TDRepro: A Neurosymbolic Approach to Reproducing Timing-Dependent Flaky Test Failures

Shanto Rahman
The University of Texas at Austin
Austin, Texas, USA
shanto.rahman@utexas.edu

Talank Baral
George Mason University
Fairfax, Virginia, USA
tbaral@gmu.edu

August Shi
The University of Texas at Austin
Austin, Texas, USA
august@utexas.edu

Wing Lam
George Mason University
Fairfax, Virginia, USA
winglam@gmu.edu

Abstract

Tests that nondeterministically pass or fail on the same version of code are known as flaky tests, and their failures substantially hinder regression testing and debugging. Reproducing failures from timing-dependent (TD) flaky tests is particularly challenging because these failures depend on rare thread interleavings that may not occur even after running the test many times. Prior approaches reproduce such failures by searching for locations at which to inject delays during execution, but this search process can be expensive.

We present TDRepro, an automated approach that leverages large-language models (LLMs) and program analysis to efficiently reproduce TD-test failures. TDRepro first identifies relevant methods by collecting executed methods during the test run and identifying those that execute concurrently with other methods. It then ranks these methods by computing the embedding similarity between each method's code and the observed failure log. Finally, TDRepro uses an LLM to identify candidate locations for delay injection within the top-ranked methods. TDRepro validates these candidates by injecting delays and rerunning the test, iteratively refining the candidate locations based on unsuccessful attempts until the target failure is reproduced or the maximum number of iterations is reached. We evaluate TDRepro on 82 TD tests across 25 modules from open-source Java Maven projects. TDRepro reproduces 100% more failures than the prior state-of-the-art approach, FlakeRake, and achieves more reliable reproduction, with 55 tests reaching a failure reproduction rate of >99%, compared with 30 tests for FlakeRake. Repeatedly reproducing failures with TDRepro requires 1482.1× and 10.4× less time and costs 888.4× and 9.6× less than repeatedly rerunning tests and using FlakeRake, respectively.

CCS Concepts

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

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834384

Keywords

Flaky tests, timing-dependent tests, failure reproduction, LLMs

ACM Reference Format:
Shanto Rahman, Talank Baral, August Shi, and Wing Lam. 2026. TDRepro: A Neurosymbolic Approach to Reproducing Timing-Dependent Flaky Test Failures. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834384

1 Introduction

Regression testing is the common practice of running tests after every code change to check whether the change introduced any bugs [17, 18, 43]. However, this practice suffers from the presence of *flaky tests*, which are tests that can nondeterministically pass or fail on the same version of code [14, 26]. Failures from these tests can mislead developers about their code changes and waste developers' time to debug. To understand the root cause of the flakiness and to validate whether a patch successfully removed the flakiness, developers need to reproduce the flaky-test failure, but doing so is often challenging as the test fails nondeterministically.

Reliably reproducing flaky-test failures is often difficult as simply rerunning a flaky test may rarely observe the same test failure. Prior work [10] found that even after rerunning tests 10,000 times, certain failures still rarely occurred. A test can be flaky because of many reasons, and prior work identified timing-dependence (TD) as one of the most prominent reasons, i.e., tests that depend on execution timing [14, 26, 34, 35]. To help reproduce failures from TD tests, Rahman et al. proposed FlakeRake [35], an approach that injects delays at strategic locations during test execution to more easily induce specific thread interleavings that can cause TD-test failures. While FlakeRake improved the reproducibility of TD-test failures, it relies on expensive instrumentation and multiple test reruns.

We propose TDRepro, a neurosymbolic approach that reproduces TD-test failures reliably by combining neural reasoning with program analysis and execution-based validation. Specifically, program analysis uses execution tracing, coverage information, and concurrency information to constrain the search space of relevant methods. Neural components then rank these methods using embeddings and employ an LLM to identify promising locations for delay injection. These neural outputs are guided by program-analysis

information and subsequently validated through source-level delay injection and concrete test executions. Thus, rather than using neural and symbolic components independently, TDRepro uses program analysis and program execution to constrain, inform, and validate neural outputs. Our insight is that the cost of an approach like FlakeRake comes from its search process that needs to rerun the test many times to check whether any injected delay makes the test fail. Hence, we use an LLM to predict and rank potential candidate locations where injecting a delay can make the test fail. If the LLM can accurately inject delays (assisted by program analysis to provide information and context to help prediction), then we can dramatically reduce the number of candidate locations to try before reproducing the failure.

