Neurosymbolic Repair of Test Flakiness

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

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

Abstract

Test flakiness, a non-deterministic behavior of builds irrelevant to code changes, is a major and continuing impediment to delivering reliable software. The very few techniques for the automated repair of test flakiness are specifically crafted to repair either Order-Dependent (OD) or Implementation-Dependent (ID) flakiness. They are also all symbolic approaches, i.e., they leverage program analysis to detect and repair *known test flakiness patterns and root causes*, failing to generalize. To bridge the gap, we propose FLAKYDOCTOR, a neuro-symbolic technique that combines the power of LLMs—generalizability—and program analysis—soundness—to fix different types of test flakiness.

Our extensive evaluation using 873 confirmed flaky tests (332 OD and 541 ID) from 243 real-world projects demonstrates the ability of FLAKYDOCTOR in repairing flakiness, achieving 57% (OD) and 59% (ID) success rate. Comparing to three alternative flakiness repair approaches, FLAKYDOCTOR can repair 8% more ID tests than DexFix, 12% more OD flaky tests than ODRepair, and 17% more OD flaky tests than iFixFlakies. Regardless of underlying LLM, the non-LLM components of FLAKYDOCTOR contribute to 12–31 % of the overall performance, i.e., while part of the FLAKYDOCTOR power is from using LLMs, they are not good enough to repair flaky tests in real-world projects alone. What makes the proposed technique superior to related research on test flakiness mitigation specifically and program repair, in general, is repairing 79 *previously unfixed* flaky tests in *real-world projects*. We opened pull requests for all cases with corresponding patches; 19 of them were accepted and merged at the time of submission.

CCS Concepts

• **Software and its engineering** → **Software maintenance tools**; **Software testing and debugging**.

Keywords

Test Flakiness, Large Language Models, Program Repair

ACM Reference Format:
Yang Chen and Reyhaneh Jabbarvand. 2024. Neurosymbolic Repair of Test Flakiness. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680369

ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680369

1 Introduction

Test flakiness is the problem of observing non-determinism in test execution results without any changes in the code under tests. This phenomenon can drastically impact the effectiveness of regression testing in software products. The root cause of test flakiness is code smells in the test suite. However, developers cannot distinguish if the test failure is due to a bug in the code or flakiness, which can waste the valuable time of developers [21] and computing resources [25, 37] without resolving the underlying issue.

To minimize the negative impact of test flakiness, several techniques have been proposed to characterize, detect, and repair them. Compared to detecting flakiness, there is a dearth of work focusing on their repair. All such techniques repair a specific type of test flakiness. For example, iFixFlakies [44], iPFlakies [47], and ODRepair [32] are all designed to repair Order-Dependent (OD) flakiness, which non-deterministically pass or fail under different test execution orders. DexFix [59] proposes a set of domain-specific strategies to repair Implementation-Dependent (ID) flaky tests, which happen due to unrealistic assumptions about non-ordered collections. TRaF [40] repairs asynchronous waits, a specific type of Non-Order-Dependent (NOD) tests that non-deterministically pass or fail due to concurrency issues or dependency on the execution platform, memory, and time.

Regardless of the flakiness category of interest, all prior techniques are symbolic, [1], i.e., they use human knowledge to devise and implement analytical pattern-based rules for repairing test flakiness. As a result, they cannot generalize to repairing flaky tests with unknown root causes that analytical rules do not implement. More importantly, their abilities are limited due to the potential limitation of underlying program analysis techniques in generalizing to new programming features and various development styles.

Large Language Models (LLMs) are effective in generative programming tasks, making them a natural solution for overcoming the generalizability limitations of fixing flaky tests. However, LLMs also suffer from a series of limitations, namely, (L1) generating (syntactically and semantically) incorrect code [33, 34, 38], (L2) the need for proper context in the prompt to perform reasonably [39, 45, 54], and (L3) limited context window, which makes leveraging them for real-world programs and test suites challenging [34, 38, 39]. To use LLMs for fixing flakiness in real-world problems, one can mitigate these challenges by augmenting LLMs with sound symbolic approaches to resolve syntactic issues and validate the generated code (L1), and extract the *minimum* amount of *relevant context* for the prompt to achieve the best possible result (L2 and L3).

[1]Please note that the keyword symbolic here refers to a general term of symbolic learning in contrast to machine learning and should not be confused with symbolic execution. We refer to combining LLMs and program analysis as a neuro-symbolic approach.

We propose FlakyDoctor, a neuro-symbolic approach that combines the generalizability power of LLMs with the soundness of program analysis for repairing OD and ID flaky tests. FlakyDoctor takes the name and type of flaky test as input (§3.1) and extracts its test code and body of other tests that partnered in crime. It then executes them and localizes the source of flakiness. By including the above information as problem context to generate a prompt (§3.2), it instructs LLMs to create a patch for repairing flakiness (§3.3). If the patch has compilation errors, FlakyDoctor first tries to solve the compilation issues offline and then forwards the patch for validation if resolved (§3.4). It terminates with success if the validation confirms the generated patch resolves the flakiness. Otherwise, FlakyDoctor updates the prompt with *concise* information about unresolved issues and makes subsequent repair attempts (§3.5). Repairing terminates after a fixed number of iterations or when all the flaky tests are repaired. Our notable contributions are:

- **Technique.** To our knowledge, FlakyDoctor is the first technique for repairing more than one category of test flakiness. Prior work focused on repairing one type of test flakiness, OD flaky tests [32, 44, 47] or ID flaky tests [59]. Also, none of the prior techniques has leveraged the power of LLMs in repairing test flakiness. The power of FlakyDoctor is not directly from the underlying LLM: offline fixing of issues and precise flakiness localization by minimizing the amount of feedback using static analysis contributes to 12–31 % of its performance, depending on the underlying LLM. FlakyDoctor is publicly available [2] and works with both API- and open-access LLMs.
- **Evaluation** We comprehensively evaluated the effectiveness of FlakyDoctor in repairing 873 flaky tests from 243 real-world projects. Our empirical results corroborate the ability of FlakyDoctor in repairing 58% of studied flakiness (57% OD and 59% ID) in 103 seconds, on average. Among the correct patches, 79 of them were not previously fixed by developers or any existing automated techniques. We opened PRs for those repaired flaky tests, and 19 of them were accepted and merged by the time of submission. Compared to alternative approaches, FlakyDoctor can repair 8% more ID tests than DexFix, 12% more OD flaky tests than ODRepair, and 17% more OD flaky tests than iFixFlakies.

2 Background and Motivation

Depending on whether changing the execution order plays a part in manifesting test flakiness, prior research categorizes flaky tests into OD and NOD. This section explains these categories with real-world examples, challenges in repairing different types of flaky tests, and why FlakyDoctor could repair flakiness in the examples, while alternative approaches failed.

OD Flaky Tests. Such flakiness occurs when two or more tests in the test suite are coupled through a shared state that the developers do not properly manage, e.g., in `tearDown` or `setUp` methods [60]. Test prioritization [43] or test parallelization [13] can change the execution order of the tests, altering their outcome from pass to fail or vice versa. Tests that change the outcome due to polluted shared status are called *victim* or *brittle* [44]. Victim tests pass when executed alone (but can fail if executed after some other tests), while brittle tests fail when run alone (but can pass when run after some other tests). A test that changes the shared state for the *victim* test

```
1  // OD-Polluter
2  @Test
3  public void assertGetEventTraceRdbConfigurationMap() {
4    Properties properties = new Properties();
5    properties.setProperty(BootstrapEnvironment.
         EVENT_TRACE_RDB_DRIVER, "org.h2.Driver");
6    properties.setProperty(BootstrapEnvironment.EVENT_TRACE_RDB_URL,
         "jdbc:h2:mem:job_event_trace");
7    properties.setProperty(BootstrapEnvironment.
         EVENT_TRACE_RDB_USERNAME, "sa");
8    properties.setProperty(BootstrapEnvironment.
         EVENT_TRACE_RDB_PASSWORD, "password");
9    ReflectionUtils.setFieldValue(bootstrapEnvironment,"properties",
         properties);
10     //...
11 +  ReflectionUtils.setFieldValue(bootstrapEnvironment,
12 +     "properties", new Properties());
13 }
14 // OD-Cleaner (Does not exist in the original test suite and has
       been added for illustration)
15 @Test
16 public void cleaner() {
17     ReflectionUtils.setFieldValue( bootstrapEnvironment, "
       properties", new Properties());
18 }
19 // OD-Victim
20 @Test
21 public void assertWithoutEventTraceRdbConfiguration(){
22     assertFalse(bootstrapEnvironment.getTracingConfiguration().
       isPresent());
23 }
```

Figure 1: Example of a previously unfixed OD flakiness in Elasticjob [9] repaired by FlakyDoctor that cannot be repaired by alternative approaches

is called *polluter*, while the test that changes the shared state for the *brittle* is called *state-setter*. In addition to polluter/victim and state-setter/brittle tests, *cleaners* [44] and *state-unsetters* [18] are also important concepts related to OD flaky tests. When a cleaner test runs between a polluter and a victim, it cleans the polluted state so the victim can pass. Likewise, when a state-unsetter runs between a state-setter and a brittle, it neutralizes the state change impact, and the brittle fails.

