NIODebugger: A Novel Approach to Repair Non-Idempotent-Outcome Tests with LLM-Based Agent

Kaiyao Ke
University of Illinois Urbana-Champaign
Urbana, IL, USA
kaiyaok2@illinois.edu

***Abstract*—Flaky tests, characterized by inconsistent results across repeated executions, present significant challenges in software testing, especially during regression testing. Recently, there has been emerging research interest in non-idempotent-outcome (NIO) flaky tests—tests that pass on the initial run but fail on subsequent executions within the same environment. Despite progress in utilizing Large Language Models (LLMs) to address flaky tests, existing methods have not tackled NIO flaky tests. The limited context window of LLMs restricts their ability to incorporate relevant source code beyond the test method itself, often overlooking crucial information needed to address state pollution, which is the root cause of NIO flakiness.**

This paper introduces NIODebugger, the first framework to utilize an LLM-based agent to repair flaky tests. NIODebugger features a three-phase design: detection, exploration, and fixing. In the detection phase, dynamic analysis collects stack traces and custom test execution logs from multiple test runs, which helps in understanding accumulative state pollution. During the exploration phase, the LLM-based agent provides instructions for extracting relevant source code associated with test flakiness. In the fixing phase, NIODebugger repairs the tests using the information gathered from the previous phases. NIODebugger can be integrated with multiple LLMs, achieving patching success rates ranging from 11.63% to 58.72%. Its best-performing variant, NIODebugger-GPT-4, successfully generated correct patches for 101 out of 172 previously unknown NIO tests across 20 large-scale open-source projects. We submitted pull requests for all generated patches; 58 have been merged, only 1 was rejected, and the remaining 42 are pending. The Java implementation of NIODebugger is provided as a Maven plugin accessible at https://github.com/kaiyaok2/NIOInspector.

I. Introduction

Flaky tests, as documented in numerous studies [1], [2], [3], [4], [5], [6], [7], [8], exhibit inconsistent outcomes upon repeated executions of the same version of the code. This inconsistency stems from various factors, such as pollution of shared states. Often, flaky tests originate from preexisting issues within the codebase which predates recent code modifications. Consequently, their unpredictable behavior poses significant challenges in regression testing. Failures of previously passed tests after code changes may potentially lead developers to draw erroneous conclusions regarding the introduction of new bugs [9], [10]. Furthermore, flaky tests can obscure genuine defects since they typically fail under specific, often uncommon, circumstances [11]. To address this issue, numerous research endeavors have concentrated on automating the prediction [12], [13], [14], [15], [16], [17], detection [18], [19], [20], [21], and mitigation [22], [23], [24], [25], [26] of specific types of flaky tests.

Order-dependent (OD) flaky tests [27], [28] represent one extensively studied category. OD tests exhibit deterministic behavior in some test orders but fail in others. This behavior arises from undesirable dependencies between tests, often unnoticed due to implicit ordering among prevailing implementations of testing frameworks, like JUnit [29] for Java. Detecting OD tests is crucial, particularly with respect to the resilience of test suites to framework updates or migration to advanced regression testing techniques such as test prioritization, selection, and parallelization.

Exploring all possible orders of test execution can be excessively time-consuming, especially for large test suites. Therefore, one strategy for detecting OD tests entails identifying latent-polluters and latent-victims within test suites. Latent-polluters are tests that modify shared states without restoring them, while latent-victims rely on deterministic shared states. A recent study [30] has revealed that only a small portion of latent polluters and victims are convertible into actual test order dependency, as they mainly involve non-public states or have negligible impact. In response, the study proposed focusing on non-idempotent-outcome (NIO) tests [30], which exhibit changes in test outcomes across repeated runs due to their pollution of certain shared states. An NIO test is both a latent-victim and a latent-polluter and can be easily identified by running tests twice in the same environment. Detecting NIO tests can aid in preemptively addressing OD test issues, because an NIO test can become OD when a new test pollutes the shared state, and can turn a new test into OD if it reads a part of the NIO-polluted state. For example, in Figure 1, `t1()` is an NIO test. Consider when `t2()` is added - now `t1()` becomes a real polluter of `t2()`. On the other hand, when `t3()` is added, the NIO test `t1()` becomes a victim of `t3()`. The key to fixing an NIO test is to reset the polluted state before or after test execution, which, in this case, is achieved by resetting `w` to `0`.

In recent years, numerous techniques have emerged to fix program defects and even a subset of flaky tests. Multiple research studies have focused on automatically fixing buggy programs using Large Language Models (LLMs). CodeBERT

```
void t1() { assertEquals(0, w); w = 1; } // NIO Test
void t2() { assertEquals(0, w); ... }
void t3() { w = 1; ... }
```

Fig. 1: A Sample NIO Test

was the first LLM explored for automatic program repair (APR) [31], while subsequent studies have shown promising results using more advanced LLMs [32]. Among these, GPT models have demonstrated superior performance in APR tasks compared to other LLMs [33]. More recent research on LLM-based bug fixing [34], [35] has emphasized the importance of providing sufficient context, including buggy code, to enhance the performance of LLMs in bug-fixing tasks. This has led to the development of LLM-based agents [36], [37], [38], which treat the LLM as an autonomous agent capable of planning and executing actions to achieve the goal of fixing bugs. As for flaky tests, though non-agentic LLM-based techniques have achieved state-of-the-art results in addressing some types of flaky tests that can be fixed without knowledge of the main code under test [39], [40], they do not generalize well to others [41] - due to the context window limitations of LLMs, these techniques do not consider rich information from the source code beyond the test code itself. In particular, non-agentic techniques cannot fix NIO tests effectively, as it often requires additional knowledge to properly clean up state pollution.

This paper presents NIODebugger, a three-phase approach that uses an LLM-based agent to address NIO tests.

During the detection phase, NIODebugger reruns tests in an isolated environment to record test status and flag potential NIO tests, and also utilizes a custom summary listener to detect variations in stacktraces across multiple test runs, which may unveil patterns of state pollution—the cause of NIO tests.

During the exploration phase, NIODebugger retrieves the test code for each NIO test and integrates it with the collected dynamic analysis data. It queries an LLM for instructions to find relevant source code that can assist in fixing the NIO test. NIODebugger equips the agent with numerous code-extracting workflows, allowing it to interact with the codebase for context-specific information, similarly to a human developer.

Following the agent's instructions, the fixing phase of NIODebugger collects the relevant source code and queries an LLM to fix tests. The LLM first generates a patch for the specific test, and then uses the patch and the original test file to generate a compilable file that replaces the original test file.

To evaluate the effectiveness of NIODebugger, we ran the detection phase on a selection of popular open-source projects. After running the detection phase at scale, we identified 172 flaky tests across 20 popular GitHub projects. We utilize four different underlying LLMs (two open-source and two proprietary) for NIODebugger to compare their performance against existing non-LLM-based baselines that can be extended to fix NIO tests. The best variant, NIODebugger-GPT-4, significantly outperforms baseline techniques by successfully fixing 101 of these tests, of which 58 were accepted by the time of this submission. Only one pull request was rejected, while the rest are still pending. 52 patches were directly accepted without modifications, while 6 were accepted after changes requested, such as moving the cleanup routine to an `@After` method. These results demonstrate the practical applicability and effectiveness of NIODebugger in detecting and repairing flaky tests in real-world open-source projects.

In summary, this paper contributes the following:

- The first LLM-based agent for flaky test fixes, leveraging the novel approach of using LLMs to provide guidance in searching for relevant context during the fixing process.
- A Java implementation of NIODebugger that effectively detects and fixes NIO flaky tests. The tool is published on Maven Central.
- A framework that incorporates dynamic analysis during the detection phase, enabling the detection and fixing of flaky tests to occur within a single lifecycle.
- A dataset comprising previously unidentified NIO flaky tests found in popular open-source projects, along with their corresponding auto-generated patches produced by NIODebugger, if applicable.

II. Related Work

A. NIO flaky tests

The sole study on non-idempotent-outcome (NIO) flaky tests [30] conducted a comprehensive analysis across open-source projects, identifying a total of 223 NIO Java tests. Utilizing the iDFlakies tool [18] tailored for detecting order-dependent (OD) flaky tests, the study facilitated the detection of Java NIO tests by test repetition within a single execution. The study also manually fixed the majority of the identified tests. Despite its significant contribution, the study faced two notable limitations:

- Unspecialized Detector with Limited Accessibility: Its detector, derived from modifications to the existing iDFlakies tool, is restricted to supporting JUnit 4 and lacks the code conciseness and optimal efficiency needed for NIO test detection. Besides reporting possible NIO tests, it does not produce further information for debugging.
- Manual Fixes: All test fixes were performed manually, demanding in-depth understanding of the source code and consuming significant time resources. Previous study shows that a fix to one NIO test usually requires at least an hour from multiple authors [30].