Given a TD test, its failure log, and the project codebase, TDRepro first executes the test to collect method-level coverage to focus only on the executed methods. It also identifies which methods execute concurrently with one another through light instrumentation to further narrow the search space. TDRepro then encodes the failure log, consisting of the failure message and stack trace, and each candidate method into embeddings, which it then uses to compute cosine similarities between each candidate method and the failure log to estimate semantic relevance. Using these similarity scores, TDRepro ranks the most relevant methods and prompts an LLM to rank candidate locations within these methods, where a method may result in one or multiple candidate locations. Injecting a delay at a candidate location may trigger the TD-test failure. For the top-ranked locations returned by the LLM, we rerun the test while injecting a delay at each location to validate whether the failure is reproduced. If the failure is not reproduced from any candidate locations, TDRepro applies a feedback-driven refinement strategy to prompt the LLM again for new candidates, up to a maximum of five attempts. Once TDRepro observes a failure, TDRepro reports the delay location used to reproduce the failure, where each location is delayed for five seconds. This approach combines dynamic execution signals, semantic embeddings, and LLM-guided reasoning to reproduce TD-test failures.

We evaluate TDRepro on a dataset of 25 Maven project modules, consisting of 82 TD tests, taken from prior work [2, 35]. Following the prior work [35], we run each test in isolation (Isolated Rerun) to obtain the baseline failure log that TDRepro aims to reproduce. Our evaluation shows that TDRepro reproduces 100% more flaky-test failures than FlakeRake, successfully reproducing 60 failures compared to the 30 reproduced by FlakeRake. Moreover, TDRepro achieves a higher failure rate than both FlakeRake and Isolated Rerun, reaching a failure rate of >99% for 55 tests. In comparison, FlakeRake achieves this failure rate level for only 30 tests, while Isolated Rerun achieves it for none. We also compare the cost of reproducing failures with TDRepro and Isolated Rerun. TDRepro incurs a one-time cost to search for a delay location, after which each reproduction is inexpensive. Thus, repeatedly observing the same failure is much cheaper with TDRepro than with Isolated Rerun, which incurs the cost of multiple test reruns for each failure reproduction. To assess the role of different LLMs used within TDRepro, we conduct different ablation studies. By default, TDRepro uses GPT-2 for ranking candidate methods and GPT-5.6 for recommending specific locations to inject delays, which is the combination that achieves the best overall performance. As GPT-5.6 is a proprietary model, we also explore open-source alternatives by replacing it with two open-source models, and find that the open-source models also perform well within TDRepro.

Our paper makes the following main contributions:

- We present TDRepro, a neurosymbolic-based approach that uses LLMs to automatically identify locations to inject delays to reproduce TD-test failures. TDRepro reduces the search space by leveraging embedding-based similarity between the failure log and project methods to rank candidate methods and identify where in these methods to inject delays.
- We provide a comprehensive analysis and ablation study covering: (i) the reliability of failure reproduction; (ii) design choices, such as embedding-guided ranking versus single-shot prompting, and embedding model sensitivity (GPT-2-Medium embedding, Qwen/Qwen2-7B embedding, and meta-llama/Meta-Llama-3-8B embedding); (iii) model substitution, replacing GPT-5.6 with popular open-source LLMs; and (iv) hyperparameter sensitivity.
- We evaluate TDRepro on 82 flaky tests across 25 Maven project modules. TDRepro reproduces 100% more flaky-test failures than FlakeRake. The reproduced failures are more reliable, with 55 tests achieving a failure reproduction rate of >99%, compared with 30 tests for FlakeRake and none for Isolated Rerun. The cost of reproducing failures is much lower than that of Isolated Rerun and FlakeRake.