Figure 1 shows polluter and victim tests from Elasticjob project [9]. The shared state causing the dependency is the `bootstrapEnvironment`, a global variable in the test class. If the polluter runs before the victim, it will alter the state of `bootstrapEnvironment` (Lines 4–9 in polluter code), which causes the assertion in Line 22 of the victim to fail. Otherwise, the victim test passes. The cleaner—which does not exist in reality and has been added for illustration—neutralizes the impact of polluters by resetting the shared state, resulting in the victim pass.

Repair Challenge. The obvious solution to repair the OD flakiness is to remove the dependency. In the illustrative example of Figure 1, the patch should reset the properties of `bootstrapEnvironment` at the beginning of victim or the end of the polluter. Without the privilege of code synthesis ability of LLMs, prior techniques rely on the existence of cleaners to extract specific statements to clean the pollution and generate patches [44]. To alleviate this need, subsequent techniques automatically generate the cleaner first [32] and use it for patch generation. Those techniques are still limited to the ability of testing techniques to generate correct cleaners.

FlakyDoctor leverages checks the potential shared state/variables between tests (§3.1) and then instructs LLM to modify the code of polluter to clear the polluted state (§3.2). The highlighted line at the end of the polluter (Line 11) shows the patch for this real-world example generated by FlakyDoctor. iFixFlakies could not fix this flakiness due to the absence of cleaners in the test suite. ODRepair detected the shared state successfully but could not generate the cleaner test to use it for repair further.

NOD Flaky Test. NOD flakiness happens due to misuse or misunderstanding of programming APIs, concurrency problems, execution platforms, runtime environment, etc. Compared to OD flakiness, NOD flakiness occurs for each test in isolation and regardless of test execution order. As a result, one can detect or validate the patch by re-executing it without shuffling the test order. Still, detecting or validating the patch for NOD flakiness is challenging when the probability of observing flaky behavior is tiny [18]. A special sub-category of NOD tests is ID flakiness, which occurs due to incorrect assumptions about the non-ordered collections [59]. Prior research [30] has demonstrated that ID tests are more prevalent than other NOD categories. In the IDoFT [6] dataset of real-world flaky tests, ID tests also greatly outnumber other NOD tests.

Figure 2 illustrates an ID flaky test from Hadoop [8], which happens due to converting an unordered collection (a `Json` object) into `String`. This is not problematic unless we assume a specific order for an unordered collection: the assertion (Lines 8–9) checks if the conversion equals to "{"A":6,"B":2,"C":2}", assuming that the string conversions of the same `Json` objects are similar. As a result, the execution of this test non-deterministically passes or fails.

Repair Challenge. The first step in repairing ID tests is understanding the source of non-determinism, i.e., localizing the source of flakiness. The key idea here is that test execution failure can help localize the source of flakiness systematically. To extract such information, FlakyDoctor analyzes the stack trace of the test execution failure and identifies the tests and statements within them that incorporate non-determinism. It then instructs the LLM to focus on specific lines in the culprit tests to repair the issue.

To repair the ID flakiness in Figure 2, the patch generated by FlakyDoctor first transforms the converted `JSon` object (`csv1`) into a `LinkedHashMap` (Line 10–12), and then reconstructs the expected output in the previous assertion as a `LinkedHashMap` (Lines 13–16). Comparing these two objects in the new assertion (Lines 17–18) resolves the flakiness. We suspect the LLM component of FlakyDoctor was able to reason about the return type of `mbs.get ttribute` being `JSon`, based on the format of {"A":6,"B":2,"C":2} in the assertion argument (Line 9). The chance of data leakage is narrow since this test was previously unfixed. The alternative approach for fixing ID tests, DexFix, fails to fix the test as it looks for particular patterns and explicit usages of unordered collections, which do not exist.

In this paper, we only focus on repairing OD and ID flakiness. The main reason is that *reliable* detection of NOD tests in general, hence validating the generated patches, is still an open problem. Previous studies [10, 30] rely on re-executing test suites 10–10,000 times to detect NOD flakiness. Even with this large number of executions, not observing flaky behavior does not mean it does not exist. Flakiness localization and patch validation for general NOD flaky tests still pose open challenges for LLMs to repair them [17].

```
1     @Test // ID flaky test
2     public void testPriority() throws Exception {
3       //...
4       MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
5       ObjectName mxbeanName = new ObjectName(
6        "Hadoop:service="+ namespace + ",name=DecayRpcScheduler");
7       String cvs1 = (String)mbs.get ttribute(mxbeanName,"
         CallVolumeSummary");
8   -   assertTrue("Get expected JMX of CallVolumeSummary before
9   -       decay", cvs1.equals("\{" ":6,\"B\":2,\"C\":2\}"));
10  +   Map<String, Integer> map1 = new Gson().fromJson(
11  +       cvs1, new TypeToken<LinkedHashMap<String, Integer>()
12  +       .getType());
13  +   Map<String, Integer> expectedMap1 = new LinkedHashMap<>();
14  +   expectedMap1.put(" ", 6);
15  +   expectedMap1.put("B", 2);
16  +   expectedMap1.put("C", 2);
17  +   assertEquals("Get expected JMX of CallVolumeSummary before
18  +        decay", expectedMap1, map1);
19    }
```

Figure 2: Example of a previously unfixed ID flakiness in Hadoop [8] repaired by FlakyDoctor that cannot be repaired by alternative approaches

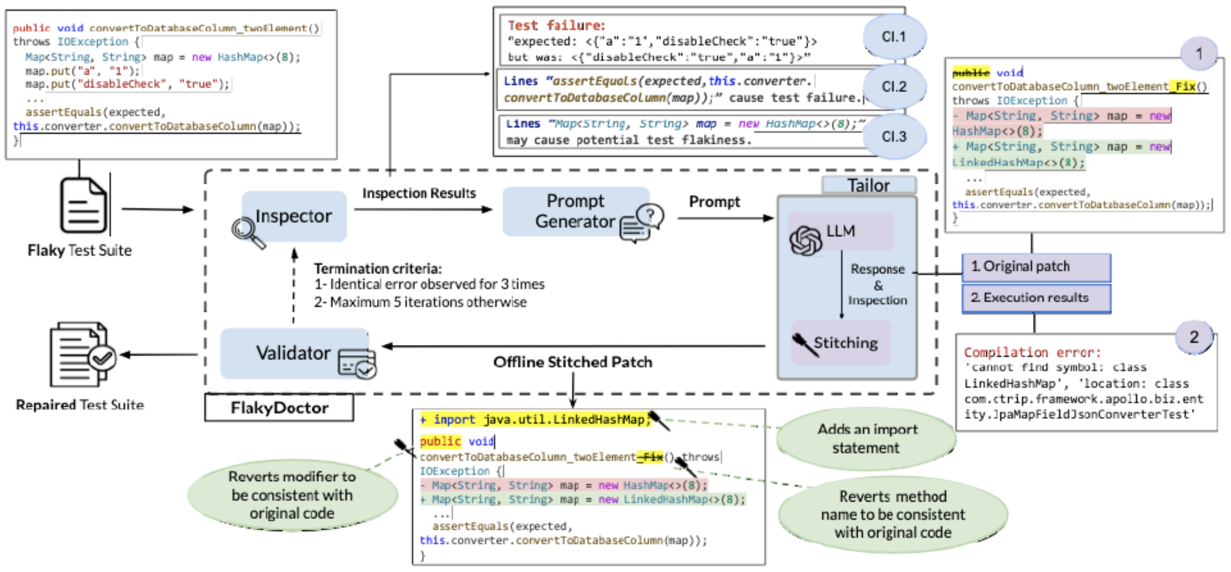
3 FlakyDoctor

Figure 3 shows the overview of FlakyDoctor, consisting of *four* main components, namely, *Inspector*, *Prompt Generator*, *Tailor*, and *Validator*. The *Inspector* takes a flaky test suite as input, analyzes the test execution results, and localizes the source of test failures in the test code. Depending on the type of flakiness, a combination of inspection results, culprit test method(s), and relevant global variables and helper methods in the test class will be used by *Prompt Generator* to create the prompt. The prompt specifically instructs LLM to focus on particular statements and generate the patch by modifying the provided tests, variables, and helpers.

Once the LLM responds to the prompt with a patch, *Tailor* first checks for compilation errors, which are inevitable in the code produced by LLMs. In case of compilation issues in the patch, the *Tailor* tries to resolve them offline in *Stitching* sub-component. If the modified code passes compilation, it goes to the *Validator* to check if it resolves the flakiness. If validated, FlakyDoctor terminates successfully. Otherwise, the incorrect patch generated from the current iteration, along with concise compilation or test execution outputs, goes for another round of inspection and repair. The iterative repair terminates upon generating a successful patch or for a fixed number of iterations. We will explain the details of each component in the remainder of this section.

