `Class B` (which extends `Class A`) is created without setting the `id` property. Consequently, the JSON serialization of `b` in Line 17 triggers an `IllegalArgumentException` with the error message *Comparison method violates its general contract.* This error commonly occurs when using the `JSON.toJSONString()` method to serialize objects, and the objects being compared do not adhere to the contract expected by the sorting algorithm. When employing `JSON.toJSONString()`, the objects being serialized establish a natural order through the `compareTo` method. In this case, the presence of a `null` object for the `id` property causes a comparison operation to handle `null` value incorrectly. This issue results in a comparison logic error, indirectly impacting the serialization process. To resolve the test failure, NODoctor generates a patch to explicitly set the `id` property for the instance `b` before serializing it, effectively resolving the flakiness.

```
@Test
public void test_groovy() throws Exception {
   ClassLoader parent = Thread.currentThread().getContextClassLoader
      ();
   GroovyClassLoader loader = new GroovyClassLoader(parent);
   Class AClass = loader.parseClass("class A {\n" + // " int id\n" +
      // "}");
   GroovyObject a = (GroovyObject) AClass.newInstance();
   a.setProperty("id", 33);
   String textA = JSON.toJSONString(a);
   GroovyObject aa = (GroovyObject) JSON.parseObject(textA, AClass);
   Assert.assertEquals(a.getProperty("id"), aa.getProperty("id"));
   System.out.println(a);
   // Class B, inherited from A
   Class BClass = loader.parseClass("class B extends A {\n" + //"
      String name\n" + // "}");
   GroovyObject b = (GroovyObject) BClass.newInstance();
+  b.setProperty("id", 33);
   b.setProperty("name", "jobs");
   String textB = JSON.toJSONString(b);
   GroovyObject bb = (GroovyObject) JSON.parseObject(textB, BClass);
   Assert.assertEquals(b.getProperty("id"), bb.getProperty("id"));
   ...
}
```

Figure 5: A NOD test successfully repaired by NODoctor in project fastjson [9]

4.2.2 NOD tests that are false positive. In some cases, although the generated patch enables the test to *pass during the validation process*, manual inspection reveals that the corresponding code changes do not effectively address the NOD flakiness. Instead, the patches either (1) modify the logic of assertions triggering the test failures or (2) alter the original test logic. Consequently, we classify such patches as *false positives*. Six tests have been identified as false positives, and we will delve into their examples in this section.

False positive patches alter the assertions that trigger test failures. In *four* of the plausible patches, the code changes only modify or remove the assertions to make the tests pass without actually resolving the flakiness. Figure 6 displays a patch generated for the `count` test. As shown in Figure 1, researchers previously fixed this test, and the issues were rooted in the `setUpClass` method with a wrong file name. However, even though the `setUpClass` method is provided along with the test code to the model, it cannot accurately identify the real root cause. Instead, GPT-4 modifies the assertion `assertEquals(1,query().where(filter).fetchCount());` to `assertTrue(count >= 1);`. Since the query result will always be greater than 1, the new assertion allows the test to pass but does not resolve the underlying flakiness. This limitation is because, while test execution results are provided to the prompt, the changes pertain to local file systems, and the content of local configuration files is not disclosed to GPT-4. Therefore, it is challenging for GPT-4 to pinpoint the root cause accurately.

False positive patches change the original test logic. *Two* tests are associated with patches that modify the original test logic. Figure 7 presents an example in this category, where the test `testGetStructureForDomain1` attempts to download a structure from a string in line 4, which may not always be available.

```
@Test
public void count() {
+  synchronized(this) {
       BooleanExpression filter = user.emailAddress.eq("bob@example.
       com");
+      long count = query().where(filter).fetchCount();
-      assertEquals(1, query().where(filter).fetchCount());
+      assertTrue(count >= 1);
+  }
}
```

Figure 6: A False positive patch generated by NODoctor in project querydsl [10], which alters test assertions