2 Background

2.1 Reproducing TD-Test Failures

To repair a flaky test, a developer often needs to reproduce its failure to better understand and debug the reason for the failure. Furthermore, after implementing a patch, a developer needs to confirm that the failure is no longer possible or its occurrence is substantially reduced with the patch. During this debug and repair process, nondeterministic flaky-test failures can mislead the developer, so the ability to reliably observe the failure is of high importance. A timing-dependent (TD) flaky test fails due to a difference in expected execution timing [35], such as expecting a separate thread to finish computations within a certain time, and so the test fails if those computations are not completed in time. Prior work [14, 26] identified TD tests as the most prominent type (a combination of asynchronous wait and concurrency flaky tests within their categorizations). There are two main approaches for reproducing TD-test failures: (1) rerunning the test many times until it fails or (2) introducing timing delays at strategic locations to make the test fail more easily.

2.1.1 Isolated Rerun. To confirm whether a test is flaky, developers commonly rely on rerunning the test until the test both passes and fails across the multiple runs [23, 28]. Prior work has found that rerunning a test in isolation, i.e., on its own within separate processes between reruns, is most likely at forcing the test to fail [11, 29]. We refer to the approach of rerunning tests in isolation to reproduce a flaky-test failure as *Isolated Rerun*.

While rerunning the test in isolation will eventually confirm the test is flaky, it may require many reruns just to observe a failure. Alshammari et al. previously found that some flaky tests may require

```
1  public class FileAppenderResilienceTest {
2    @Test public void smoke() {
3      Runner runner = new Runner(fa);
4      Thread t = new Thread(runner);
5      t.start();
6      double delayCoefficient = 2.0;
7      for (int i = 0; i < 5; i++) {
8        Thread.sleep(RecoveryCoordinator.COEFFICIENT_MIN
9          * delayCoefficient);
10       closeLogFileOnPurpose();
11     }
12     ...
13     double bestCaseRatio = 1 / delayCoefficient;
14     // expect to lose at most 35% of the events
15     double lossiness = 0.35;
16     double resiliance = (1 - lossiness);
17     ResilienceUtil.verify(logfileStr, "...",
18       runner.getCounter(), bestCaseRatio * resiliance);
19   }
20 }
21 public class RecoveryCoordinator {
22   ...
23   public final static long COEFFICIENT_MIN = 20;
24   private long getBackoffCoefficient() {
25     long currentCoeff = backOffCoefficient;
26     if (backOffCoefficient < BACKOFF_COEFFICIENT_MAX)
27       backOffCoefficient *= 4;
28     return currentCoeff;
29   }
30 }
```

Figure 1: Example flaky test from `qos-ch/logback` [6].

rerunning upwards to 10,000 times just to observe a failure [10]. Despite its simplicity, Isolated Rerun can be expensive due to the large number of reruns needed, especially if a developer needs to observe the failure multiple times as they debug the flaky test [35]. A recent study [9] on developer debugging found that developers run tests to reproduce failures an average of six times and up to 32 times when debugging.

2.1.2 FlakeRake. Inserting delays at strategic locations during test execution can make TD tests more likely to fail [24, 35, 36], i.e., forcing key operations for Async wait flaky tests to run too slow or expose new thread interleavings that can lead to concurrency flaky-test failures. Rahman et al. previously proposed FlakeRake as a way to identify the locations where one can inject delays that can more reliably reproduce a TD-test failure during a test run [35].

Given a TD test, FlakeRake identifies timing-related APIs used during test execution and inserts delays at those locations during the execution of a specific thread. Once it confirms that rerunning the TD test with injected delays can lead to a test failure, FlakeRake then conducts a search process to minimize the set of delay locations down to a minimal set that still reproduces the failure. FlakeRake finally outputs a configuration, consisting of where to inject a delay for a specific thread along with the amount of delay. A developer can then use this configuration to construct a reproduction script to reproduce the failure reliably.

2.2 Example

Figure 1 illustrates a simplified version of an example flaky test from project `qos-ch/logback` used in our evaluation. The flaky test is `FileAppenderResilienceTest.smoke()`, a previously known

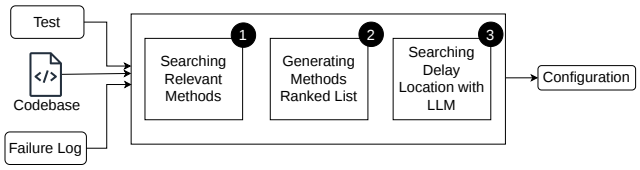

Figure 2: Overview of TDRepro.