3.1 *Inspector*

The *Inspector* takes the flaky test suite as input and extracts proper and concise contextual information required to repair the flakiness. If a given test suite contains more than one flaky test, the FlakyDoctor will analyze them individually, and each will be patched separately. *Inspector* should generate three contextual information (CI) for the *Prompt Generator*: (CI.1) test execution errors, (CI.2) corresponding failed assertions, and (CI.3) potential source of flakiness. To that end, it first executes the tests to reproduce the failure based on different types of flakiness. For OD-Victim tests, *Inspector* uses



```
public void convertToDatabaseColumn_twoElement()
throws IOException {
  Map<String, String> map = new HashMap<>(8);
  map.put("a", "1");
  map.put("disableCheck", "true");
  ...
  assertEquals(expected,
this.converter.convertToDatabaseColumn(map));
}
```

```
Test failure:
"expected: <{"a":"1","disableCheck":"true"}>
but was: <{"disableCheck":"true","a":"1"}>"
```

```
Lines "assertEquals(expected,this.converter.
convertToDatabaseColumn(map));" cause test failure.
```

```
Lines "Map<String, String> map = new HashMap<>(8);"
may cause potential test flakiness.
```

```
~~public~~ void
convertToDatabaseColumn_twoElement_Fix()
throws IOException {
- Map<String, String> map = new
HashMap<>(8);
+ Map<String, String> map = new
LinkedHashMap<>(8);
  ...
  assertEquals(expected,
this.converter.convertToDatabaseColumn(map));
}
```

```
Compilation error:
'cannot find symbol: class
LinkedHashMap', 'location: class
com.ctrip.framework.apollo.biz.ent
ity.JpaMapFieldJsonConverterTest'
```

```
+ import java.util.LinkedHashMap;
public void
convertToDatabaseColumn_twoElement~~_Fix~~() throws
IOException {
- Map<String, String> map = new HashMap<>(8);
+ Map<String, String> map = new LinkedHashMap<>(8);
  ...
  assertEquals(expected,
this.converter.convertToDatabaseColumn(map));
}
```

Figure 3: Overview of FlakyDoctor for repairing test flakiness

a modified version of Surefire [1] to specify the execution order of the polluter and victim, i.e., executes the polluter test before the victim to make it fail. For OD-Brittle tests, FlakyDoctor executes them in isolation, as they fail by default. For ID tests, *Inspector* executes them with NonDex [4], which randomly explores different behaviors of certain APIs during test execution through multiple rounds to produce the failure outcome.

After test execution and reproducing the failure, *Inspector* extracts the errors (CI.1) directly from the execution result. In the running example of Figure 3 that shows an ID flaky test `convertToDatabaseColumn_twoElement`, the error message (CI.1) is `expected:{"a":"1","disableCheck":"true"} but was:{"disableCheck":"true","a":"1"}`). By parsing the stack trace, *Inspector* can extract the line number in the test class and get the assert statement causing the failure accordingly (CI.2).

Repairing without problem localization information is searching for a needle in a haystack. *Inspector* employs a method-level localization, i.e., only includes the flaky test methods instead of the entire test suite. It additionally employs the following heuristics to localize the source of flakiness at the statement level as possible:

For ID flaky tests, *Inspector* performs a backward flow-sensitive analysis to pinpoint unordered collections (e.g., `HashMap`) or APIs (e.g., `getFields`) before the failed assertions, which may lead to a non-deterministic order of elements. In the running example, *Inspector* identifies that `Map<String, String> map = new HashMap<>(8)` initializes an unordered collection and returns it as the potential cause of flakiness (CI.3).
For OD-Victim tests, *Inspector* extracts global variables and helper methods such as `setUp` and `tearDown`. Global variables can be potential sources of dependency between tests. Including the helper methods is two-fold: they can be either a source of dependency between tests due to improper management of global variables and resources, or the patch can implement the fix inside them.

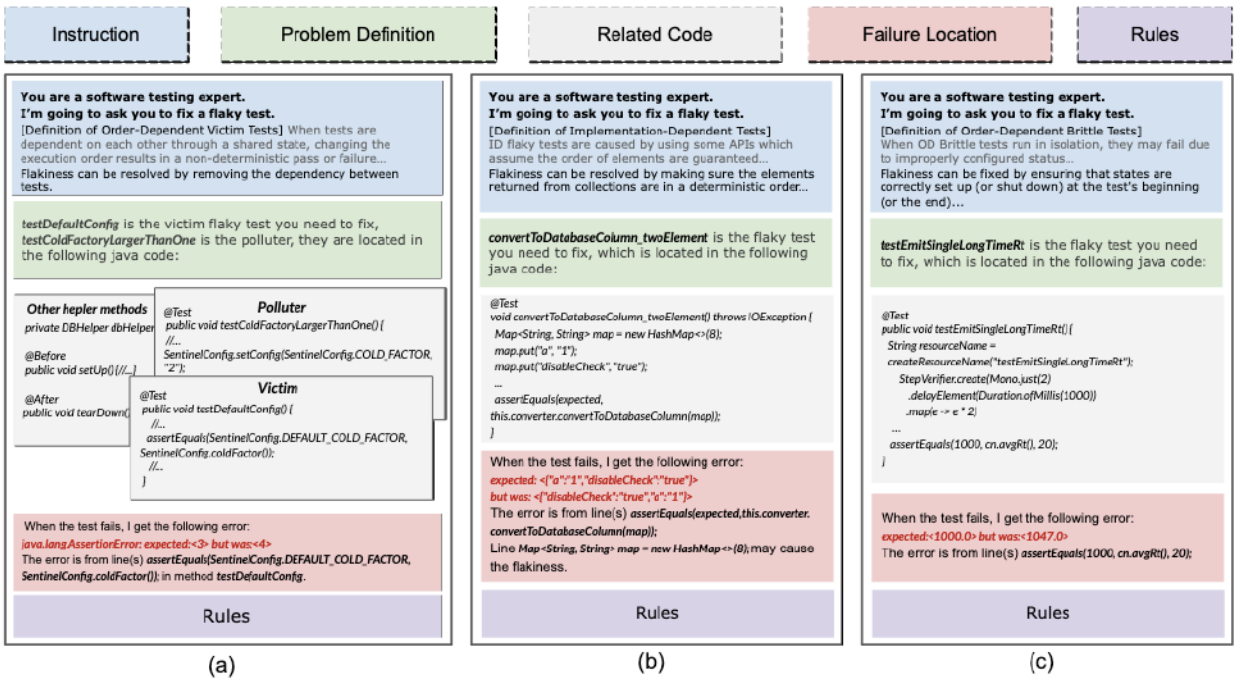
3.2 *Prompt Generator*

FlakyDoctor currently supports fixing ID and OD (OD-Victim and OD-Brittle) flaky tests and has three prompt templates corresponding to each type. Figure 4 shows the templates for OD-Victim (Figure 4a), ID (Figure 4b) and OD-Brittle (Figure 4c) flakiness. The structure of prompt templates is similar, but *Prompt Generator* fills them differently according to flakiness type.

The prompt starts with a natural language instruction, asking the LLM to repair the flaky test (Instruction section). If the LLM is instruction-tuned, the prompt asks it to act as a software testing expert to increase the chance of LLM producing a better response [3]. Depending on the type of flakiness, the instructions provide more specific information and general advice in repairing them.

Next, the prompt introduces the problem that LLM should solve, i.e., repairing flakiness, by listing the names of the tests involved (Problem Definition), followed by relevant source code (Related Code) extracted by *Inspector* (CI.2). For ID and OD-Brittle flaky tests, the Related Code section only includes the flaky test declaration and implementation. For OD-Victim tests, this section includes the code of the victim, polluter, global variables, and helper methods. With this design decision, repairing a polluter or helper methods may resolve several other related flakiness in the test suite. *Prompt Generator* also concatenates statements that raise errors/failures (CI.1) and potential sources of flakiness (CI.2) to the prompt (Failure Location section) to help models localize the flakiness better, and, hopefully, generate a higher-quality patch.

Prompt Generator concludes the prompt with a list of *six* rules for LLM to follow: (1) Solve the problem with implicit Chain-of-Thoughts (CoT) [52], (2) Update the imports and build files if needed, (3) Generate syntactically correct code, (4) Ensure all the arguments are correct, (5) Use compatible types for all variables, and (6) Follow the specified formatting (to facilitate response processing). As we



(a)

You are a software testing expert.
I'm going to ask you to fix a flaky test.
[Definition of Order-Dependent Victim Tests] When tests are dependent on each other through a shared state, changing the execution order results in a non-deterministic pass or failure...
Flakiness can be resolved by removing the dependency between tests.

testDefaultConfig is the victim flaky test you need to fix, *testColdFactoryLargerThanOne* is the polluter, they are located in the following java code:

Other hepler methods
```
private DBHelper dbHelper

@Before
public void setUp() {//...}

@After
public void tearDown(
```

Polluter
```
@Test
public void testColdFactoryLargerThanOne() {
//...
SentinelConfig.setConfig(SentinelConfig.COLD_FACTOR,
"2");
```

Victim
```
@Test
public void testDefaultConfig() {
  //...
  assertEquals(SentinelConfig.DEFAULT_COLD_FACTOR,
SentinelConfig.coldFactor());
  //...
}
```

When the test fails, I get the following error:
java.lang.AssertiorError: expected:<3> but was:<4>
The error is from line(s) ***assertEquals(SentinelConfig.DEFAULT_COLD_FACTOR, SentinelConfig.coldFactor());*** in method ***testDefaultConfig***.