These limitations call for a more specialized and accessible solution to the mitigation process for NIO tests.

B. LLM-Based Flaky Test Fixing Techniques

Though there is no previous work that fixes NIO flaky tests automatically, LLM-based techniques have achieved state-of-the-art results in addressing other categories of flaky tests. FlakyDoctor [39] presents an approach that directly queries LLMs to fix flaky tests by providing error messages from failed test runs. FlakyDoctor is robust at fixing order-dependent (OD) and implementation-dependent (ID) tests (tests making false assumptions on underdetermined APIs) [42]. It extracts these error messages from executions of the ID test detection tool NonDex [19] or the OD test detection tool iDFlakies [18]. Although the FlakyDoctor framework could potentially be extended to resolve NIO tests, its current approach is limited for 3 major reasons. Firstly, FlakyDoctor only includes the code of the flaky test method when prompting the LLM, and this does not generalize well to fixing NIO tests, which often require invoking custom cleaner methods defined in main classes. Secondly, it treats the failure message of a test as constant. While this approach aligns with most other types of flaky tests, it may not accurately capture the behavior of NIO tests. For instance, consider an NIO test that increments a static variable initialized at `0` and then asserts it to be `1`. In successive test runs, this test may yield different error messages (e.g., `"expected:<1> but was:<2>`" and "`expected:<1> but was:<3>`"). The variability in error messages across successive runs provides valuable insights for LLMs to understand errors stemming from the accumulation of state pollution, including incremented counters, retrieval of different objects from the head of a collection, and other factors. Furthermore, given the extensive research on ID and OD flaky tests, FlakyDoctor's zero-shot architecture has shown promising results. However, the landscape differs for NIO tests, with only one previous study conducted in 2022.

FlakyFix [40] introduces an innovative framework that leverages neural networks to predict fix categories before utilizing LLMs to address flaky tests. The authors use publicly reported flaky tests from the IDoFT [43] dataset to generate labeled datasets for 13 heuristically defined fix categories. They train a model to predict the fix category using the flaky test code before directly employing LLMs to fix the test. While FlakyFix could also potentially be extended to include NIO-related fix categories, its effectiveness is also limited by its reliance solely on test code for prompting and its omission of valuable information from stack traces.

C. LLM-Based Agents

LLM-based agents represent a new line of research in automated program repair (APR). This approach augments the LLM into an agent capable of autonomously planning and executing actions to fix bugs by invoking suitable tools or APIs. It leverages the LLM's ability to understand the root cause of issues and effectively retrieve context. RepairAgent [36] is the first work to address the program repair challenge using an LLM-agent-based technique, applied to the Defects4J [44] dataset. AutoCodeRover [38] introduces an LLM-agent-based approach for resolving pending GitHub issues to autonomously achieve program improvement. FixAgent [37] proposes the first automated, unified debugging framework via LLM agent synergy, where two LLM agents act as a bug localizer and program repairer. These agents are prompted to explicitly track key variables at critical points in the buggy program and discuss how such tracking guides their task completion. Additionally, they help construction of the program context concerning its specifications and dependencies.

There is no agent-based approach in the field of flaky test repair. This paper is the first to incorporate such a design to glean source code that can inspire patching NIO tests.

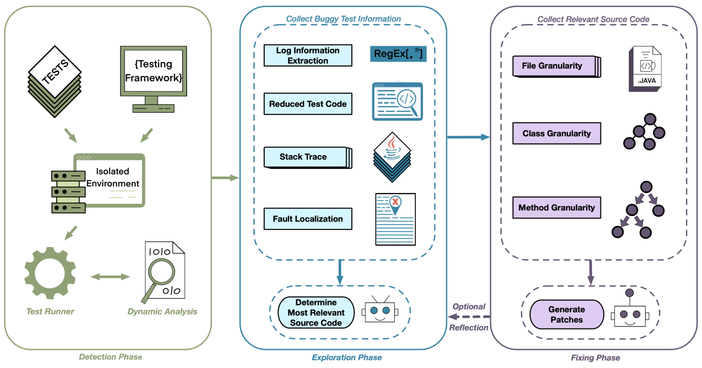


Fig. 2: NIODebugger Architecture

III. Approach

Figure 2 provides an overview of NIODebugger, a three-phase framework designed to tackle non-idempotent-outcome (NIO) flaky tests by an LLM-based agent. In the detection phase, NIODebugger reruns tests within the same environment to pinpoint potential NIO tests while simultaneously performing dynamic analysis to capture essential information useful for debugging. In the exploration phase, NIODebugger combines the identified test code with dynamic analysis data and consults an LLM for guidance on locating relevant source code. This equips the LLM with code-extracting techniques, enabling it to retrieve context necessary for rectifying NIO tests. In the fixing phase, NIODebugger gathers the pertinent source code as directed by the previous phase and utilizes an LLM to generate patches. Finally, an optional reflection phase is available to iteratively refine the patch based on previous results. This approach effectively addresses NIO test flakiness by leveraging dynamic analysis insights, the gathered information, and a one-shot learning example.

The majority of flaky tests identified in the previous study [30] originate from open-source Maven Java projects, reflecting Maven's status as the preferred build tool for large-scale software applications due to its robust dependency management and build handling capabilities. A recent study [45] highlights that over 76% of Java developers use Maven for their builds. Similarly, JUnit is widely adopted, with approximately 85% of Java developers using it as their unit testing framework, according to a recent survey [46]. In light of these trends, we developed NIODebugger as a command-line plugin available on Maven Central, specifically designed to detect and address JUnit NIO tests in Maven Java projects. However, the NIODebugger technique is not limited to Java, Maven, or JUnit. The three-phase workflow and LLM agent can be generalized to support other programming languages, build tools, and testing frameworks. By customizing the detection phase to be framework-specific and tailoring the exploration and fixing phases to be language-specific, NIODebugger can be adapted to a wide range of environments.

Algorithm 1: NIO Flaky Tests Detection and Dynamic Analysis

```
Inputs: Project $P$, Number of Reruns $numReruns$
Output: Possible NIO Flaky Tests $possibleNIOTests$, Other Flaky Tests
        $otherFlakyTests$, Dynamic Analysis Output
        $dynamicAnalysisLog$
$firstRunResult, rerunResults \leftarrow \{\}$;
$rerunStacktraces, rerunExtraInfo \leftarrow \{\}$;
$possibleNIOTests, otherFlakyTests \leftarrow \emptyset$;
$allTests \leftarrow findAllTests(P)$;
foreach $t \in allTests$ do
    $runner \leftarrow createIsolatedTestRunner(t)$;
    $env \leftarrow createIsolatedEnvironment(t)$;
    $result \leftarrow runTest(runner, env, t)$;
    $firstRunResult[t] \leftarrow result$;
    $rerunResults[t] \leftarrow []$;
    $rerunStacktraces[t] \leftarrow []$;
    $rerunExtraInfo[t] \leftarrow []$;
    for $i \leftarrow 1$ to $numReruns$ do
        $result, extraInfo \leftarrow$
          $runTestWithExtraInfoCollector(runner, env, t)$;
        $rerunResults[t].append(result)$;
        $rerunExtraInfo[t].append(extraInfo)$;
        if $result = fail$ then
            $stacktrace \leftarrow getStackTrace(env)$;
            $rerunStacktraces[t].append(stacktrace)$;
foreach $t \in allTests$ do
    if $(firstRunResult[t] = pass) \wedge (\forall result \in$
      $rerunResults[t], result = fail)$ then
        $possibleNIOTests \leftarrow possibleNIOTests \cup \{t\}$;
    else if $\exists result1, result2 \in$
      $(firstRunResult[t] \cup rerunResults[t]), result1 \neq result2$
      then
        $otherFlakyTests \leftarrow otherFlakyTests \cup \{t\}$;
foreach $t \in possibleNIOTests$ do
    $runner \leftarrow getIsolatedTestRunner(t)$;
    $env \leftarrow createCleanIsolatedEnvironment(t)$;
    $result \leftarrow runTest(runner, env, t)$;
    if $result = fail$ then
        $possibleNIOTests \leftarrow possibleNIOTests \setminus \{t\}$;
        $otherFlakyTests \leftarrow otherFlakyTests \cup \{t\}$;
        $continue$;
    for $i \leftarrow 1$ to $numReruns$ do
        $result \leftarrow runTest(runner, env, t)$;
        if $result = pass$ then
            $possibleNIOTests \leftarrow possibleNIOTests \setminus \{t\}$;
            $otherFlakyTests \leftarrow otherFlakyTests \cup \{t\}$;
            $break$;
$dynamicAnalysisLog \leftarrow$
  $log(possibleNIOTests, rerunStacktraces, rerunExtraInfo)$;
return $possibleNIOTests, otherFlakyTests, dynamicAnalysisLog$;
```

A. Detection Phase

To address the key limitations of the detector used in the original study of NIO flaky tests [30], which lacked useful debugging information, we developed a specialized workflow for detecting NIO tests. Algorithm 1 illustrates our systematic approach involving dynamic analysis. Initially, NIODebugger identifies all tests in a given project. For each test, it first creates an isolated testing environment and runs the test once, recording this initial result. It then enters a loop to rerun the test multiple times within the same environment. During each rerun, NIODebugger captures the result of the test execution. If a test fails, it retrieves and stores the corresponding stack trace. Regardless of the test's result, additional runtime information, such as logging outputs and warnings, is also collected. After all reruns are completed for each test, NIODebugger assesses whether the test qualifies as a possible NIO flaky test. This determination is based on the observation that if a test passed on the first run but consistently failed in subsequent reruns, it is flagged as a potential NIO flaky test. For each test flagged as a potential NIO flaky test, NIODebugger re-creates a clean isolated environment and reruns the test to confirm whether it consistently exhibits non-idempotent behavior. The detection phase concludes by logging the identified NIO tests along with the results of the dynamic analysis.