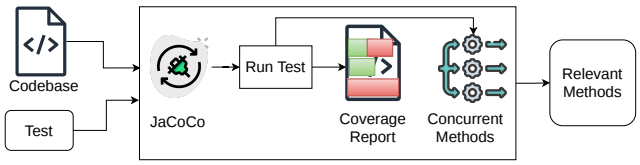

Figure 3: Searching relevant methods for a test.

flaky test from a public dataset [2], meaning prior work observed the test flakiness when rerunning it on the same version of code.

The `smoke` test in `FileAppenderResilienceTest` evaluates the logging system's ability to recover from intentional failures during concurrent logging. It launches a background thread at Line 5 that continuously logs messages, while the main thread intermittently and deliberately closes the underlying log file's `FileChannel` at Line 10. This setup is meant to simulate real-world I/O failures.

The test finally terminates the logging thread and verifies the system's resilience by checking the continuity and integrity of the logged messages. Using `ResilienceUtil.verify(...)` at Line 17, the test asserts that, while some message loss is acceptable, the system must recover quick enough to not lose more than 35% of the messages. In the `smoke` test, the main thread repeatedly calls `Thread.sleep(...)` using a delay derived from the `getBackoffCoefficient` method at Line 8. `getBackoffCoefficient` implements an exponential backoff strategy for retrying operations. Meanwhile, the background thread logs messages as the main thread intermittently disrupts the log file.

Injecting a delay at the beginning of `getBackoffCoefficient` slows down the main thread's call to `closeLogFileOnPurpose`, reducing the frequency of file disruptions. As a result, fewer messages are lost during logging. However, the test expects some message loss to exercise the recovery mechanism at Line 17. When the percentage of lost messages becomes too low, the test fails because the intended failure-and-recovery behavior is insufficiently exercised.

When we apply TDRepro on this test, TDRepro suggests the correct delay location quickly, using only 22.3 seconds to reproduce the failure. On the other hand, FlakeRake takes 5565.1 seconds to reproduce the failure. We find that Isolated Rerun takes even more time, requiring 18695.2 seconds. Our results show that TDRepro can be 99.6% and 99.9% faster compared to FlakeRake and Isolated Rerun, respectively, at reproducing this test failure.

3 TDRepro

We present TDRepro, an automated approach to reproduce timing-dependent (TD) flaky-test failures. Prior approaches, such as FlakeRake [35], reproduce TD-test failures by injecting delays during test execution. While effective, these approaches are often expensive because they rely on exploring and trying to inject delays at many possible delay locations during execution. Our intuition is that a large-language model (LLM), when combined with program

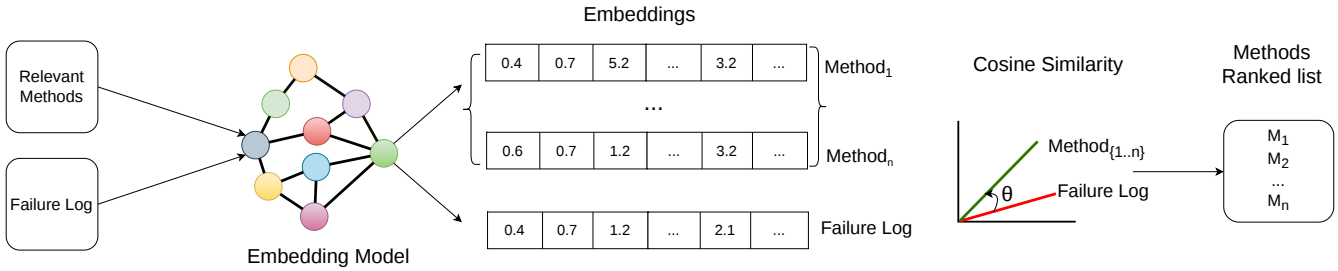

Figure 4: Generating methods ranked list.