Rules

(b)

You are a software testing expert.
I'm going to ask you to fix a flaky test.
[Definition of Implementation-Dependent Tests]
ID flaky tests are caused by using some APIs which assume the order of elements are guaranteed...
Flakiness can be resolved by making sure the elements returned from collections are in a deterministic order...

convertToDatabaseColumn_twoElement is the flaky test you need to fix, which is located in the following java code:

```
@Test
void convertToDatabaseColumn_twoElement() throws IOException {
 Map<String, String> map = new HashMap<>(8);
 map.put("a", "1");
 map.put("disableCheck", "true");
 ...
 assertEquals(expected,
this.converter.convertToDatabaseColumn(map));
}
```

When the test fails, I get the following error:
expected: <{"a":"1","disableCheck":"true"}>
but was: <{"disableCheck":"true","a":"1"}>
The error is from line(s) ***assertEquals(expected,this.converter.convertToDatabaseColumn(map));***
Line ***Map<String, String> map = new HashMap<>(8);*** may cause the flakiness.

Rules

(c)

You are a software testing expert.
I'm going to ask you to fix a flaky test.
[Definition of Order-Dependent Brittle Tests]
When OD Brittle tests run in isolation, they may fail due to improperly configured status...
Flakiness can be fixed by ensuring that states are correctly set up (or shut down) at the test's beginning (or the end)...

testEmitSingleLongTimeRt is the flaky test you need to fix, which is located in the following java code:

```
@Test
public void testEmitSingleLongTimeRt() {
 String resourceName =
 createResourceName("testEmitSingleLongTimeRt");
   StepVerifier.create(Mono.just(2)
     .delayElement(Duration.ofMillis(1000))
     .map(e -> e * 2)
  ...
  assertEquals(1000, cn.avgRt(), 20);
}
```

When the test fails, I get the following error:
expected:<1000.0> but was:<1047.0>
The error is from line(s) ***assertEquals(1000, cn.avgRt(), 20);***

Rules

Figure 4: Prompt templates for repairing OD-Victim (a), ID (b) and OD-Brittle (c) flaky tests

will show later, including these rules helps only to a limited extent, which requires additional effort to compensate for the subpar performance of LLMs [38].

3.3 *Tailor*

Tailor consists of two sub-components: LLM and *Stitching*. The LLM carries most of the repair burden. FlakyDoctor can work with any LLM with minimal changes in the prompts, and its current implementation uses GPT-4 [3] as an API-access LLM and Magicoder [53] as an open-access LLM[2]. When dealing with real-world code and tests, LLMs' performance can drastically degrade [39]. As an obvious consequence, they generate a code that does not compile, even though being asked during prompting [38].

The ultimate goal of FlakyDoctor is to repair real-world flaky tests, making it vulnerable to this limitation of LLMs. Specifically, without being compilable, passing the patch to the *Validator* component is worthless. As a result, *Stitching* sub-component of *Tailor* attempts to resolve common compilation issues in the generated patch offline. As we will show later (§4.4),

Stitching contributes to 10% and 41% of the correct patches generated by GPT-4 and Magicoder *in the first iteration* (total numbers across all iterations are 12% and 31%).

These numbers are more significant, by a huge margin, than asking LLMs to resolve compilation errors through iterative textual feedback [14, 48]. *Stitching* also reduces the computational cost and carbon footprint by avoiding re-promoting LLMs for fixing trivial or frequent compilation issues. *Stitching* resolves the following issues in the LLM-generated patches systematically using Algorithm 1:

[2]These models have been shown to surpass their equivalent models of the same size in several programming tasks.

Inconsistency with the original code (Lines 3–7). Patches likely differ from the original code in a few statements. Due to non-determinism intrinsic to LLMs, it is possible that the generated code, although implementing the correct repair logic, has such trivial inconsistency issues and cannot be compiled. To check for this, *Stitching* inspects if *modifiers*, *return types*, and *annotations* of the test method(s) in the patch match the original code. If not, it reverts the changes at those places. In the running example of Figure 3, the LLM-generated patch removes the `public` modifier, which prevents the test runner in JUnit4 from executing the method. Thereby, *Stitching* adds the `public` modifier back.

Missing class dependency (Lines 10–17). Adding new code may require importing new dependencies. If a compilation error is related to missing dependencies (i.e., *missing class symbols* error), *Stitching* looks for the missing class in the local JDK specified in the build file of the project and imports the corresponding one that resolves the error. In the patch generated for the running example of Figure 3, LLM replaces `HashMap` with `LinkedHashMap`, but fails to import `java.util.LinkedHashMap`. Based on the error message, *Stitching* looks for the class `LinkedHashMap` and adds the corresponding import to the patch. If multiple classes share the same short name, *Stitching* initially parses all potential classes by matching the short name within the JDK and returns a list that includes all relevant imports. It then traverses the list within a loop. If the first import doesn't resolve the issue, it proceeds to the next until the correct class is imported or the loop ends.

Missing external dependency (Lines 18–20). Some patches require updating the *pom.xml*. For example, the patch for ID flakiness in Figure 2 should not only import `com.google.gson.Gson` and `com.google.gson.reflect.TypeToken` to the test class, but also update *pom.xml* by adding `gson 2.8.6` as a dependency (or rewrite the `artifactId` if the dependency exists).

Conflicting dependencies (Lines 21–25). LLMs may add dependencies that conflict with the existing ones. For example, adding `org.assertj.core.api.Assertions.assertThat` to a test that already imports `org.junit.Assert.assertThat` results in a compilation error due to an ambiguous reference. In such situations, we maintain the original imports and discard the conflicting ones from the patch. Additionally, if the original code imports `org.junit.Assert.*`, simply matching by short name is insufficient. In this scenario, *Stitching* traverses each import node in the patch to determine if its removal resolves the ambiguity, continuing this process until all conflicts are resolved.

Algorithm 1: Stitching Component

```
   Input: Original Related Code $RC$, LLM-generated Code $LC$,
          Compilation Errors $E$
   Output: Stitched Code $SC$
1  foreach $m_i \in LC$ do
2      if $hasError(m_i, E)$ then
3          $T \leftarrow$ getCorrespondingMethod($RC$, $m_i$)
4          $DT \leftarrow$ identifyMethodDeclaration($T$)
5          $DM \leftarrow$ identifyMethodDeclaration($m_i$)
6          if $DT \neq DM$ then
7              $SC \leftarrow$ revert $DM$ in $m_i$ to $DT$
8          $SLib \leftarrow getJavaStandardLibs$
9          foreach $e_i \in E$ do
10             if $isMissingClassSymbol(e_i)$ then
11                 $eSymbol \leftarrow$ extractClassSymbol($e_i$)
12                 $Slib_i \leftarrow$ searchJavaLib($eSymbol, SLib$)
13                 foreach $lib_r \in Slib_i$ do
14                     $P_r \leftarrow$ addImportLib($lib_r, SC$)
15                     $E_r \leftarrow$ compile($P_r$)
16                     if $e_i \notin E_r$ then
17                         $SC \leftarrow P_r$
18             if $packageNotExist(e_i)$ then
19                 $pack \leftarrow$ extractMissingPackage($e_i$)
20                 $SC_{build} \leftarrow$ searchRepository($pack$)
21         $PImports \leftarrow$ getImportStats($m_i$)
22         $TImports \leftarrow$ getImportStats($T$)
23         foreach $PImp_i \in PImports$ do
24             if $isConflictWith(PImp_i, TImports)$ then
25                 $SC \leftarrow$ exclude($SC, PImp_i$)
26 return $SC$
```

3.4 *Validator*

FlakyDoctor can generate plausible patches. However, the final decision of whether the patch resolves test flakiness needs further validation. For OD-Victim tests, *Validator* executes the patched polluter and victim in two different orders (polluter before victim and victim before polluter) using a modified version of Surefire [1]. If the victim passes in both, FlakyDoctor accepts the patch as the ultimate repair. Given that a single polluter (*P*) may pollute multiple victims ($\{V_0, V_1, \ldots V_n\}$), *Validator* also checks whether a patch removes the pollution of other victims ($\{V_1, \ldots V_n\}$). This can reduce the need for additional re-prompts to fix each victim separately, thereby minimizing the costs.

For OD-Brittle, *Validator* executes the patched brittle test and accepts it as an ultimate fix if it passes. We did not use iDFlakies to validate OD tests, mainly due to the non-determinism intrinsic to the implemented algorithm. One threat of validating tests in isolation from other tests is *overfitting* [31]: introducing a new problem when fixing the current one. While the chances of overfitting are narrow in our experiments, we performed a lightweight static analysis check to ensure the shared state between OD tests is unique to them, and no other test in the test suite has such dependency. The *Validator* uses NonDex (configured with *nondexRuns=5* similar to the original paper) to validate ID patches. If NonDex does not mark the patch as flaky, we accept it as the ultimate fix.