Consequently, querying this structure encounters an exception of *FileNotFoundException*. To overcome this, NODoctor mocks the `AtomCache` by returning a predefined structure instead of attempting to download it. While this modification enables the test to pass without relying on the availability of the URL, it fails to address the underlying issues and changes the original test logic.

```
@Test
public void testGetStructureForDomain1() throws IOException,
      StructureException {
-   String ranges = "A:328-396,B:518-527";
-   Structure whole = cache.getStructure("1h6w");
-   AtomPositionMap map = new AtomPositionMap(StructureTools
-     .getAllAtomArray(whole), AtomPositionMap.ANYTHING_MATCHER);
-   List<ResidueRangeAndLength> rrs = ResidueRangeAndLength
-     .parseMultiple(ranges, map);
-   int expectedLengthA = rrs.get(0).getLength();
-   int expectedLengthB = rrs.get(1).getLength();
-   Structure structure = cache.getStructureForDomain("d1h6w.2");
-   assertEquals(2, structure.getPolyChains().size());
-   Chain a = structure.getPolyChainByPDB("A");
-   Chain b = structure.getPolyChainByPDB("B");
-   assertEquals(expectedLengthA, a.getAtomGroups().size());
-   assertEquals(expectedLengthB, b.getAtomGroups().size());

+   AtomCache cache = Mockito.mock(AtomCache.class);
+   Structure structure = Mockito.mock(Structure.class);
+   when(cache.getStructure("1h6w")).thenReturn(structure);
+   Structure whole = cache.getStructure("1h6w");
+   assertNotNull(whole);
}
```

Figure 7: A False positive patch generated by NODoctor in project biojava [8], which changes test logic

4.2.3 NOD tests that cannot be repaired. In addition to the false positive tests and repaired flaky tests, NODoctor was not able to generate a plausible patch for the remaining 110 tests. We divide the reasons for these unsuccessful cases into two primary categories: **(R1) Challenges in locating root causes.** GPT-4 finds it challenging to identify the actual root causes of NOD tests. NOD test failures can come from factors beyond test orders, such as concurrency, deadlock, etc. These issues are inherently difficult to debug. While test execution results are provided as feedback for GPT-4, the lack of sufficient local system dynamic information still presents challenges in achieving effective fixes. **(R2) Struggles in providing effective fixes.** GPT-4 faces difficulties in generating effective fixes for NOD test issues. Sometimes, the model may successfully identify the root cause of NOD flakiness but cannot generate correct code changes to address the issues. Alternatively, it might only eliminate specific assertions to ensure test passage, thereby altering the test logic.

Figure 8 shows an example in which GPT-4 fails to locate the root cause **(R1)**. Test `testDelimitedTextFileWriter` can fail due to the conflict dependency issues of third-party framework `Mockito`, which is used in the method `mockDelimitedTextFileWriter` (called at line 4) for mocking the FileSystem class. The underlying issue in the mocking framework leads to an *IllegalStateException* being thrown at Lines 22 and 23 in the `mockDelimitedTextFileWriter` method when attempting to mock the class. The patch used by developers [5] involves replacing `Mockito` with `PowerMockito` to stub the filesystem. However, NODoctor struggles to identify such complex root causes in external libraries. Instead, it proposes that the `FileSystem` class should not be directly mocked; rather, the `FileSystem.get()` method should be used to obtain an instance of the FileSystem and then employ that instance in the test. Consequently, GPT-4 fails to pinpoint the actual root cause of such tests.

```
@Test
public void testDelimitedTextFileWriter() throws Exception {
   setupDelimitedTextFileWriterConfig();
   mockDelimitedTextFileWriter(false);
   ...
}

private void mockDelimitedTextFileWriter(boolean isCompressed)
throws Exception {
-  PowerMockito.mockStatic(FileSystem.class);
+  Configuration conf = new Configuration();
+  FileSystem fs = FileSystem.get(conf);
+  PowerMockito.spy(FileSystem.class);
    FileSystem fs = Mockito.mock(FileSystem.class);
-  Mockito.when(
-      FileSystem.get(Mockito.any(URI.class),
-  Mockito.any(Configuration.class))).thenReturn(fs);
+  PowerMockito.doReturn(fs).when(FileSystem.class,
+     "get", Mockito.any(URI.class),
+     Mockito.any(Configuration.class));
  ...
  Mockito.when(fs.open(fsPath)).thenReturn(fileInputStream);
  Mockito.when(fs.create(fsPath)).thenReturn(fileOutputStream);
  ...
}
```

Figure 8: A Test can not be repaired by NODoctor in alibabacloud-tairjedis-sdk wasp [7], in which GPT-4 can not locate root cause