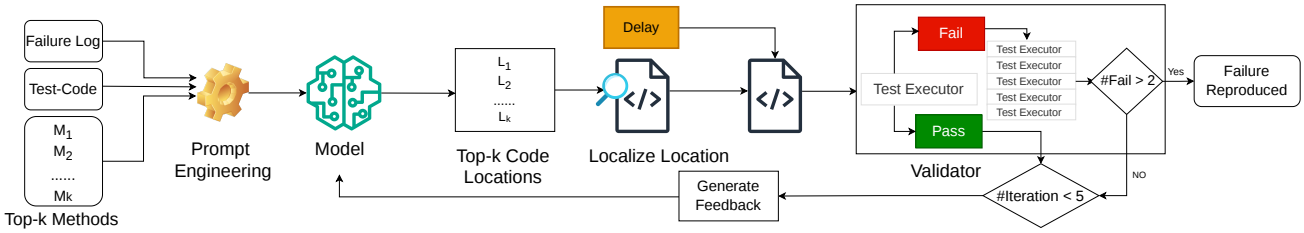

Figure 5: Searching delay location with LLM.

analysis, can substantially improve the efficiency of searching for delay locations. Specifically, the LLM can help prioritize likely delay locations and allow us to avoid many of the expensive search steps required by prior approaches.

Figure 2 shows the high-level architecture of TDRepro. The inputs to TDRepro include the program-under-test (codebase), test code, and a previously observed failure log containing the desired failure to reproduce. TDRepro consists of three main components: (1) a method search component that finds the set of potentially relevant methods for a given test, (2) a ranking component that ranks these methods based on their relevance to the flaky-test failure, and (3) an LLM-based pipeline that identifies candidate delay locations that validates whether the injected delays at those locations reproduce the failure, and iteratively refines the suggested delay locations based on the validation results. The final output of TDRepro is a failure-reproducible configuration, namely, a specific location in the code and a delay to inject at this location such that injecting the delay at this location causes the test to fail with the same failure as the one given.

3.1 Method Search

Figure 3 illustrates the process of finding the relevant methods in which to inject a delay to reproduce the TD-test failure. We first resolve any detected dependency errors that might hinder test execution, namely, issues that can lead to compilation errors. We then modify the project's build configuration (e.g., `pom.xml` for Maven projects) to enable code coverage collection. Specifically, we integrate JaCoCo [4] to collect coverage information for Java, Maven-based projects. We then execute the flaky test under this configuration to obtain coverage data for the target test and parse the resulting report to extract the set of methods executed during the test run. We then collect the methods executed concurrently while running the test by following a similar approach used in prior work [32] that finds concurrent methods for detecting concurrency bugs by instrumenting method start and end points to determine when multiple methods are executing concurrently. We perform this instrumentation at the bytecode level and record method-entry and method-exit events along with the executing thread to identify methods whose executions overlap across different threads. This design is motivated by prior work that found TD-test failures primarily arise from interactions among concurrent threads [26].

3.2 Method Ranking

Although collecting just the relevant methods by the test reduces the search space from the full set of methods in the project, the number of relevant methods still remains substantially large for many tests in our evaluation (average of 417.0 per test, as shown in Table 1). Prior work found that typically one can reproduce a TD-test failure through just one delay location [35, 36], so searching for that one delay location across all these methods is costly. To address this challenge, we aim to rank the methods based on their likelihood of containing a valid delay location for reproducing a given TD-test failure. We then consider the top-K methods based on this ranking for future steps.

We perform this ranking between the initial set of relevant methods obtained from the previous method search component using advanced embedding models. As shown in Figure 4, the key inputs to this stage include the collected relevant method bodies and the raw textual failure log from the failing execution. The embedding model, i.e., a pre-trained LLM tailored for code or natural language, processes each of these inputs, transforming them into fixed-dimensional numerical representations, known as embeddings. In our experiments, we use the GPT-2 embedding model [1], which generates 1024-dimensional vectors for each input. We choose GPT-2 because it is lightweight and less memory-intensive, and it also preserves fine-grained lexical cues (identifiers, function names, and error strings). This transformation is essential, as it maps both textual (e.g., failure log) and structural (e.g., method code) information into a continuous vector space where semantic relationships are preserved, enabling meaningful quantitative comparisons.