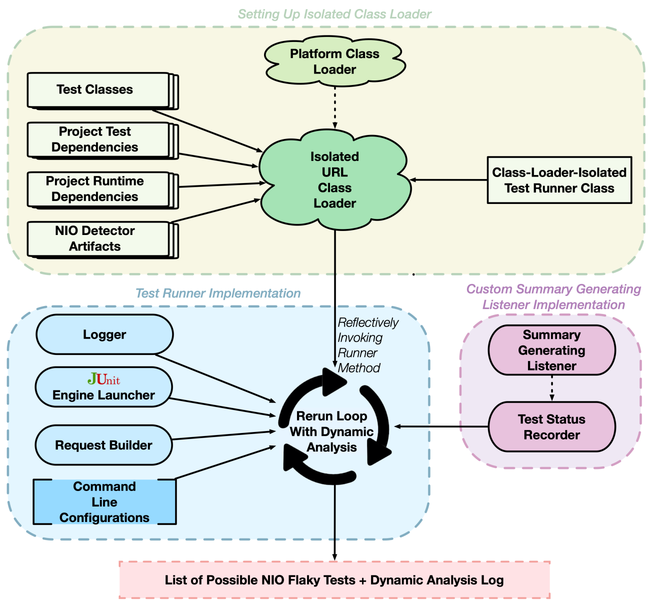

Setting Up Isolated Class Loader
Platform Class Loader
Test Classes
Project Test Dependencies
Project Runtime Dependencies
NIO Detector Artifacts
Isolated URL Class Loader
Class-Loader-Isolated Test Runner Class
Test Runner Implementation
Custom Summary Generating Listener Implementation
Reflectively Invoking Runner Method
Logger
JUnit Engine Launcher
Request Builder
Command Line Configurations
Rerun Loop With Dynamic Analysis
Summary Generating Listener
Test Status Recorder
List of Possible NIO Flaky Tests + Dynamic Analysis Log

Fig. 3: Java Implementation of Detection Phase

In addition, we describe our efforts to integrate the algorithm seamlessly with the modern Java ecosystem. In Maven projects, the Maven Surefire Plugin [47] is commonly used during the build lifecycle to execute JUnit tests. However, Surefire automatically spawns and terminates forked Java Virtual Machines (JVMs) for testing, making it challenging to rerun tests in the same JVM. To address this limitation for Maven Java projects, the Java implementation of the detection phase initializes a custom isolated class loader to load all test classes, necessary artifacts for test execution, and the JUnit test engine. Subsequently, it employs a class-loader-isolated JUnit runner capable of rerunning tests within the same JVM. While the workflow is tailored to Java, its underlying principles can be adapted to other programming languages.

Figure 3 shows the workflow of the detection phase for JUnit Java tests. Specifically, the core detection architecture includes four major components:

1) Isolated Class Loader: To ensure a highly streamlined environment that includes only the essential dependencies necessary for repeated unit test execution, the isolated class loader minimizes the number of dependencies. The class loader provides a concise environment encompassing test classes, essential runtime dependencies, specialized unit-testing artifacts, NIODebugger's proprietary classes, and the entire Java standard library. Notably, the bootstrap class loader in Java versions 9 and later exclusively loads core Java classes from the runtime environment, such as those within the `java.lang` package. However, it does not load packages like `java.sql` which require the `--add-module` flag during JVM launch. To ensure comprehensive access to Java SE platform APIs, including JDK-specific runtime classes, we configure the parent of our isolated class loader to be the platform class loader. We implement the isolated class loader as a `URLClassLoader`, facilitating the loading of artifacts via path URLs.

2) Class-Loader-Isolated Test Runner: With the class loader prepared, the runner can rerun specified test classes or methods, or the entire test suite by default. Using reflection, the runner invokes the JUnit launcher factory's creator method from within the isolated class loader. The request builder, which accepts designated test classes or methods identified by JUnit discovery selectors, is passed to the test engine launcher. The launcher activates the appropriate JUnit Engine, which processes the request and orchestrates the repetitive execution of tests. The runner employs a custom logger to capture useful test information for debugging. During this process, the test runner reruns unit tests, examines their outcomes, and flags tests that pass initially but fail in subsequent runs.

3) Custom Summary Generating Listener: NIODebugger uses an extension of JUnit's summary generating listener to provide a comprehensive overview of test execution. While the JUnit listener only captures the names of failed tests, our custom listener maintains a map of test statuses, updated with each test's completion. This map ensures that the runner has information about tests that pass initially but fail later, and also supports the extraction of additional dynamic information such as stack traces and warnings.

4) Executor: An executor operates on top of the three components described above. Implemented as a Maven Mojo, it is responsible for executing tasks within the Maven build process. Our rerun execution Mojo identifies all tests slated for execution, processes command-line inputs, retrieves URLs for the isolated class loader, and loads the class-loader-isolated test runner class into the customized class loader. Finally, it invokes the JUnit runner method via reflection.

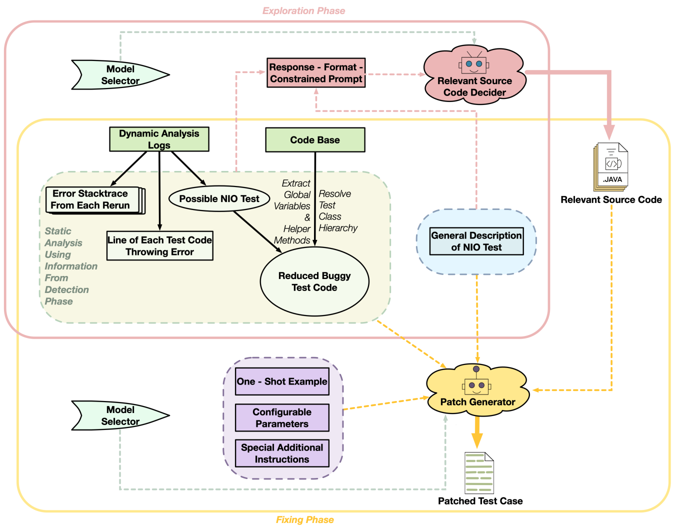
B. Exploration Phase

The exploration phase of NIODebugger is a two-step process designed to utilize our LLM-based agent for NIO flaky tests. Initially, it performs static analysis on the results from the detection phase to gather the most relevant information for each test. Subsequently, the relevant information, along with the test code, is used to query an LLM for suggestions on relevant code that may aid in debugging. This phase can be easily generalizable to any programming languages and testing frameworks. Part of Figure 4 illustrates the overall workflow of the exploration phase, while Algorithm 2 details the specific process of interacting with an LLM-based agent to generate instructions for relevant source code extraction.