Figure 9 shows another test that NODoctor fails to repair. For this test, the NODoctor can identify the root cause but struggles to generate an effective patch **(R2)**. The `testUpdatePK` test fails not due to code within the test body but due to incorrect setup in a helper method, `setUpBeforeClass`. In Line 7, a *NullPointerException* occurs when `TEST_UTIL` trying to start a mini cluster, indicating that the `WaspTestingUtility` class or its dependencies

are not properly initialized when TEST_UTIL is created. NODoctor manages to locate the root cause and attempts to initialize TEST_UTIL in the setup, but it fails to generate an effective patch. The generated patch tries to create a new WaspTestingUtility, as shown in Figure 9, but it does not resolve the issues. Additionally, during the iterations of the feedback loop, sometimes NODoctor either addresses partial compilation issues or introduces new compilation errors. Consequently, it may not produce a successful patch.

```
public void setUpBeforeClass() throws Exception {
   WaspTestingUtility.adjustLogLevel();
+  if(TEST_UTIL == null){
+     TEST_UTIL = new WaspTestingUtility();
+  }
   TEST_UTIL.getConfiguration().setInt("wasp.client.retries.number",
       3);
   TEST_UTIL.startMiniCluster(3);
   TableSchemaCacheReader.getInstance(TEST_UTIL.getConfiguration()).
      clearCache();
   TEST_UTIL.createTable(TABLE);
   TEST_UTIL.getWaspAdmin().disableTable(TABLE);
   ...
}
@Test
public void testUpdatePK() throws IOException {
   Map<String, String> sqlGroup = new HashMap<String, String>();
   sqlGroup.put(INSERT, "Insert into " + TABLE_NAME + " (column1,
      column2,column3) values (5,234,'abc');");
   ...
}
```

Figure 9: A Test cannot be repaired by NODoctor in project wasp [11], in which GPT-4 can locate the root cause but fail to generate a correct fix

5 RELATED WORK

Numerous empirical studies emphasize the significance of the test flakiness problem from a developer's standpoint[21, 36]. Various techniques have been proposed to characterize [19, 28, 29, 31, 32, 42], detect [14, 26, 30, 37, 44, 45, 48, 49, 54, 57, 59], or repair [20, 33, 43, 46, 49, 53] test flakiness. iFixFlakies [46] and iPFlakies [49], address the repair of OD test flakiness in Java and Python test suites, respectively. iFixFlakies focuses on OD tests, utilizing information about the test order and leveraging iDFlakies for required inputs. It modifies the execution order of test sub-sequences to identify tests that impact shared states, generating patches accordingly. iPFlakies follows similar steps but is limited to repairing victim OD tests in Python test suites. ODRepair [32] addresses the limitation of iFixFlakies, which relies on cleaner tests to repair victim OD tests. It analyzes static fields and serialized heap states to identify polluted shared states between victim and polluter tests. By enforcing the execution of cleaner tests before victims, ODRepair resolves test flakiness. DexFix [58] is a tool for repairing ID flakiness by implementing domain-specific repair strategies, effectively repairing 119 out of 275 ID flaky tests. However, these strategies are tailored to specific flaky tests and may not generalize.

Two research projects have proposed techniques to repair NOD flaky tests caused by asynchronous waits. FaTB [29] identifies method calls in the test code associated with timeouts or thread waits. It subsequently computes the flaky-test-failure rate, representing the frequency at which the flaky test is expected to fail. Leveraging this rate, FaTB explores different time values and provides developers with the minimum time values they should consider based on their tolerance for flaky test failures. TRaF [43] addresses test flakiness in JavaScript test suites of web-based applications by adjusting waiting times for asynchronous calls to break time dependencies between tests.

None of the prior techniques aimed to devise a general-purpose NOD test flakiness repair approach. Also, while LLMs have been explored in repairing code in general, we are the first to explore the use of GPT-4 in repairing test flakiness.

6 FUTURE WORK

The capacity for in-depth analysis of NOD flaky tests remains constrained and localizing NOD flakiness and validating corresponding patches continue to pose open challenges. In our upcoming research, our goal is to explore efficient methods for categorizing, detecting, reproducing, and repairing NOD tests without requiring extensive resources for rerunning test suites. Regarding categorization, while several related works have attempted to understand different root causes for NOD test flakiness, there is no work on deeply analyzing instances of each NOD sub-category to identify similarities and commonalities between them. Such insight would be the first step in detecting and ultimately repairing NOD test flakiness. The natural next step would be enabling a reliable reproduction of NOD flaky tests. Relying just on re-execution is naive, and we need to work on sort of guarantees to judge the existence (or non-existence) of flaky tests in the test suites. With all these insights, we can revisit the problem of repairing NOD tests. On a related line of research, we plan to investigate the potential of leveraging LLMs to automatically generate pull requests for patches. This exploration encompasses assessing whether LLMs can (1) streamline unnecessary code changes and (2) effectively articulate the functioning of the patch. Furthermore, the non-deterministic behavior of LLMs shows similarity to flakiness, which remains an open problem for future research.