Validator categorizes the validation outcomes into three groups: *test pass*, *test failure*, and *compilation error*. A *test pass* indicates the patch successfully resolves the issue, while the last two types indicate the patch from the current response does not fix the flakiness correctly. In such cases, the process will cycle the patch through subsequent iterations in a feedback loop for further refinement.

3.5 *Feedback Loop*

Flaky tests are complex, and LLMs may not repair them with a single round of prompting, motivating the re-prompting of LLMs iteratively. At the end of each iteration, the *Prompt Generator* component takes the compilation errors or test failures as inputs, modifies the Related Code and Failure Location of the previous prompt by adding new contextual information (CI.1–CI.3), and prompts LLM again. One of the core strengths of FlakyDoctor over related work that employs iterative textual feedback to improve LLM performance [14, 39, 48] is trimming down long compilation error or test failure results (sometimes as long as 1000+ lines) to a handful of concise contextual information (§3.1). This will improve the performance of FlakyDoctor, as recent research shows that LLMs provide the best results when given fewer, more relevant contexts rather than larger, unfiltered ones [35]. The iterative repair of a given flaky test repeats five times. However, FlakyDoctor terminates the feedback loop sooner if it observes an identical compilation error in three consecutive rounds or repairs flakiness.

In-context learning [12] may improve the performance of FlakyDoctor. However, including examples results in prompts exceeding the LLMs' context window size in many cases, especially those for repairing OD-Victim tests. Furthermore, FlakyDoctor's prompts are enriched with practical natural language instructions and concise context in the Related Code and Failure Location sections. Given that LLMs inherently understand instructions in natural language better than in-context examples in different modalities [49], the need for in-context examples in FlakyDoctor is negligible.

4 Evaluation

We investigate the following research questions:

RQ1: Effectiveness in fixing Test Flakiness. To what extent FlakyDoctor can repair previous fixed or unfixed OD and ID flakiness in real-world projects?

RQ2: Comparison with Alternative Approaches. To what extent FlakyDoctor can fix flaky tests that alternative approaches cannot? Are there flaky tests that FlakyDoctor cannot fix but other techniques can?

RQ3: Contribution of Different Components. To what extent do error extraction, prompt crafting, stitching, and feedback loop help FlakyDoctor to successfully repair flaky tests?

RQ4: Performance. How much does it take and cost for FlakyDoctor to repair flaky tests?

4.1 Experimental Setup

Alternative Approaches. Prior research focuses on repairing only one type of test flakiness; ODRepair [32] and iFixFlakies [44] repair Java OD tests and iPFlakies [47] repairs Python OD tests. DexFix [59] repairs ID tests. TRaF [40] repairs a special category of NOD tests caused by asynchronous waits. We excluded TRaF from alternative approaches, as FlakyDoctor currently only fixes ID and OD tests. Since most flakiness repair approaches deal with Java unit tests, we excluded iPFlakies from the evaluation. Among the remaining tools, DexFix is not publicly available[3], but their dataset is. As a result, we evaluated FlakyDoctor on their dataset of ID tests without running their tool on additional ID flakiness. For OD flakiness, we compared with both ODRepair and iFixFlakies (ODRepair overcomes the limitations of iFixFlakies by generating cleaner tests, while iFixFlakies can fix tests that ODRepair can not).

Subjects. Alternative approaches come with a dataset of ID (from DexFix) and OD-Victim flaky tests (from ODRepair). We excluded 38 tests from four projects in DexFix dataset and 28 OD-Victim tests from eight projects in ODRepair dataset that we were not able to compile or reproduce the flakiness in a reasonable amount of time, which left us with 237 ID tests and 299 OD-Victim tests.

We further augmented these datasets with flaky tests from IDoFT, a repository of different types of flakiness in real-world projects. The reasons for augmentation are to include (1) OD-Brittle tests, which were not included in the dataset of prior work, and (2) flaky tests that were not previously fixed by human developers or automated flakiness repair techniques. From IDoFT, we excluded the projects that (1) were removed from the repositories mentioned in IDoFT, (2) we were not able to compile with Java 8 or Java 11 due to non-trivial issues such as deprecated dependencies, (3) did not finish compilation in one hour, and (4) we were not able to reproduce their flakiness. The filtering process left us with 193 projects with at least one module, where different modules of the same project may have different flaky tests in the IDoFT dataset. Augmentation, along with the tests from the dataset of alternative approaches, provides us with 541 ID, 299 OD-Victim[4], and 33 OD-Brittle tests from total 243 projects. Among the total of 873 tests, there are 114, 98, and 14, previously unfixed ID, OD-Victim, and OD-Brittle tests.

[3]This was confirmed by the paper's authors.

[4]No additional OD-Victim tests found in the selected projects.

LLMs. FlakyDoctor is designed to work with API- and open-access LLMs. The former does not require the availability of GPU resources and is more accessible to a wider range of users. However, most API-access models, even though negligible, charge for prompting. Open-access LLMs, on the other hand, are free to use, assuming the availability of (non-trivial) GPU resources. Our experiments use GPT-4 [3] and Magicoder [53] as API- and open-access LLMs, given their superiority to alternative models of similar size in code synthesis [3, 53]. LLMs are inherently non-deterministic, which impacts the reproducibility of their results. We believe this is not a threat to the validity of our results: once the synthesized code repairs the flakiness, the problem is considered to be solved. Furthermore, the iterative nature of FlakyDoctor, utilizing sound program analysis as part of the approach, large-scale evaluation on real-world data (repairing 507[5] out of 873 flaky tests), and repairing previously unfixed flaky tests (79 previously unfixed by developers or alternative approaches) increases confidence in the rigor of the technique rather than being luck.

4.2 RQ1: Effectiveness in Repairing Test Flakiness

4.2.1 *Repairing ID Flakiness.*

Table 1 shows the result of running FlakyDoctor and DexFix on subject ID flaky tests. Columns *PF* and *PU* indicate the number of previously fixed and unfixed ID tests. After the automated validation, we manually checked all the repaired patches to ensure the correctness. Such false positives (listed under column *FP*) include deleting assert statements in the patch, surrounding them inside `try/catch` blocks, or replacing the failing assert statement with one that always passes. The reported numbers under PF and PU do not include FP patches.

From these results, we can see that FlakyDoctor-GPT-4 and FlakyDoctor-Magicoder were able to repair 57% (39% previously unfixed) and 16% (9% previously unfixed) ID flaky tests. While Magicoder repairs less tests compared to GPT-4, it can, in fact, repair 6 tests that GPT-4 cannot. As we will show in RQ3, augmenting the power of LLMs with program analysis enables some emerging abilities for smaller open-access models.

We also wanted to see to what extent FlakyDoctor advances state-of-the-art ID flakiness repair technique, DexFix. Given that DexFix is not publicly available, to have a fair comparison, we show the performance of FlakyDoctor for a subset of ID flaky tests in the last four projects that overlap with DexFix dataset inside the parenthesis. Overall, FlakyDoctor-GPT-4 and FlakyDoctor-Magicoder repair 52% and 11% of the ID flaky tests in the DexFix Dataset, while DexFix achieves 44% repair success rate. The repaired ID tests by FlakyDoctor-GPT-4 from the DexFix dataset subsume that of FlakyDoctor-Magicoder.

4.2.2 *Repairing OD Flakiness.*

The numbers under *OD-Victim* column of Table 2 compare the effectiveness of FlakyDoctor with ODRepair and iFixFlakies. *OD-Brittle* column only compares FlakyDoctor and iFixFlakies, as ODRepair cannot fix such flakiness without knowing corresponding state-unsetters (a test that pollutes the state for brittle tests) [18].

[5]We count *unique* tests repaired by two versions of FlakyDoctor.

Table 1: Effectiveness of FlakyDoctor and DexFix in repairing ID flakiness. P: Projects; M: Modules; PF: Previously Fixed; PU: Previously Unfixed; FP: False Positive. Green rows indicate the superiority of FlakyDoctor, and the red row indicates the superiority of DexFix. The white rows belong to augmented tests.