Specifically, for each test identified as a potential NIO flaky test during the detection phase, NIODebugger extracts error

Algorithm 2: Exploration Phase of NIODebugger

```
Inputs: Project $P$, Possible NIO Flaky Tests $possibleNIOTests$, Dynamic
        Analysis Log $dynamicAnalysisLog$, Number of Reruns
        $numReruns$
Output: Instructions for Relevant Source Code Extraction $instructions$
$instructions, errorLineNums, errorLines \leftarrow \{\}$;
$stackTraces, extraInfo, reducedTestCode \leftarrow \{\}$;
foreach $t \in possibleNIOTests$ do
    $testFileCopy \leftarrow collectTestFile(t, getCodeBase(P))$;
    $errorLineNums[t] \leftarrow$ [];
    $errorLines[t] \leftarrow$ [];
    for $i \leftarrow 1$ to $numReruns$ do
        $errorLineNum \leftarrow$
          $getStackTrace(dynamicAnalysisLog, t)$;
        $errorLineNum \leftarrow errorLineNum.getErrorLineNum(i)$;
        $errorLineNums[t].append(errorLineNum)$;
    foreach $lineNum \in errorLineNums[t]$ do
        $errorLine \leftarrow extractLine(testFileCopy, lineNum)$;
        $errorLines[t].append(errorLine)$;
    foreach $method \in getAllTestMethodsFromFile(testFileCopy)$
      do
        if $method.getName() \neq t.getName()$ then
            $testFileCopy \leftarrow$
              $removeMethod(testFileCopy, method)$;
    $reducedTestCode[t] \leftarrow testFileCopy$;
    $extraInfo[t] \leftarrow parseExtraInfo(dynamicAnalysisLog, t)$;
    $stackTraces[t] \leftarrow$ [];
    for $i \leftarrow 1$ to $numReruns$ do
        $stackTrace \leftarrow getStackTrace(dynamicAnalysisLog, t)$;
        $stackTrace \leftarrow$
          $stackTrace.extractStackTraceAtRerunNum(i)$;
        $stackTraces[t].append(stackTrace)$;
$generalDescription \leftarrow getGeneralDescriptionOfNIOTests()$;
foreach $t \in possibleNIOTests$ do
    $prompt \leftarrow buildResponseFormatConstrainedPrompt($
      $t, errorLines[t], stackTraces[t], reducedTestCode[t],$
        $extraInfo, generalDescription)$;
    $instruction \leftarrow queryLLM(prompt)$;
    $instructions[t] \leftarrow instruction$;
return $instructions$;
```



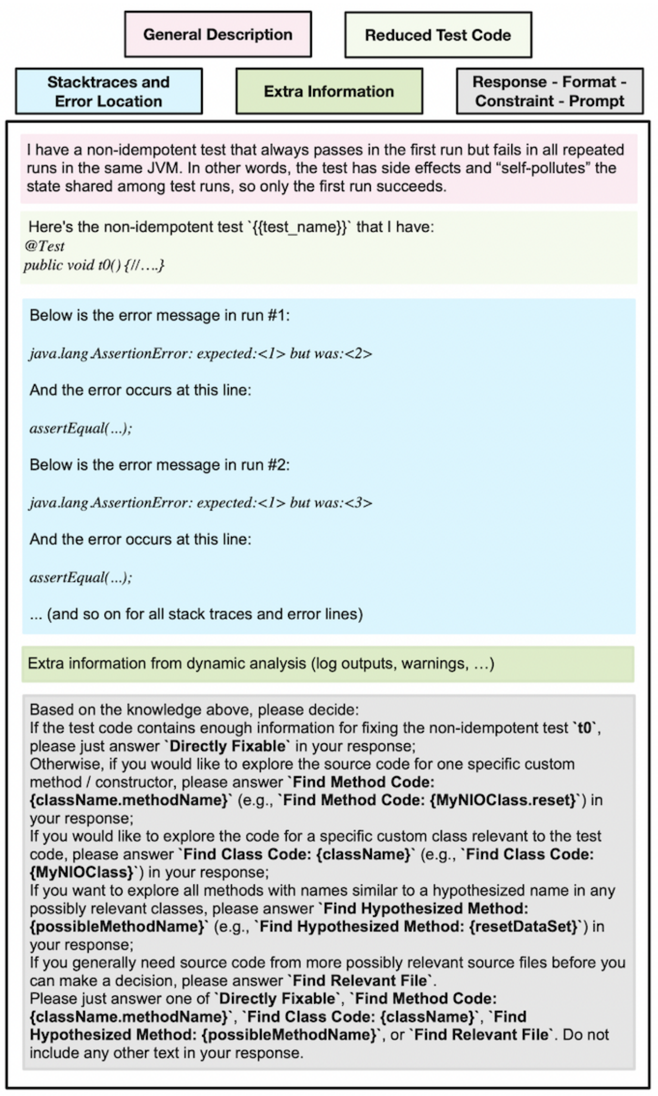
Exploration Phase
Model Selector
Response - Format - Constrained Prompt
Relevant Source Code Decider
Dynamic Analysis Logs
Code Base
Error Stacktrace From Each Rerun
Possible NIO Test
Extract Global Variables & Helper Methods
Resolve Test Class Hierarchy
Line of Each Test Code Throwing Error
Reduced Buggy Test Code
Static Analysis Using Information From Detection Phase
General Description of NIO Test
.JAVA
Relevant Source Code
One - Shot Example
Configurable Parameters
Special Additional Instructions
Model Selector
Patch Generator
Patched Test Case
Fixing Phase

Fig. 4: Exploration and Fixing Phases of NIODebugger

line numbers from the dynamic analysis log and parses the corresponding test file to locate these lines. It then creates a reduced version of the test file, adhering to the principle that unit tests should be independent. To minimize noise and manage context length, NIODebugger removes other test methods while retaining helper methods, fields, and classes within the file. Additionally, it includes inherited fields and methods from superclasses, resulting in a streamlined yet informative codebase for each NIO test. Meanwhile, NIODebugger parses extra information from the dynamic analysis log, including stack traces from various reruns. All collected data, along with a general description of NIO tests, is fed into an LLM agent, which then decides to extract relevant code from the repository that assists in fixing the NIO test.

To query the LLM agent, we construct a **response-format-constrained prompt**, which includes explicit instructions on the desired structure and content of the LLM's response. The prompt includes guidelines or templates which the LLM must adhere to in generating its response, thereby ensuring that the output is consistent with the specified format to invoke a **parameterized workflow**. The response-format-constrained prompt not only contains test-specific information, but also includes a section that specifies all custom source-code-searching parameterized workflows from which the LLMs can choose, detailing the inputs to be passed into these workflows and the format of the LLM response to ensure it is automatically parsable for invoking such workflows. Figure 5 shows the format of the response-format-constrained prompt. Below is a detailed overview of all parameterized workflows:

1) **Find Code of a Specific Method**: This workflow is used when the LLM identifies that a method might be, or is associated with, a latent polluter or cleaner, and requires access to the method's implementation for verification or additional relevant information. NIODebugger requires the LLM to respond `Find Method Code: {className.methodName}` in this scenario.
2) **Find Code of a Specific Class**: This workflow is employed when the LLM determines that understanding the implementation of a type as a whole is essential, especially when it needs a comprehensive view of a class's potential fields that may be shared and polluted. NIODebugger requires the LLM to respond `Find Class Code: {className}` in this scenario.
3) **Function Name Inference and Code Matching**: Used when the LLM anticipates potential functions to clean up states but does not know in advance which class defines such methods. The LLM can then infer a function name based on the desired functionality of state cleaning and locate code for multiple functions with names similar to the guessed name. This feature can also be utilized when the LLM needs to explore a function that appears in the code but does not know which class defines the function, such as in long call chains. NIODebugger requires the LLM to respond `Find Hypothesized Method: {possibleMethodName}` in this scenario.
4) **Explore File With Similar Names**: This workflow is utilized when the LLM is unable to make a specific decision at the class or method level and opts to explore source files whose names are similar to the file containing the NIO test, potentially uncovering more insights. NIODebugger requires the LLM to respond `Find Relevant File` in this scenario.
5) **No Source Code Needed Beyond Reduced Test Code**: If the reduced test code contains enough information for a fix, NIODebugger can skip further exploration of the code base. NIODebugger requires the LLM to respond `Directly Fixable` in this scenario.



General Description
Reduced Test Code
Stacktraces and Error Location
Extra Information
Response - Format - Constraint - Prompt

I have a non-idempotent test that always passes in the first run but fails in all repeated runs in the same JVM. In other words, the test has side effects and "self-pollutes" the state shared among test runs, so only the first run succeeds.

Here's the non-idempotent test `{{test_name}}` that I have:
@Test
public void t0() {//….}

Below is the error message in run #1:

java.lang.AssertionError: expected:<1> but was:<2>

And the error occurs at this line:

assertEqual(...);

Below is the error message in run #2:

java.lang.AssertionError: expected:<1> but was:<3>

And the error occurs at this line:

assertEqual(...);

... (and so on for all stack traces and error lines)

Extra information from dynamic analysis (log outputs, warnings, …)

Based on the knowledge above, please decide:
If the test code contains enough information for fixing the non-idempotent test **`t0`**, please just answer **`Directly Fixable`** in your response;
Otherwise, if you would like to explore the source code for one specific custom method / constructor, please answer **`Find Method Code: {className.methodName}`** (e.g., **`Find Method Code: {MyNIOClass.reset}`**) in your response;
If you would like to explore the code for a specific custom class relevant to the test code, please answer **`Find Class Code: {className}`** (e.g., **`Find Class Code: {MyNIOClass}`**) in your response;
If you want to explore all methods with names similar to a hypothesized name in any possibly relevant classes, please answer **`Find Hypothesized Method: {possibleMethodName}`** (e.g., **`Find Hypothesized Method: {resetDataSet}`**) in your response;
If you generally need source code from more possibly relevant source files before you can make a decision, please answer **`Find Relevant File`**.
Please just answer one of **`Directly Fixable`**, **`Find Method Code: {className.methodName}`**, **`Find Class Code: {className}`**, **`Find Hypothesized Method: {possibleMethodName}`**, or **`Find Relevant File`**. Do not include any other text in your response.