LLM Prompt for Finding Delay Location

```
You are an expert Java developer. You will be provided with:
1. A non-deterministic test's failure log.
2. A list of methods.
3. The test code itself.
Your task:
- Carefully analyze each provided method
and identify the single most likely location
...
**Rules:**
1. Output a ranked list of 10 code locations
...
<Input>
    <Failure>
        ...
    </Failure>
    <Methods-Under-Test>
        ...
        ...
    </Methods-Under-Test>
    <Test-Code>
        ...
    </Test-Code>
</Input>
```

Figure 6: Prompt for finding delay locations. A more concrete example is on our website [7].

LLM Output for Delay Location

```
ch.qos.logback.core.recovery.ResilientOutputStreamBase:write:
  ([BII)V:46 ( attemptRecovery();)
ch.qos.logback.core.recovery.ResilientOutputStreamBase:write:
  ([BII)V:48 ( return;
ch.qos.logback.core.recovery.RecoveryCoordinator:isTooSoon:()
  Z:28 ( long now = getCurrentTime();)
```

Figure 7: Candidate lines for delay injection.

Once we generate embeddings for the different inputs, we quantify their semantic similarity using cosine similarity. Specifically, we compute the cosine similarity between each relevant method and the failure log. Cosine similarity measures the angle between two non-zero vectors in an inner product space; values closer to 1 indicate high semantic similarity, while values near 0 indicate little to no similarity. Cosine similarity is particularly useful in our context, as it captures the directional alignment between embeddings, thereby revealing how closely a method's behavior aligns with the language and patterns observed in the failure log. For instance, a high similarity between a method and the failure log suggests that the method is likely related to the failure.

3.3 Delay Location Search with LLM

Given the top-K most relevant methods from the previous component, we now aim to pinpoint an exact location where injecting a delay can reproduce the given failure. Figure 5 illustrates this process, which uses the provided relevant methods and failure log to construct prompts for the LLM to generate a configuration, i.e., a delay location where injecting a delay reproduces the failure.

3.3.1 Prompt Construction. Using the top-K most relevant methods for a TD-test failure, we construct a prompt that guides the LLM to identify candidate delay locations. The prompt includes natural language instructions that guide the LLM to analyze each method body and identify the likely location to inject a delay that can trigger the given failure. Figure 6 illustrates our prompt template.

More specifically, we craft a task-specific prompt that includes: (1) the failure log, (2) the test code, and (3) the top-K most relevant methods. For each method, we format the structure that we provide to the LLM to include its class name, method name, descriptor, line range, and entire method body. Crucially, we annotate each line of the method body with its corresponding source file line number, enabling the LLM to reason about and output the exact location in which to inject a delay. The prompt explicitly instructs the LLM to output a ranked list of potential delay locations in the format: `Class:Method:Descriptor:FileLineNumber` as shown in Figure 7, for the example in Section 2.2. The LLM outputs the top-L locations within the given methods where it predicts that injecting a delay at that location reproduces the failure. We enforce strict constraints in the prompt, allowing only complete statements, i.e., valid delay locations are never partial lines, continuations, or insertions within multi-line constructs.

3.3.2 Prompt Execution and Injecting Delay. Once we generate a prompt using all the information for a specific TD test, we structure a conversation using the OpenAI Chat API [3]. In the prompt, we have a system message that defines the task and its constraints, followed by a user message containing the full prompt, including the failure log, test code, and candidate methods. This conversation is submitted to the LLM within a loop that includes a retry mechanism to ensure robustness against API failures.

The model's response is then appended to the conversation history as an assistant message, enabling future refinement. We specifically extract structured output enclosed between the `<Output>` and `</Output>` tags. Each extracted line is expected to follow the format: `Class:Method:Descriptor:LineNumber`. Each of these lines represents a candidate configuration, i.e., a delay location where the model believes injecting a delay there should reproduce the failure.