GitHub ID	#P [#M]	#Tests		GPT-4			Magicoder			DexFix
		PF	PU	PF	PU	FP	PF	PU	FP	#Fixed
FasterXML	10 [9]	12	2	**9**	**2**	1	1	0	1	-
SAP	4 [2]	4	1	**2**	0	0	1	0	0	-
IBM	4 [4]	4	0	**3**	0	0	0	0	0	-
adobe	3 [2]	3	1	**2**	0	0	1	0	0	-
DataDog	3 [1]	2	1	**2**	**1**	0	0	0	1	-
oracle	3 [3]	3	0	**1**	0	0	**1**	0	0	-
wildfly	2 [5]	3	2	**1**	0	0	**1**	0	1	-
intel	2 [1]	1	1	**1**	0	0	**1**	0	0	-
networknt	2 [5]	5	0	**3**	0	0	2	0	0	-
gchq	2 [5]	4	2	**4**	0	1	1	0	0	-
opengoofy	2 [3]	4	0	**4**	0	1	2	0	0	-
eclipse-ee4j	2 [6]	1	5	0	**3**	0	0	1	0	-
SP [6]	2 [2]	2	0	**2**	0	0	0	0	0	-
HubSpot	2 [2]	2	0	**2**	0	0	1	0	0	-
twitter	2 [2]	2	0	**1**	0	0	0	0	0	-
dromara	2 [3]	3	0	**2**	0	0	0	0	0	-
GCP [7]	2 [1]	1	1	**1**	0	0	**1**	0	0	-
eBay	2 [2]	1	1	0	**1**	0	0	0	0	-
jdereg	2 [1]	0	2	0	0	0	0	0	1	-
apache	36 [87]	136 (98)	46	**75 (49)**	16	0	17 (5)	0	9	29
square	4 [4]	3 (9)	9	**2 (1)**	2	0	0 (0)	0	0	0
AC [8]	2 [5]	9 (4)	0	**7 (2)**	0	0	4 (1)	0	0	0
intuit	2 [2]	1 (2)	2	**1 (1)**	1	0	0 (0)	0	0	0
Others	114 [143]	182 (91)	37	**117 (51)**	17	3	40 (18)	9	5	50
alibaba	4 [6]	39 (33)	1	25 (20)	1	0	3 (2)	0	0	**25**
Total	215 [306]	427 (237)	114	267 (124)	44	6	77 (26)	10	18	104

Similar to the previous experiment, we manually checked and excluded false positives from the results. FlakyDoctor-GPT-4 can repair 58% (27% previously unfixed) OD-Victim tests. FlakyDoctor-Magicoder repairs 27%, all subsumed by FlakyDoctor-GPT-4. On the other hand, ODRepair and iFixFlakies repair 45% and 40% of OD-Victim tests. To recall, we have to exclude 28 tests from ODRepair and 38 tests from DexFix dataset due to non-trivial deprecated dependencies or non-reproducible flakiness, out of which, ODRepair successfully repairs only five, and DexFix successfully repairs 15. This still makes FlakyDoctor superior to ODRepair and DexFix, given the notable gap in repairing OD-Victim and ID tests. Fixing OD-Brittle is a tie-in competition for FlakyDoctor-GPT-4 and iFixFlakies, FlakyDoctor achieves 51% success rate and iFixFlakies achieves 39%. FlakyDoctor-Magicoder can only repair 9% of the OD-Brittle flaky tests, where one of them could not be fixed by FlakyDoctor-GPT-4.

As demonstrated, FlakyDoctor was able to repair 79 previously unfixed ID and OD flaky tests. We have opened PRs for such fixes, where 19 of them have been accepted and merged by the time of submission [9]. We consider this ability of FlakyDoctor to make it superior to flakiness repair approaches in particular, and to a wider range of LLM-based program repair techniques, in general. Comparing FlakyDoctor with general State-of-the-art APR techniques [55, 57], even those that leverage LLMs such as GPT-4 [56], have been only proven to be effective on known datasets such as Defects4J [5] and QuixBugs [7]. Most of these techniques [22, 23, 56] also assume perfect bug localization before the repair. In contrast, FlakyDoctor repairs many flaky tests from real-world projects, where humans or automated techniques previously could not repair a reasonable number. As we will show in subsequent research questions, this power comes from the synergy of LLMs and symbolic approaches, not just LLMs.

[6]spring-projects
[7]GoogleCloudPlatform
[8]apolloconfig
[9]The links to the opened PRs are available on the artifact website [2].

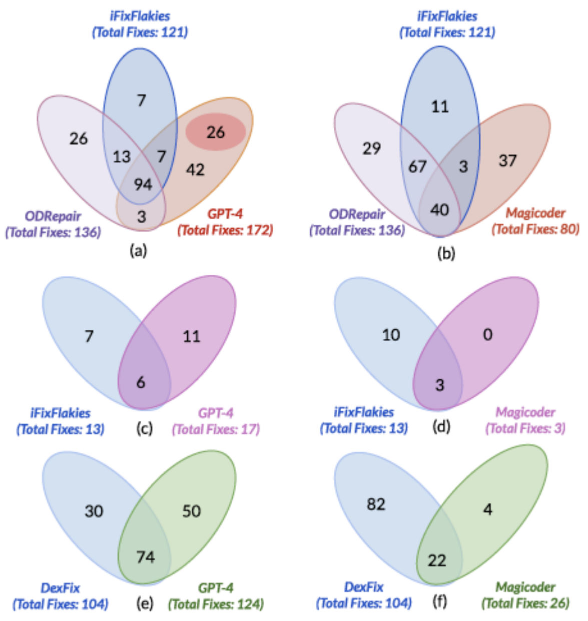


Figure 5: Comparison between the correct patches generated by different approaches. Sub-figures a-b compare OD-Victim, c-d compare OD-Brittle, and e-f compare ID patches

4.3 RQ2: Comparison with Alternative Approaches

We further wanted to explore the properties of flaky tests repaired by different approaches. To that end, we illustrate the overlap of successful patches generated by FlakyDoctor and those from alternative approaches in Figure 5. For OD-Victim tests (Figures 5a-5b), FlakyDoctor-GPT-4 exclusively repairs 68 OD-Victim tests, including 26 that were previously unfixed. FlakyDoctor-Magicoder can fix 37 tests that neither ODRepair nor iFixFlakies could fix. Alternative OD repair approaches fail at repairing such cases due to the need for existing cleaners (iFixFlakies) or difficulties in identifying complex shared statuses beyond static variables (ODRepair). For OD-Brittle tests (Figures 5c-5d), FlakyDoctor-GPT-4 exclusively repairs 11 of them (nine previously unfixed). FlakyDoctor-Magicoder can repair only three OD-Brittles, all fixed by iFixFlakies.

Regarding ID tests (Figures 5e-5f), FlakyDoctor-GPT-4 successfully repairs 50 ID flaky tests that DexFix cannot, and FlakyDoctor-Magicoder manages to repair four tests that are beyond the

Table 2: Effectiveness of FlakyDoctor, ODRepair and iFixFlakies in repairing OD flakiness. P: Github Projects; M: Modules; PF: Previously Fixed ID tests; PU: Previously Unfixed ID tests; FP: False Positive. Green rows indicate the superiority of FlakyDoctor, yellow rows indicate tie, red rows indicate the superiority of alternative approaches, and the white row indicates cases where none of the techniques repaired flaky tests.

GitHub ID	#P [#M]	OD-Victim											OD-Brittle								
		#Tests		GPT-4			Magicoder			ODRepair	iFixFlakies		#Tests		GPT-4			Magicoder			iFixFlakies
		PF	PU	PF	PU	FP	PF	PU	FP				PF	PU	PF	PU	FP	PF	PU	FP	
winder	1 [2]	5	1	5	1	0	0	0	0	5	5		-	-	-	-	-	-	-	-	-
tbsalling	1 [1]	2	0	2	0	0	0	0	0	0	0		-	-	-	-	-	-	-	-	-
tools4j	1 [1]	1	0	1	0	0	0	0	0	0	0		-	-	-	-	-	-	-	-	-
yangfuhai	1 [1]	0	6	0	6	0	0	0	0	0	0		-	-	-	-	-	-	-	-	-
jnr	1 [1]	0	4	0	3	0	0	0	0	0	0		-	-	-	-	-	-	-	-	-
Activiti	1 [1]	0	11	0	9	0	0	0	0	0	0		0	8	0	7	0	0	0	0	0
wildfly	1 [1]	37	0	37	0	0	37	0	0	0	0		1	0	0	0	1	0	0	0	1
vmware	1 [1]	1	0	0	0	0	0	0	0	0	0		0	1	0	1	0	0	0	0	0
wikidata	1 [1]	2	0	2	0	0	0	0	0	0	2		3	0	3	0	0	2	0	0	3
alibaba	2 [2]	4	0	3	0	1	1	0	0	1	1		1	1	0	1	1	1	0	1	2
apache	8 [21]	45	41	6	7	2	0	0	0	33	10		5	1	2	0	0	0	0	0	1
fhoeben	1 [1]	-	-	-	-	-	-	-	-	-	-		1	0	1	0	0	0	0	1	1
OpenHFT	1 [1]	-	-	-	-	-	-	-	-	-	-		2	0	2	0	0	0	0	0	2
undertow-io	1 [1]	1	0	1	0	0	0	0	0	1	1		-	-	-	-	-	-	-	-	-
kevinsawicki	1 [1]	28	0	28	0	0	0	0	0	28	28		-	-	-	-	-	-	-	-	-
Thomas-S-B	1 [1]	46	0	46	0	0	40	0	0	46	46		-	-	-	-	-	-	-	-	-
ktuukkan	1 [1]	12	0	12	0	0	0	0	0	12	12		-	-	-	-	-	-	-	-	-
google	1 [1]	1	0	1	0	0	0	0	0	1	0		-	-	-	-	-	-	-	-	-
spring-projects	2 [3]	2	11	2	0	0	2	0	1	0	2		0	1	0	0	0	0	0	0	0
ConsenSys	1 [2]	3	3	0	0	0	0	0	0	0	3		-	-	-	-	-	-	-	-	-
ctco	1 [1]	1	0	0	0	0	0	0	0	0	1		-	-	-	-	-	-	-	-	-
dropwizard	1 [1]	1	0	0	0	0	0	0	0	0	1		-	-	-	-	-	-	-	-	-
networknt	1 [6]	9	5	0	0	0	0	0	0	9	9		-	-	-	-	-	-	-	-	-
hexagonframework	1 [1]	-	-	-	-	-	-	-	-	-	-		1	0	0	0	0	0	0	0	1
pinterest	1 [1]	-	-	-	-	-	-	-	-	-	-		2	0	0	0	0	0	0	0	2
Others [10]	9 [9]	0	16	0	0	0	0	0	0	0	0		3	2	0	0	0	0	0	0	0
Total	43 [64]	201	98	146	26	3	80	0	1	136	121		19	14	8	9	2	3	0	2	13