Fig. 5: Response-Format-Constrained Prompt for Relevant Source Code Exploration

C. Fixing Phase

The fixing phase of NIODebugger is follows instructions from exploration phase to extract relevant source code information and query an LLM for generating a final patch for each test. Part of Figure 4 illustrates the architecture of the fixing phase, while Algorithm 3 outlines the workflow of this phase. It takes as input the maximum number of characters allowed in the collected source code (to fit within the context window), project source code, a set of all possible NIO tests, and a dictionary of instructions from the exploration phase.

Algorithm 3: Fixer Phase of NIODebugger

```
Inputs: Maximum number of characters allowed in collected source code $n$,
        Project $P$, Set of all Possible NIO Tests $possibleNIOTests$,
        Dictionary of instructions $instructions$
Output: Set of patches $patches$
1  $fileCodeMap, classCodeMap, methodCodeMap, patches \leftarrow \{\}$;
2  $generalDescription \leftarrow getGeneralDescriptionOfNIOTests()$;
3  $errorLines, stackTraces, extraInfo, reducedTestCode \leftarrow$
     $getInfoFromPreviousPhases()$;
4  $oneShotExample \leftarrow getOneShotExample()$;
5  foreach $file \in findAllSourceFiles(P)$ do
6      $fileCodeMap[file] \leftarrow extractCode(file)$;
7      $AST \leftarrow getAST(file)$;
8      foreach $node \in performDFS(AST)$ do
9          if $isInstance(node, method)$ then
10             $methodName \leftarrow getFullPathMethodName(node)$;
11             $methodCodeMap[methodName] \leftarrow$
                 extractCode($file$, $methodName$);
12         else if $isInstance(node, class)$ then
13             $className \leftarrow getFullPathClassName(node)$;
14             $classCodeMap[className] \leftarrow$
                 extractCode($file$, $className$);
15 $relevantSourceCode \leftarrow$ “”;
16 foreach $t \in possibleNIOTests$ do
17     $workflow, nameToSearch \leftarrow$
         $parseInstruction(instructions[t])$;
18     if $workflow$ = “Find Code of a Specific Method” then
19         $relevantSourceCode \leftarrow$
             $methodCodeMap[nameToSearch]$;
20     else if $workflow$ = “Find Code of a Specific Class” then
21         $relevantSourceCode \leftarrow classCodeMap[nameToSearch]$;
22     else if $workflow$ = “Function Name Inference and Code Matching”
         then
23         $distances \leftarrow \{\}$;
24         foreach $key \in methodCodeMap$ do
25             $distance \leftarrow$
                 $ComputeLevenshteinDistance(nameToSearch, key)$;
26             $distances.append((distance, key))$;
27         Sort $distances$ by distance;
28         $relevantSourceCode \leftarrow \{\}$;
29         foreach _, $key \in$
             $copyFromHeadUntilGettingNCharacters(distances, n)$
             do
30             $relevantSourceCode$.append($methodCodeMap[key]$);
31     else if $workflow$ = “Explore File With Similar Names” then
32         $distances \leftarrow \{\}$;
33         foreach $key \in fileCodeMap$ do
34             $distance \leftarrow$
                 $ComputeLevenshteinDistance(nameToSearch, key)$;
35             $distances.append((distance, key))$;
36         Sort $distances$ by distance;
37         $relevantSourceCode \leftarrow \{\}$;
38         foreach _, $key \in$
             $copyFromHeadUntilGettingNCharacters(distances, n)$
             do
39             $relevantSourceCode$.append($fileCodeMap[key]$);
40     $prompt \leftarrow$
         $buildFixerPrompt(t, errorLines[t], stackTraces[t], extraInfo[t]$
41         $oneShotExample, reducedTestCode[t], generalDescription,$
42             $relevantSourceCode)$;
43     $patches[t] \leftarrow queryLLM(prompt, n)$;
44 return $patches$;
```

In this phase, NIODebugger iterates over all project source files, extracting their code and constructing an Abstract Syntax Tree (AST) [48] for each file. By performing a Depth-First Search (DFS) on the AST, NIODebugger identifies and stores the code for each method and class. This step is specific to the language used; for Java, we utilized JavaParser [49], but similar functionality is available in other languages, such as the `ast` module [50] for Python and Clang [51] for C++.

For each NIO test, NIODebugger parses the instruction from the exploration phase to determine the required workflow. NIODebugger retrieves the relevant source code from the constructed maps when an exact match is found, or identifies the most similar entries using the Levenshtein distance [52], a common metric to map LLM-generated outputs to executable tools in agent-based program repair [36]. The relevant source code is then written unless the agent decides that no additional source code is needed. Following the source code extraction, the fixing phase synthesizes a comprehensive prompt for each possible NIO test, including a general description of NIO tests, the test method name, the reduced source code with the test method, stack traces from multiple test reruns, instructions for fixing NIO flaky tests, a one-shot example, and additional information from previous phases. Additionally, the fixer supports custom requirements for fix generation, such as specifying “fix the method itself without adding `setUp()` or `tearDown()` methods.” Figure 6 illustrates the final prompt to query a patch. As the input test file is streamlined to exclude other test methods, the patch contains only the relevant test method and its associated helper methods or classes.

During post-processing, we provide the LLM with both the original test file and the patch for one method, and prompt it to generate a compilable version to replace the original file. The separation between the fixer phase and code replacement is based on the intuition that the fixer should not be exposed to extraneous code (e.g., other test methods) when formulating a patch for one test. Additionally, this approach allows optional manual verification and code patching, potentially reducing the cost of invoking the LLM.

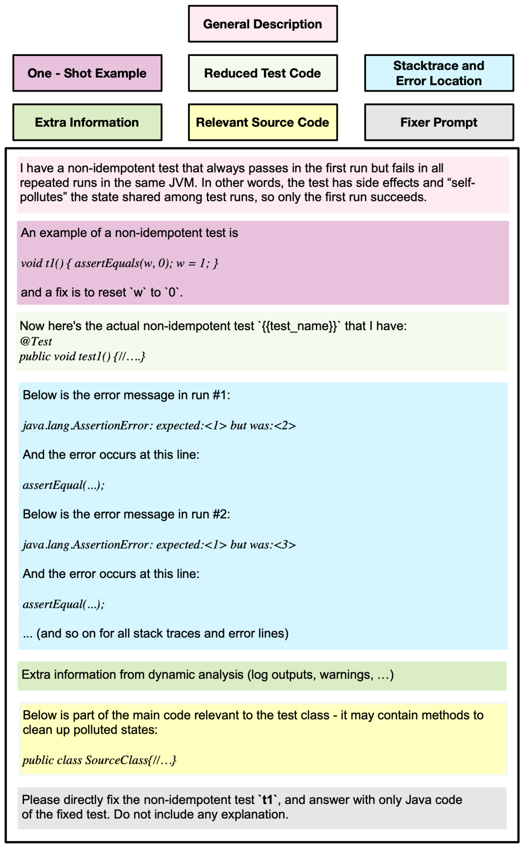
D. Optional Reflection Phase

NIODebugger supports iterative refinement as an optional step, which was employed in our evaluation. If the fixer fails to produce compilable code or to resolve test non-idempotency (i.e., the detection phase still reports non-idempotency after patching), the agent re-performs the exploration and fixing steps. During this phase, the prompt is enhanced with the previous parameterized workflow decisions, the generated patch, and new execution results. While we do not make this step mandatory due to the high computational and monetary costs associated with LLMs, we have configured our Maven plugin to automatically incorporate previous run information into the exploration and fixing phases, whenever available. Additionally, we offer a fully automatic setup that includes up to three reflection runs, which the user can choose to enable.

IV. Evaluation

To evaluate the effectiveness of NIODebugger, we investigate the following research questions:

RQ1: *Effectiveness, Generalizability, and Baseline Comparisons in Fixing NIO Tests*: How effective is NIODebugger in generating patches for NIO tests that preserve the original test logic while eliminating non-idempotency? How generalizable



I have a non-idempotent test that always passes in the first run but fails in all repeated runs in the same JVM. In other words, the test has side effects and "self-pollutes" the state shared among test runs, so only the first run succeeds.

An example of a non-idempotent test is

void t1() { assertEquals(w, 0); w = 1; }

and a fix is to reset `w` to `0`.