REFERENCES

[1] 2023. GitHub Repository of NODoctor. https://github.com/Intelligent-CAT-Lab/NODoctor.
[2] 2023. GPT-4 Technical Report. https://cdn.openai.com/papers/gpt-4.pdf.
[3] 2023. International Dataset of Flaky tests. https://github.com/TestingResearchIllinois/idoft.
[4] 2023. Nondex Test Flakiness Detection Tool. https://github.com/TestingResearchIllinois/NonDex.
[5] 2023. PR for test testDelimitedTextFileWriter. https://github.com/pinterest/secor/pull/1687.
[6] 2023. A previously-fixed NOD test. https://github.com/querydsl/querydsl/pull/2658.
[7] 2023. Repository alibabacloud-tairjedis-sdk. https://github.com/alibaba/alibabacloud-tairjedis-sdk.
[8] 2023. Repository biojava. https://github.com/biojava/biojava.
[9] 2023. Repository Fastjson. https://github.com/alibaba/fastjson.
[10] 2023. Repository querydsl. https://github.com/querydsl/querydsl.
[11] 2023. Repository wasp. https://github.com/alibaba/wasp.
[12] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
[13] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*. 1572–1584. https://doi.org/10.1109/ICSE43902.2021.00140

[14] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering*. IEEE, 433–444.

[15] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 18–30.

[16] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[17] Yang Chen, Alperen Yildiz, Darko Marinov, and Reyhaneh Jabbarvand. 2023. Transforming test suites into croissants. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1080–1092.

[18] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861* (2023).

[19] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 211–224.

[20] Saikat Dutta, August Shi, and Sasa Misailovic. 2021. Flex: fixing flaky tests in machine learning projects by updating assertion bounds. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 603–614.

[21] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–840.

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[23] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[24] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[25] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263* (2023).

[26] Tariq M King, Dionny Santiago, Justin Phillips, and Peter J Clarke. 2018. Towards a Bayesian network model for predicting flaky automated tests. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion*. IEEE, 100–107.

[27] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and ranking flaky tests at Apple. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 110–119.

[28] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 101–111.

[29] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1471–1482.

[30] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 312–322.

[31] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *2020 IEEE 31st International Symposium on Software Reliability Engineering*. IEEE, 403–413.

[32] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages* 4 (2020), 1–29.

[33] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing order-dependent flaky tests via test generation. In *Proceedings of the 44th International Conference on Software Engineering*. 1881–1892.

[34] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[35] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).

[36] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 643–653.

[37] Maximiliano A Mascheroni and Emanuel Irrazabal. 2018. Identifying key success factors in stopping flaky tests in automated REST service testing. *Journal of Computer Science and Technology* 18, 02 (2018), e16–e16.

[38] John Micco. 2017. The state of continuous integration testing@ google. (2017).

[39] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[40] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828* (2023).

[41] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the Effectiveness of Large Language Models in Code Translation. *arXiv preprint arXiv:2308.03109* (2023).

[42] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* (2021), 74 pages. https://doi.org/10.1145/3476105

[43] Yu Pei, Jeongju Sohn, Sarra Habchi, and Mike Papadakis. 2023. TRaf: Time-based Repair for Asynchronous Wait Flaky Tests in Web Testing. *arXiv preprint arXiv:2305.08592* (2023).

[44] Suzette Person and Sebastian Elbaum. 2015. Test analysis: Searching for faults in tests (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 149–154.

[45] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the vocabulary of flaky tests?. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 492–502.

[46] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 545–555.

[47] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. 31693–31715.

[48] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. 2021. Know you neighbor: Fast static prediction of test flakiness. *IEEE Access* 9 (2021), 76119–76134.

[49] Ruixin Wang, Yang Chen, and Wing Lam. 2022. iPFlakies: A framework for detecting and fixing python order-dependent flaky tests. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 120–124.

[50] Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji. 2023. LeTI: Learning to Generate from Textual Interactions. *arXiv preprint arXiv:2305.10314* (2023).

[51] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[52] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[53] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting flaky tests via Non-Idempotent-Outcome tests. In *International Conference on Software Engineering*. 1730–1742.

[54] Anjiang Wei, Pu Yi, Tao Xie, Darko Marinov, and Wing Lam. 2021. Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 270–287.

[55] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382* (2023).

[56] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).

[57] Pu Yi, Anjiang Wei, Wing Lam, Tao Xie, and Darko Marinov. 2021. Finding polluter tests using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 46, 3 (2021), 37–41.

[58] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. 2021. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 50–61.

[59] Celal Ziftci and Diego Cavalcanti. 2020. De-flake your tests: Automatically locating root causes of flaky tests in code at google. In *2020 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 736–745.