capability of DexFix, which is limited to specific heuristics and fails to generalize beyond them. Figure 1 and Figure 2 show examples of cases where FlakyDoctor was able to repair flaky tests, but alternative OD and ID repair approaches could not.

4.3.1 OD tests that FlakyDoctor cannot repair. Through manual investigation of cases where FlakyDoctor could not fix but alternative approaches did, we identified two main recurring patterns: (1) Even though our program analysis provided the polluted variable(s) as context, the patches focused on resetting other variables. In most cases, these variables were directly used in the assertion of the victim method but were not the polluted states. (2) Even if LLM identified polluted states correctly, FlakyDoctor could not generate a correct patch due to hallucination. Examples of such hallucinations include adding variables that do not exist or applying APIs incompatible with the polluted field type.

4.3.2 ID tests that FlakyDoctor cannot repair. There are 30 tests fixed by DexFix but not FlakyDoctor. Breaking down these tests: (1) FlakyDoctor successfully located the unordered collections but could not generate a correct patch due to overfitting into the provided context. For example, if the assertion failure in the context is related to specific elements in the `HashMap`, while LLM creates a `LinkedHashMap` (which is an ordered collection), it only populates it with those specific elements and discards others. This may result in resolving the previous assertion failure but failing new ones. (2) FlakyDoctor successfully located the unordered collections and sorted the elements in a deterministic order. However, it consistently faced compilation errors due to hallucinating unsupported operators or invoking non-existent APIs.

These results confirm that the FlakyDoctor can complement existing tools for repairing flaky tests, serving as a complementary technique along with others. For tests where symbolic techniques DexFix, iFixFlakies, and ODRepair fail to generate a patch based on existing heuristics, developers may use FlakyDoctor.

4.4 RQ3: Contribution of Different Components

In this research question, we evaluate the effectiveness of three notable contributions of FlakyDoctor: effective flakiness localization, prompt crafting, and iterative repair.

4.4.1 Flakiness Localization. Without analyzing the test report by the *Inspector*, FlakyDoctor should take the entire test execution report of compilation error stack trace. To show the impact of precise flakiness localization, we sorted all flaky tests based on the length

[10]The GitHub ID of nine projects are: vaadin, danfickle, jenkinsci, c2mon, CloudSlang, jitsi, flaxsearch, javadelight, querydsl.

Table 3: Impact of precise flakiness localization on the effectiveness of FlakyDoctor. Avg. Lines: Average length of entire test reports; Patches and O-Patches indicate correct patches with longer prompts and original FlakyDoctor.

Flakiness	#Tests	Avg. Lines	Model	#Patches	#O-Patches
ID	216	606	GPT-4	2	111
			Magicoder	0	28
OD-Victim	120	164	GPT-4	52	106
			Magicoder	42	77
OD-Brittle	13	489	GPT-4	0	12
			Magicoder	0	3

of the original test failure report, selected the top 40% (the budget caps the percentage) (349 tests), and replaced the *Failure Location* part of the prompt with the entire test report. Table 3 compares the effectiveness of the FlakyDoctor with (O-Patches column) or without (Patches column) precise flakiness localization by the *Inspector*. These results demonstrate the necessity of minimizing contextual information for LLMs to achieve a higher performance: Without trimming, 435 prompts (aggregated for both models) exceed the context window of the models. FlakyDoctor LLMs that originally could repair all selected flaky tests only repair 96 of them.

4.4.2 Prompt Crafting. To show the impact of the proposed prompt crafting approach of FlakyDoctor, we asked FlakyDoctor-GPT-4 and FlakyDoctor-Magicoder to repair all subject ID and OD flaky tests through *vanilla prompting*: to perform a task without providing additional context. Table 4 shows the result of this experiment. In this experimental setting, FlakyDoctor-GPT-4 and FlakyDoctor-Magicoder only produced 13 and two correct patches, compared to 500 and 170 original patches. Vanilla prompting results in zero patches for OD-Brittle flaky tests. In most of the failed cases, LLM either explicitly mentioned that it does not understand the problem or only explained the test code without producing any patch. **The huge performance drop (98%) in vanilla prompting indicates the impact of providing the proper context into the prompt.**

4.4.3 Stitching and Iterative Feedback. To investigate the impact of iterative feedback and *Stitching*, we tracked back the lifetime of patched flaky tests during multiple repair iterations. Overall, **for 500 flaky tests fixed by FlakyDoctor-GPT-4 and 170 by FlakyDoctor-Magicoder, *Stitching* contributes to 12% and 31% of them, respectively.**

Figures 6a-b illustrates the evolution of 170 and 500 patches from FlakyDoctor-Magicoder and FlakyDoctor-GPT-4. The left grey bar shows the initial state of tests, i.e., being flaky. After applying the patch in each iteration, the status can be *Test Pass (P)* (flakiness fixed), *Test Failure (F)* (flakiness still exists), or *Compilation Error (CE)* (patching resulted in compilation issue). We are specifically interested in patches that *Stitching* contributes to changing their status and labeled them with **[iteration number]:[status1] To [status2]**. For example, "2:CE To P" shows *Stitching* changes the state of patches in iteration 2 from *compilation error* to *test pass*.

Among the 170 tests successfully repaired by FlakyDoctor-Magicoder, *Stitching* converts compilation errors to *test pass* for 32 of them (**CE To P**). For eight tests, while *Stitching* addressed the compilation errors, the patches resulted in test failures (**CE To F**). Additionally, for 12, *Stitching* resolved partial but not all compilation issues (**CE To CE**), which also helps to generate improved patches in the subsequent iteration. For 500 correct patches generated by FlakyDoctor-GPT-4, *Stitching* helped 58 during the repair process. Among these patches, 37 were improved by *Stitching* directly into successful patches. The impact of *Stitching* is much higher on Magicoder since it is a smaller and weaker model compared to GPT-4. This entails the importance of neuro-symbolic approaches for the ultimate democratization of open-source LLMs.

24% to 60% of patches, including those fixed by *Stitching*, were generated in the first iteration. The feedback loop contributed to generating the remaining patches in subsequent iterations: **feedback loop contributes to more than doubling the number of patches generated in the first iteration.**

Table 4: The results of vanilla prompting compared to FlakyDoctor. Patches: Total generated patches; C-Patches: Correct patches; FP-Patches: False Positives.

Flakiness	#Tests	Model	#Patches	#C-Patches	#FP-Patches
ID	541	GPT-4	336	11	12
		Magicoder	255	1	10
OD-Victim	299	GPT-4	251	2	2
		Magicoder	173	1	2
OD-Brittle	33	GPT-4	13	0	1
		Magicoder	11	0	0

4.5 RQ4: Performance

To address this research question, we evaluated the time and costs involved in using FlakyDoctor during repairing tests. The iterative workflow of FlakyDoctor attempts the repair between one to five times. Tests for which a successful patch is generated may finish earlier, whereas those that cannot be repaired persist longer. On average, GPT-4 requires 87.2 seconds and costs \$0.12 to repair an ID test or OD-Brittle test, and takes 232.8 seconds at a cost of \$0.27 to complete a repair attempt for unsuccessful tests. For OD tests, GPT-4 needs 107.5 seconds at a cost of \$0.18 to repair a test successfully, and 214.2 seconds costing \$0.35 for unsuccessful repair attempts. Magicoder, on the other hand, takes 109.2 seconds to successfully repair an ID test and 355.9 seconds for an unsuccessful attempt; for OD tests, it requires 110.6 seconds for a successful repair and 247.7 seconds for an unsuccessful attempt.