Now here's the actual non-idempotent test `{{test_name}}` that I have:
@Test
public void test1() {//….}

Below is the error message in run #1:

java.lang.AssertionError: expected:<1> but was:<2>

And the error occurs at this line:

assertEqual(...);

Below is the error message in run #2:

java.lang.AssertionError: expected:<1> but was:<3>

And the error occurs at this line:

assertEqual(...);

... (and so on for all stack traces and error lines)

Extra information from dynamic analysis (log outputs, warnings, …)

Below is part of the main code relevant to the test class - it may contain methods to clean up polluted states:

public class SourceClass{//…}

Please directly fix the non-idempotent test **`t1`**, and answer with only Java code of the fixed test. Do not include any explanation.

Fig. 6: Prompt for LLM to Generate a Patch

is its effectiveness across different LLMs? Is NIODebugger more effective than existing techniques that could potentially be adapted to address NIO tests?

RQ2: *Contributions to Real-World Software Development*: What is the attitude of developers working on large-scale, popular projects toward patches generated by NIODebugger?

RQ3: *Contributions of Key Components in NIODebugger*: What is the extent of the contribution of the key components in the LLM-based agent?

A. Evaluation Setup

We used the GitHub API query to obtain a list of popular Java repositories, sorted by the number of stars, with a push within a year. From the query results, we selected the top 300 repositories and filtered them to retain only those containing a `pom.xml` file in the root directory. This process resulted in a final list of 242 repositories, and we successfully executed NIODebugger on 174 of them. The remaining projects failed due to Maven build issues or test hang-ups (unrelated to NIODebugger), or lack of support for Java 9+ (a prerequisite for running NIODebugger).

Due to the time-consuming nature of initializing a separate JVM for each test class or method in large projects, we executed the detection process at the module granularity, using `numRerun = 3`—which spawns one JVM to execute the module's test suite four times consecutively. Note that this method may overlook some NIO tests whose polluted states are resolved by preceding methods. Our script identified 192 potential NIO flaky tests across 21 projects. After that, we reran the detection phase of NIODebugger at the reported test granularity with `numRerun = 10` to verify non-idempotent behavior. We failed to observe non-idempotency in 17 of the 192 tests, and found they were actually order-dependent. For example, if the original order of a test suite is `t1, t2`, then if `t2` modifies a state used by `t1`, `t1` may pass in the first run but fail afterwards. Another 3 of the reported tests were found to be nondeterministic; they passed in the initial run but failed in subsequent reruns by chance and did not exhibit the same behavior when rerun in isolation. Finally, we confirmed 172 possible NIO tests across 20 projects.

Our goal is to evaluate NIODebugger on all the 172 tests. In line with existing LLM-agent-based program repair techniques [36], [37], [38], we selected GPT-3.5 Turbo and GPT-4 as the two proprietary, API-based state-of-the-art LLMs for the exploration, fixing, and optional reflection phases. To evaluate NIODebugger's generalization capability across multiple LLMs, including open-source models, we also included two top-performing instruction-tuned open-source models, namely DeepSeek-Coder-33B-Instruct [53] and Qwen2.5-Coder-32B-Instruct [54], identified from Aider's Code Editing Leaderboard [55], which ranked models for their ability to generate code edits seamlessly integrated into existing codebase. Note that we selected the instruction-tuned coder variants to address both instruction-following needs during the exploration phase and patch generation in the fixing phase.

Using each LLM, we ran the exploration and fixing phases to generate solutions for each NIO test, also allowing up to 3 reflection runs. We conducted parameter tuning on the temperature setting for each model, utilizing the 149 previously fixed NIO tests recorded by IDoFT [43]. The tuning process began with greedy decoding (temperature = 0) and increased temperature by 0.1 in each iteration, up to a maximum of 2.

Importantly, the 149 tests used for parameter tuning were *entirely disjoint* from the 172 tests used for evaluation, which comprised newly detected, previously unknown tests, ensuring no data contamination. The optimal temperature values found were 0.7 for both GPT models, 0.5 for DeepSeek-Coder-33B-Instruct, and 0.6 for Qwen2.5-Coder-32B-Instruct. For all other parameters, we adhere to the default settings recommended in their documentation as best practice. Additionally, we allowed up to three iterations in the reflection phase, enabling the agent to incorporate insights from the execution results of previous patches. The experiment was conducted using Java 17 on Ubuntu 22.04.3.

B. RQ1: Effectiveness, Generalizability, and Baseline Comparisons in Fixing NIO Tests

We ran the aforementioned experiment on the 172 detected NIO tests using each of the four LLMs. A patch is considered correct if it satisfies the following conditions: (1) it passes the detection phase with 50 reruns, (2) it does not cause any other test to fail, and (3) it does not alter the essential test logic upon manual examination, confirmed by an experienced Java developer after reading the patches line by line. All experiment scripts and generated patches are publicly available in the artifact repository for verification.

To compare the results against the baselines, we duplicated each detected NIO test to introduce test order dependencies and ran `iFixFlakies` and `ODRepair`—two non-LLM-based approaches designed to fix order-dependent tests without using LLMs. Table I presents an overview of the results. Among the four LLMs integrated with NIODebugger, NIODebugger-GPT-4 achieved the best performance, producing correct patches for 101 tests (58.72%), far surpassing the other models. NIODebugger-GPT-3.5-Turbo ranked second, also outperforming the open-source LLMs by a wide margin. When integrated with open-source LLMs, NIODebugger-Qwen2.5-Coder-32B-Instruct and NIODebugger-DeepSeek-Coder-33B-Instruct performed less effectively.

NIODebugger-GPT-3.5-Turbo produced unique fixes for five tests that were not addressed by other LLMs, while all patches generated by the two open-source LLMs were subsumed by GPT-4 patches. NIODebugger-GPT-4 significantly outperformed non-LLM baseline approaches, and NIODebugger-GPT-3.5-Turbo also demonstrated superior performance. iFixFlakies failed to fix any tests, as it relies on "state cleaner tests" in the same test suite, which were absent for the 172 NIO tests in our study. ODRepair fixed some tests but was less effective than GPT-based NIODebugger models, since it relies on correctly identifying the polluted state and using Randoop to generate potential cleaner tests. Moreover, Randoop-generated tests tend to be verbose, poorly organized, and filled with low-level method calls and generic variable names, hence the patches are likely less natural.

Upon further inspection, we observed that while both GPT models successfully followed the exploration prompts for all tests, the open-source models struggled to adhere to response-format-constrained prompts. For instance, Qwen2.5-Coder-32B-Instruct occasionally responded directly with `Find Method Code: {className.methodName}` without substituting `className.methodName` with an actual method name. It also sometimes elaborated on the potential problem instead of directly deciding the correct workflow. Similarly, DeepSeek-Coder-33B-Instruct frequently attempted to fix the test directly during the exploration phase. Notably, DeepSeek-Coder-33B-Instruct failed to generate responses in the correct format in the exploration phase for 89 tests, while Qwen2.5-Coder-32B-Instruct also failed for 63 tests. Such limitations fall outside the scope of NIODebugger but could be mitigated by future advancements in open-source LLMs.

To mitigate the non-determinism of LLMs at a higher temperature, we repeated the experiment twice more using the same setup and parameters as described in the previous section for both NIODebugger-GPT-4 and NIODebugger-GPT-3.5-Turbo. We run only the GPT variants, because they have a much better performance than open-source models, and have no GPU costs. NIODebugger-GPT-4 achieved 89 and 103 correct patches, compared to 101 in the original experiment, while NIODebugger-GPT-3.5-Turbo achieved 74 and 70 correct patches, compared to 68 in the original experiment. All six runs with these two proprietary models showed performance exhibiting substantial improvement over the baselines.

Overall, our findings suggest that NIODebugger, when integrated with GPT models, achieves state-of-the-art performance, outperforming baseline techniques. The poorer performance of open-source models aligns with prior work on LLM-based flaky test repair [39] and general program repair [37], [38], where GPT models consistently outperform open-source LLMs. We present a detailed breakdown of the performance of our best-performing variant, NIODebugger-GPT-4, in Table II. The table includes the following columns: the project slug, the commit SHA used in our experiments, the count of non-commented, non-blank (NCNB) code lines in all Java source files, the total number of tests evaluated during the detection phase, the total number of non-commented test assertions across the entire test suite, the number of NIO tests identified by NIODebugger's detection phase, the number of NIO tests successfully fixed by NIODebugger-GPT-4, and the number of fixes that have already been accepted via pull request and merged into the project's main codebase.