3.3.3 Validation. For each candidate configuration, we rerun the test in isolation to validate whether injecting a delay at that location truly makes the test fail more reliably. We parse each candidate configuration to extract the class name, method name, descriptor, target line number, and corresponding source code. Using this information, we inject a delay at the given location (in Java, we would insert a call to `Thread.sleep(...)` at each location, one at a time). We set the amount of time to delay as five seconds, aligning with the delay amount used in prior work, FlakeRake [35].

By default, we run the test five times. We run the test with a default three minutes timeout, which is substantially longer than individual test runtimes in our dataset. The timeout helps deal with indefinite executions from injected delays. We collect a failure log, if any, after the test execution. We consider a candidate configuration to be valid if running the test with it results in any failures for at least three out of the five runs, following the same heuristic used in prior work [35]. We report any valid configurations. However, if we do not observe the same failure as the baseline failure at least three times, we provide feedback to the LLM to get better candidate locations, as shown in Figure 5.

3.3.4 Feedback. In case we cannot obtain a valid candidate location, we apply a feedback-driven refinement process to enable the model to make another attempt at generating candidate locations,

Follow-up Prompt with Feedback for the LLM

```
Your previous suggestion did not reproduce the failure.
Here is what is already tried:
Method: <MethodName>
Location(s): <MethodCodeWithLineNumbers>
Suggest a new batch of 10 locations for delay injection.
```

Figure 8: Follow-up prompt incorporating failure reproduction feedback and refinement instructions.

up to a set maximum limit of five attempts. We construct a structured feedback prompt summarizing prior attempts after trying all candidate locations the LLM last outputted, shown in Figure 8.

This feedback includes: (1) a ranked list of methods where we have already tried injecting delays within, and (2) the candidate locations used in the unsuccessful attempt. We also prompt the LLM explicitly to instruct it to avoid repeating earlier suggestions and to propose new delay locations, either in a different method or at an untried location within the same method. This message is appended to the ongoing conversation history, allowing the LLM to maintain context across iterations and refine its reasoning accordingly.

4 Experimental Setup

We address the following research questions:

- **RQ1**: How effective is TDRepro at reproducing TD-test failures compared with prior approaches?
- **RQ2**: How reliably can the same TD-test failures be reproduced across repeated executions?
- **RQ3**: What is the cost of reproducing failures using TDRepro compared with prior approaches?
- **RQ4**: How do different design choices affect TDRepro?

We address RQ1 to evaluate the number of TD-test failures reproduced by each approach. We address RQ2 to assess how reliably each approach can reproduce the same TD-test failures across repeated test runs. We address RQ3 to investigate the time and cost associated with each approach when reproducing the TD-test failures. Finally, we address RQ4 to investigate how choices of different components of TDRepro impact the performance of TDRepro.

4.1 Evaluation Dataset

We conduct our evaluation on TD tests collected from open-source projects that are Java and Maven-based. The tests were previously identified by FlakeRake [35] and contained in a public dataset, IDoFT [2]. We first rerun each TD test in our environment 10,000 times to confirm the presence of flakiness. We include only tests for which we observe flaky failures during our reruns, ensuring that our evaluation focuses only on failures that can be reproduced in our experimental environment.

FlakeRake. Rahman et al. [35] initially considered 811 tests that were previously identified by FlakeFlagger [10]. As a baseline ("Isolated Rerun"), representing the approach commonly used by developers to reproduce flaky-test failures, Rahman et al. [35] reruns each test in isolation 10,000 times to detect flakiness. From this process, Rahman et al. identified 160 flaky tests. We attempt to include these 160 flaky tests in our evaluation. We run each test 10,000 times, and we find that only 102 tests failed at least once. We find that 59 of these tests are environment-dependent, meaning their failures are not due to the tests themselves but rather due to the environment in which they are run, e.g., the failure messages indicate issues with port binding (i.e., throwing a `BindException`), invalid port assignments (i.e., reporting port number being out of range), or connection failures (i.e., reporting `Connection refused`). These failure patterns suggest that the tests are primarily affected by environmental or resource-related issues, such as leftover processes, occupied ports, or unsuccessful service initialization, rather than genuine timing-dependent behavior. Therefore, we exclude those 59 tests from our evaluation, leaving us with 43 tests from this dataset to use in our evaluation.