5 Related Work

Many techniques have been proposed for characterising [19, 26, 27, 29, 30], detecting [11, 24, 28, 36, 41, 42, 46, 47, 51, 58, 61], or repairing [15, 20, 32, 40, 44, 47, 50] test flakiness. Recently, Chen et al. proposed Croissant [18], a tool for modifying tests such that a non-flaky test suite shows flaky behavior. iFixFlakies [44] and iPFlakies [47] are two related research on repairing test flakiness exist in Java and Python tests suites. iFixFlakies takes the order-dependent test, the failing test order, and the passing test order. The current implementation of iFixFlakies leverages iDFlakies to get the required inputs. It then modifies the execution order of different sub-sequences of tests to find tests that modify the shared state—by

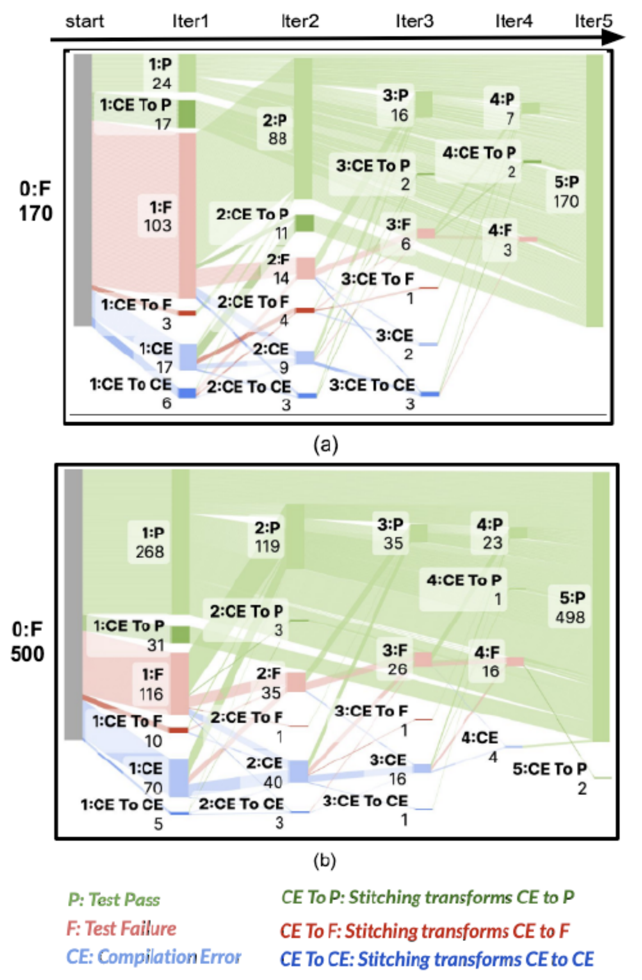


Figure 6: The evolution of patches through different repair iterations of (a) FlakyDoctor-Magicoder and (b) FlakyDoctor-GPT-4. The notation *To* indicates the applicability and impact of the *Stitching* to the current patch

setting or unsetting the shared states—with the identified victim or brittle, and uses them to generate the patch. iPFlakies follows similar steps but can only repair victim OD tests (not brittles) in Python test suites. Compared to these approaches, FlakyDoctor is more versatile in repairing both victim and brittle OD tests as well as ID flaky tests.

ODRepair [32] is proposed to overcome the limitation of iFixFlakies, which rely on the existence of *cleaner* tests to repair victim OD tests. To that end, it analyzes the static fields and serialized heap state to identify the polluted shared states between victim and polluter tests and relies on automated testing techniques to generate cleaners tests. By enforcing the execution of cleaner tests before the victim, ODRepair resolves the test flakiness. Compared to this technique, which only targets repairing victim OD tests, FlakyDoctor can repair more categories of test flakiness. Also, our proposed technique completely resolves the dependency, making the patch more realistic to resolve test flakiness.

DexFix [59] repairs ID flakiness by implementing domain-specific repair strategies that resolve implementation dependencies in both the test and the main codes. Consequently, it is limited to strategies tailored to repair studied flaky tests and may not generalize to other patterns. FlakyDoctor is not limited in that way due to relying on LLMs to perceive the nature of test flakiness and repairing based on the relevant contexts provided in the prompt.

TRaF [40] aims to address test flakiness in the JavaScript test suite of web-based applications by updating the waiting time of asynchronous calls to a value that breaks the time dependency between tests. To that end, they use code similarity and look at the relevant change history of the code, hoping to find useful hints for the efficient wait time in the existing or past code versions. Asynchronous waits are a subcategory of NOD test flakiness [18], which the current implementation of FlakyDoctor does not support. Existing research [17] indicates that merely leveraging LLMs can be challenging for repairing NOD flaky tests.

FlakyDoctor is the first test flakiness repair technique that leverages the power from a combination of LLMs and static analysis. The empirical evaluation clearly shows the benefit of this combination, i.e., repairing different categories of test flakiness and generating successful patches for flaky tests that were not previously fixed by humans or existing automated techniques.

6 Concluding Remarks

In this paper, we proposed FlakyDoctor, the first technique that combines the generalizability power of LLMs with the soundness of the program analysis, to repair different types of flakiness. Our evaluation results show that FlakyDoctor is able to generate patches for flaky tests of real-world projects that were previously unfixed. In many cases, neither prior automated techniques nor human developers were able to repair such flakiness.

We are considering several research directions on top of this work. The first obvious plan is supporting the repair of NOD flaky tests. This requires devising more complex analysis techniques to localize such flakiness issues and revising the prompt template to incorporate relevant context. Next, we plan to perform a large-scale empirical study to further pinpoint when FlakyDoctor can repair test flakiness, and when it cannot. This would provide insight into the research gap, very likely to require more advanced offline processing techniques to further help LLMs repair flaky tests.

7 Data Availability Statement

The artifacts of FlakyDoctor are publicly available at [2] and [16].

Acknowledgments

This work is supported by NSF grants CCF-2238045, CCF-1763788, CCF-1956374, IBM-Illinois Discovery Accelerator Institute, and C3.ai. We thank Darko Marinov for the valuable feedback throughout the project and anonymous reviewers for their comments throughout the review process, which resulted in this work being stronger.

References

[1] 2023. Apache Maven Surefire. https://github.com/apache/maven-surefire.

[2] 2023. The GitHub Repository of FlakyDoctor. https://github.com/Intelligent-CAT-Lab/FlakyDoctor

[3] 2023. GPT-4 Technical Report. https://cdn.openai.com/papers/gpt-4.pdf

[4] 2023. Nondex Test Flakiness Detection Tool. https://github.com/TestingResearchIllinois/NonDex.

[5] 2024. Defects4J Bug Dataset. https://github.com/jkoppel/QuixBugs/tree/master.

[6] 2024. International Dataset of Flaky tests. https://github.com/TestingResearchIllinois/idoft.

[7] 2024. QuixBugs Dataset. https://github.com/jkoppel/QuixBugs/tree/master.

[8] 2024. Repository Hadoop. https://github.com/apache/hadoop.

[9] 2024. Repository shardingsphere-elasticjob. https://github.com/apache/shardingsphere-elasticjob.

[10] 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

[11] 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.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[13] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. 2017. Test suite parallelization in open-source projects: A study on its usage and impact. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 838–848.

[14] 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).

[15] Yang Chen. 2024. Flakiness Repair in the Era of Large Language Models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. 441–443. https://doi.org/10.1145/3639478.3641227

[16] Yang Chen. 2024. *FlakyDoctor Artifact*. https://doi.org/10.5281/zenodo.12670050

[17] Yang Chen and Reyhaneh Jabbarvand. 2024. Can ChatGPT Repair Non-Order-Dependent Tests? *1st International Flaky Tests Workshop 2024* (2024).

[18] 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.

[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] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[23] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. arXiv:2302.05020 [cs.SE] https://arxiv.org/abs/2302.05020

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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*. IEEE, 312–322.

[29] 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.

[30] 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, OOPSLA (2020), 1–29.

[31] Xuan-Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. 163–163.

[32] 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.

[33] Changshu Liu, Shizhuo Dylan Zhang, and Reyhaneh Jabbarvand. 2024. CodeMind: A Framework to Challenge Large Language Models for Code Reasoning. *arXiv preprint arXiv:2402.09664* (2024).

[34] 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).

[35] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[36] 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.

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

[38] 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).

[39] 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).

[40] 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).

[41] 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*. IEEE, 149–154.

[42] 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.

[43] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999*. IEEE, 179–188.

[44] 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.

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

[46] 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.

[47] 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.

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

[49] Yufei Wang, Wanjun Zhong, Liangyou Li, Fei Mi, Xingshan Zeng, Wenyong Huang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Aligning large language models with human: A survey. *arXiv preprint arXiv:2307.12966* (2023).

[50] 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.

[51] 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.

[52] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning

in large language models. *dvances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[53] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source Code Is ll You Need. arXiv:2312.02120 [cs.CL]

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

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

[56] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[57] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the 46th IEEE/CM International Conference on Software Engineering* (, Lisbon, Portugal,) *(ICSE '24)*. ssociation for Computing Machinery, New York, NY, US, rticle 10, 13 pages. https://doi.org/10.1145/3597503.3623337

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

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

[60] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and nalysis*. 385–396.

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

Received 2024-04-12; accepted 2024-07-03