TABLE I: Performance of NIODebugger Variants & Potential Baselines

NIODebugger Variant or Baseline	Correct Patches
NIODebugger-GPT-4	101 (58.72%)
NIODebugger-GPT-3.5-Turbo	68 (39.53%)
NIODebugger-Qwen2.5-Coder-32B-Instruct	27 (15.69%)
NIODebugger-DeepSeek-Coder-33B-Instruct	20 (11.63%)
ODRepair	59 (34.30%)
iFixFlakies	0 (0%)

C. RQ2: Contributions to Real-World Software Development

To assess the developer's attitude towards NIODebugger in real-world software development, we submitted pull requests for all 101 patches generated by our best-performing variant, NIODebugger-GPT-4. We made only minor adjustments to ensure the patches passed Checkstyle, without making logical modifications. Of these, 58 patches were accepted, 1 was rejected, and the remaining 42 patches are still pending.

```
public void testDeployClass() {
+   ReferenceSavingMyVerticle.myVerticles.clear();
    vertx.deployVerticle(//...).onComplete(onSuccess(
         deploymentId -> {
      ReferenceSavingMyVerticle.myVerticles.forEach(
          myVerticle -> {
        assertEquals(deploymentId, myVerticle.deploymentID)
            ;
        assertEquals(config, myVerticle.config);
        assertTrue(myVerticle.startCalled);
      });
    }));
    // ...
}
```

Fig. 7: DeploymentTest.java in eclipse-vertx/vert.x

TABLE II: Detailed View of Projects with NIO Tests and Fixer Performance of NIODebugger-GPT-4

Project	SHA	NCNB Lines	Total Tests	Test Assertions	NIO Tests	Correct Patches	Accepted Patches
apache/dubbo	20f252d	287009	6415	13555	34	19	19
apache/hadoop	ecf665c	1929672	10874	78983	31	14	7
kiegroup/jbpm	1558f0d	295347	3675	19965	23	11	0
sismics/docs	afa7885	24600	74	999	21	17	17
apache/cxf	eea3c9b	695144	9528	30777	15	3	3
alibaba/COLA	1a8c433	11654	69	167	12	11	0
brianfrankcooper/YCSB	ce3eb9c	25434	76	733	12	12	0
apache/wicket	58d953e	220299	2811	10727	6	5	5
spring-cloud/spring-cloud-netflix	2a8b7ed	12259	235	461	4	1	1
ebean-orm/ebean	d034821	222188	1069	11791	2	2	2
stleary/JSON-java	8983ca6	14263	647	1561	2	2	2
apache/rocketmq	b37d283	242774	1676	7652	2	1	1
apache/tika	f78dc99	173227	1982	10305	1	1	0
apache/tinkerpop	8bb5d16	172540	25269	14855	1	1	1
eclipse-vertx/vert.x	0eb288b	140880	4849	9596	1	1	0
apache/incubator-kie-optaplanner	8c2fb1e	208565	3963	10910	1	0	N/A
Red5/red5-server	eb75c16	64046	170	434	1	0	N/A
spring-projects/spring-retry	9442435	11468	387	903	1	0	N/A
stanfordnlp/CoreNLP	2460079	619842	1459	7035	1	0	N/A
winder/Universal-G-Code-Sender	445cd19	92947	750	2514	1	0	N/A

Of the 58 patches accepted, 52 were accepted directly, while 6 required changes before acceptance. In 5 of these 6 cases, the logic of the patches was approved, requiring only minor adjustments. These adjustments included adding a `try - finally` block or altering when state cleanup occurs within the test. The other change suggested by the developer involved making a non-idempotent function under test idempotent directly, thus addressing issues outside the scope of the test method.

Particularly noteworthy is that the rejected pull request [56] does not stem from incorrectness of the patch generated by NIODebugger. Figure 7 illustrates our patch. Specifically, the patch deploys a verticle to the global set `ReferenceSavingMyVerticle.myVerticles` but does not clean up after deployment. Consequently, in the second execution, `myVerticles` still contains a verticle from the previous run with a different deployment ID than the currently deployed one. This discrepancy causes the assertion `assertEquals(deploymentId, myVerticle.deploymentID)` to fail in one of the iterations of the `forEach` loop. In this scenario, the developer acknowledges the patch's correctness but rejects our PR because they do not consider such state pollution hazardous.

D. RQ3: Contributions of Key Components

This section examines the contribution of two key components of NIODebugger through an ablation study on NIODebugger-GPT-4, the best performing variant. We isolate source code extraction used for context exploration and dynamic analysis used during the detection phase and perform an ablation study. Table III presents the results, where "RSCE" denotes relevant source code extraction and "DA" refers to dynamic analysis. Each row indicates the number of correct patches generated.

It is evident that both dynamic analysis and relevant source code extraction are crucial for of NIODebugger, while relevant source code extraction is more indispensable.

TABLE III: Ablation Study of NIODebugger-GPT-4

Technique	Correct Patches
NIODebugger-GPT-4 without RSCE & DA	26 (15.12%)
NIODebugger-GPT-4 without RSCE	42 (24.42%)
NIODebugger-GPT-4 without DA	55 (31.98%)
NIODebugger-GPT-4	101 (58.72%)

We provide an example illustrating how NIODebugger cannot generate a patch without relevant source code extraction. Figure 8 depicts an NIO test in`apache/rocketmq`. The test fails in subsequent runs because it registers a `NothingFilter` without unregistering it. In the repeated run, an error occurs when registering the filter since a filter with the same identity already exists. Without relevant source code extraction, although the fixer LLM identifies the source of state pollution and guesses that a method `unRegister()` exists in `FilterFactory`, it fails to recognize that `unRegister()` requires its parameter as a filter-specific string, namely `"Nothing"` as defined in `NothingFilter`. As a result, the fixer mistakenly extracts `new NothingFilter()` as a local variable and passes it to `unRegister()`, resulting in a compilation error. Conversely, the relevant source code extraction rountine guides the exploration of the `FilterFactory` class, enabling a fix accepted by the developers [57].

```
public void testRegister() {
    FilterFactory.INSTANCE.register(new NothingFilter());
    // Other logic
+   FilterFactory.INSTANCE.unRegister("Nothing");
}
```

Fig. 8: FilterSpiTest.java in apache/rocketmq

For a more detailed overview, Table IV presents the number of tests with which each agentic workflow is utilized, as well as the number of NIO tests successfully fixed when the workflow is employed by our best-performing variant, NIODebugger-

GPT-4. "Find Class Code" was the most common workflow, followed by "Find Method Code". Both of the workflows lead to a significant portion of correct patches. Less frequent workflows, such as "Find Relevant File" and "Find Hypothesized Method", also contributed to correct fixes. Additionally, NIODebugger-GPT-4 decides that no additional code is needed in 34 cases, resolving 23 of them.

TABLE IV: Summary of Workflow Usage and Effectiveness

Workflow	# Tests Invoked	# Correct Patches
Find Class Code	75	48
Find Method Code	46	23
Directly Fixable	34	23
Find Relevant File	13	5
Find Hypothesized Method	4	2

We also provide an example where NIODebugger cannot generate a patch without dynamic analysis. Figure 9 shows an NIO test in the `hadoop` project. The method `registerSubCluster()` registers a call with a specified latency, while `getLatencySucceededCalls()` returns the mean latency of all registered successful calls. Starting from a fresh state, the assertion at line 6 passes in the first run with a mean latency of `100`, as only one call with latency `100` (at line 5) is recorded before the assertion. However, in the second execution of the test, the recorded history includes two previous calls (with latencies of `100` (at line 5) and `200` (at line 9)) from the first execution, and one call with latency `100` from the second execution. Consequently, the `getLatencySucceededCalls()` method in the assertion returns a mean latency of `133.33` (mean of 100, 200, and 100) in the first rerun, followed by `125` and `120` in subsequent runs. The changing error message provides valuable insights into state pollution accumulation. While NIODebugger without dynamic analysis generates an incorrect patch that assumes `getLatencySucceededCalls()` is simply incremented by 100 after the `registerSubCluster(100)` call, it successfully generates a patch for an approved PR [58] when dynamic analysis is enabled, as shown in Figure 9. This example underscores NIODebugger's capability to address complex NIO tests by analyzing stack traces from multiple test runs.

```
1     public void testSuccessfulCalls() {
2         long totalGoodBefore =
              FederationStateScoreClientMetrics.
              getNumSucceededCalls();
3   +     long meanLatencyBefore =
           FederationStateScoreClientMetrics.
           getLatencySucceededCalls();
4         //...
5         goodStatesStore.registerSubCluster(100);
6   -     assertEquals(100, FederationStateScoreClientMetrics.
           getLatencySucceededCalls());
7   +     assertEquals((totalGoodBefore * meanLatencyBefore +
           100) / (totalGoodBefore + 1),
           FederationStateScoreClientMetrics.
           getLatencySucceededCalls());
8         //...
9         goodStatesStore.registerSubCluster(200);
10        //...
11    }
```

Fig. 9: TestFederationStateStoreClientMetrics.java in apache/hadoop

V. Threats to Validity

We identify several potential threats to the validity of our approach and evaluation, and describe how we addressed each:

1) **Data Leakage**: Our evaluation includes closed-source GPT models with unknown training data, which may include the projects we analyze. Although we evaluate NIO tests with no prior fixes, the model's prior exposure to the codebase might lead to misleadingly high performance. However, the ablation study in RQ3 addresses this concern by showing that the success rate of GPT-4-generated patches decreases significantly when dynamic analysis or relevant source code extraction are removed.
2) **Validity of Patches**: There is a risk that patches generated by NIODebugger may not be ideal. To mitigate this, we submitted pull requests (PRs) for all patches. Out of 59 patches reviewed by developers, 58 were accepted. The one rejection was not due to incorrectness but rather the developer's reluctance to address state pollution.
3) **Scalability**: Our evaluation is limited to Java projects, raising concerns about the applicability to NIO tests in other programming languages. We address this by discussing the language-specific aspects of our framework and explaining why extending support to multiple languages is practical.
4) **Non-determinism**: The LLM response is inherently non-deterministic, particularly due to the higher temperatures. To mitigate this, we repeated the experiment twice for the two best-performing variants and observed consistently better performance than the baselines.

VI. Conclusion

In conclusion, this paper presents NIODebugger, a pioneering framework that effectively addresses non-idempotent-outcome (NIO) flaky tests using an LLM-based agent. By integrating dynamic analysis during detection phase and agent-based relevant source code extraction, NIODebugger demonstrates strong performance in detecting and repairing NIO flaky tests in popular open-source projects, with numerous patches accepted by the community. While NIODebugger can interface with various off-the-shelf LLMs, it achieves more promising results with proprietary GPT-based models. The challenge of enabling state-of-the-art open-source LLMs to generalize to agentic workflows remains well recognized and continues to be an active area of research. We anticipate that future advancements in open-source LLMs will enhance their effectiveness when integrated with NIODebugger.

Acknowledgments

We thank Darko Marinov and Lingming Zhang for their valuable contributions to the development and refinement of NIODebugger. We also appreciate Feiqian Yang's assistance with the experiments. Additionally, we thank Runxiang Cheng and Yang Chen for their insightful feedback, which significantly improved the clarity and presentation of this paper.

References

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014.

[2] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM TOSEM*, 2021.

[3] W. Zheng, G. Liu, M. Zhang, X. Chen, and W. Zhao, "Research progress of flaky tests," in *SANER*, 2021.

[4] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, M. Cordy, and Y. L. Traon, "What made this test flake? pinpointing classes responsible for test flakiness," in *ICSME*, 2022.

[5] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. Le Traon, "Flaky-Cat: Predicting flaky tests categories using few-shot learning," in *AST*, 2023.

[6] G. Haben, S. Habchi, J. Micco, M. Harman, M. Papadakis, M. Cordy, and Y. Le Traon, "The importance of accounting for execution failures when predicting test flakiness," in *ASE*, 2024.

[7] N. Hashemi, A. Tahir, and S. Rasheed, "An empirical study of flaky tests in JavaScript," in *ICSME*, 2022.

[8] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *ICSE*, 2020.

[9] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in *ESEC/FSE*, 2019.

[10] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at apple," in *ICSE-SEIP*, 2020.

[11] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *ISSTA*, 2019.

[12] Y. Qin, S. Wang, K. Liu, B. Lin, H. Wu, L. Li, X. Mao, and T. Bissyandé, "Peeler: Learning to effectively predict flakiness without running tests," in *ICSME*, 2022.

[13] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," *IEEE TSE*, 2023.

[14] M. Gruber, M. Heine, N. Oster, M. Philippsen, and G. Fraser, "Practical flaky test prediction using common code evolution and test history data," in *ICST*, 2023.

[15] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting flakiness without rerunning tests," in *ICSE*, 2021.

[16] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *MSR*, 2020.

[17] A. Ahmad, O. Leifler, and K. Sandahl, "An evaluation of machine learning methods for predicting flaky tests," in *International Workshop on Quantitative Approaches to Software Quality (APSEC QuASoQ)*, 2020.

[18] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019.

[19] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications," in *FSE Tool Demo*, 2016.

[20] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *ICSE*, 2018.

[21] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "What do developer-repaired flaky tests tell us about the effectiveness of automated flaky test detection?" in *AST*, 2022.

[22] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *ESEC/FSE*, 2019.

[23] R. Wang, Y. Chen, and W. Lam, "iPFlakies: A framework for detecting and fixing python order-dependent flaky tests," in *ICSE Tool Demo*, 2022.

[24] C. Li, C. Zhu, W. Wang, and A. Shi, "Repairing order-dependent flaky tests via test generation," in *ICSE*, 2022.

[25] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, "Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications," in *ICSE*, 2021.

[26] Y. Pei, J. Sohn, S. Habchi, and M. Papadakis, "Non-flaky and nearly optimal time-based treatment of asynchronous wait web tests," *ACM TOSEM*, 2025.

[27] F. Palomba and A. Zaidman, "Notice of retraction: Does refactoring of test smells induce fixing flaky tests?" in *ICSME*, 2017.

[28] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *ISSTA*, 2014.

[29] JUnit Team, "JUnit," 2024. [Online]. Available: https://junit.org/

[30] A. Wei, P. Yi, Z. Li, T. Xie, D. Marinov, and W. Lam, "Preempting flaky tests via non-idempotent-outcome tests," in *ICSE*, 2022.

[31] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *ESEC/FSE*, 2022.

[32] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *ICSE*, 2023.

[33] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," in *ISSTA*, 2024.

[34] J. A. Prenner, H. Babii, and R. Robbes, "Can OpenAI's codex fix bugs? an evaluation on QuixBugs," in *International Workshop on Automated Program Repair (APR)*, 2022.

[35] S. D. Kolak, R. Martins, C. L. Goues, and V. J. Hellendoorn, "Patch generation with language models: Feasibility and scaling behavior," in *Deep Learning for Code Workshop (DL4C)*, 2022.

[36] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An autonomous, llm-based agent for program repair," *arXiv:2403.17134*, 2024.

[37] C. Lee, C. S. Xia, J. tse Huang, Z. Zhu, L. Zhang, and M. R. Lyu, "A unified debugging approach via LLM-Based multi-agent synergy," *arXiv:2404.17153*, 2024.

[38] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," *arXiv:2404.05427*, 2024.

[39] Y. Chen and R. Jabbarvand, "Neurosymbolic repair of test flakiness," in *ISSTA*, 2024.

[40] S. Fatima, H. Hemmati, and L. C. Briand, "FlakyFix: Using large language models for predicting flaky test fix categories and test code repair," *IEEE TSE*, 2024.

[41] Y. Chen and R. Jabbarvand, "Can ChatGPT repair non-order-dependent flaky tests?" in *Proceedings of the 1st International Workshop on Flaky Tests (FTW)*, 2024.

[42] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST*, 2016.

[43] W. Lam, "International Dataset of Flaky Tests (IDoFT)," 2020. [Online]. Available: https://github.com/TestingResearchIllinois/idoft

[44] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *ISSTA Tool Demo*, 2014.

[45] Snyk, "JVM ecosystem report 2021," 2021. [Online]. Available: https://snyk.io/reports/jvm-ecosystem-report-2021/

[46] JetBrains, "The State of Developer Ecosystem 2021," https://www.jetbrains.com/lp/devecosystem-2021/java/, 2021.

[47] The Apache Software Foundation (ASF), "Maven surefire plugin," https://maven.apache.org/surefire/, 2024, accessed: 2024-07-31.

[48] A. W. Appel, *Modern Compiler Implementation in C*. Cambridge University Press, 1997.

[49] JavaParser community, "JavaParser," 2024. [Online]. Available: https://javaparser.org/

[50] Python Software Foundation, "ast: Abstract Syntax Trees in Python," 2024. [Online]. Available: https://docs.python.org/3/library/ast.html

[51] Clang Team, *Clang: A C language family frontend for LLVM*, 2024. [Online]. Available: https://clang.llvm.org/

[52] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, 1966.

[53] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence," *arXiv:2401.14196*, 2024.

[54] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, K. Dang, Y. Fan, Y. Zhang, A. Yang, R. Men, F. Huang, B. Zheng, Y. Miao, S. Quan, Y. Feng, X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-Coder technical report," *arXiv:2409.12186*, 2024.

[55] Aider-AI, "Aider LLM Leaderboards: Code editing leaderboard," available: https://aider.chat/docs/leaderboards/, accessed: 2024-11-26.

[56] Pull Request #5190 in the Vert.x GitHub Repository, "Fixed non-idempotent test 'DeploymentTesttestDeployClass'," https://github.com/eclipse-vertx/vert.x/pull/5190, April 2024.

[57] Pull Request #8093 in the RocketMQ GitHub Repository, "[ISSUE 8092] Fixed non-idempotent test," https://github.com/apache/rocketmq/pull/8093, May 2024.

[58] Pull Request #6793 in the Hadoop GitHub Repository, "YARN-11694. Fixed non-idempotent unit tests in the Yarn Module," https://github.com/apache/hadoop/pull/6793, May 2024.