IDoFT. From IDoFT [2], we find 179 flaky tests are categorized as TD tests. We run each test 10,000 times, and find that 68 tests fail at least once. Among these tests, 29 overlap with the flaky tests already included from the FlakeRake dataset. Therefore, we include an additional 39 TD tests from IDoFT.

After combining tests from these two sources, our dataset contains 82 TD tests across 25 Maven modules[1]. Table 1 summarizes the characteristics of these modules. The "Tests" column reports the number of TD tests in each module, while "Exec. Methods" reports, for each module, the average number of methods executed by a TD test in that module. Across all TD tests in our dataset, a test executes 417.0 methods on average, indicating a large search space for potential delay injection when reproducing TD-test failures.

4.2 Baselines and Configuration

We evaluate TDRepro against two baselines, Isolated Rerun and FlakeRake (Section 2.1). For Isolated Rerun, we rerun each TD test in isolation 10,000 times. For FlakeRake, we use the tool as is, targeting a given TD test. We run FlakeRake with the default configuration in a Docker container with 16GB of RAM and 4 CPU cores. When running TDRepro, we consider the top $K = 10$ methods and $L = 10$ ranked candidate locations per prompt. We choose these values based on our ablation study in Section 5.4.

5 Evaluation

5.1 RQ1: Number of Reproduced Failures

Table 2 shows the number of TD-test failures detected (column "# Tests Failure Detected"), and reproduced (column "# Tests Failure Reproduced") by each approach. We consider a test's failure to be "detected" if the approach triggers at least one failure that fails at least three out of five runs, regardless of whether it matches the baseline failure from Isolated Rerun. We consider a failure "reproduced" if such a failure matches that baseline failure. To identify whether a failure matches the baseline failure, we encode the logs using a pretrained sentence transformer model [37] and compute the cosine similarity between their embeddings. We consider two failures to match when their cosine similarity is at least 0.9, similar to prior work [27] that showed this approach to be effective for failure-log matching. We therefore evaluate whether an approach can reproduce one of the known failures for a given TD test using the available failure log.

[1]A Maven project may be divided into multiple modules, where each module contains its own set of tests that are run isolated from one another.

Table 1: Number of TD tests per module and characteristics of the tests.

ID	Project	Module	Tests	Number of Exec. Methods	Tokens	Test Suite Runtime (sec.)
M1	activiti/activiti	activiti-engine	3	2247.0	45653.3	21.5
M2	Alluxio/alluxio	.	2	811.3	24376.2	14.7
M3	apache/dubbo	dubbo-config/dubbo-config-api	3	975.7	31293.0	17.7
M4	apache/dubbo	dubbo-rpc/dubbo-rpc-dubbo	2	568.2	19873.5	15.5
M5	apache/httpcore	httpcore	2	29.0	1250.0	7.7
M6	apache/incubator-dubbo	dubbo-cluster	1	77.0	3340.0	13.9
M7	davidmoten/rxjava2-extras	.	1	84.0	1887.0	21.2
M8	doanduyhai/Achilles	integration-test	2	1198.0	34092.0	29.9
M9	doanduyhai/Achilles	integration-test-2_1	6	422.7	13888.7	40.2
M10	doanduyhai/Achilles	integration-test-2_2	6	398.5	13494.5	27.1
M11	doanduyhai/Achilles	integration-test-3_10	2	329.0	10328.0	25.6
M12	doanduyhai/Achilles	integration-test-3_7	2	355.0	11487.0	24.1
M13	feroult/yawp	yawp-testing/yawp-testing-appengine	1	204.5	3812.0	23.1
M14	flaxsearch/luwak	luwak	2	132.2	3133.0	7.0
M15	javadelight/delight-nashorn-sandbox	.	1	69.0	1776.0	7.0
M16	jknack/handlebars.java	handlebars-humanize	1	97.0	2874.0	10.3
M17	kagkarlsson/db-scheduler	.	1	107.0	2800.0	19.1
M18	qos-ch/logback	logback-classic	2	432.0	11509.5	13.2
M19	qos-ch/logback	logback-core	1	60.0	1327.0	12.3
M20	square/okhttp	okhttp-tests	9	267.4	8436.7	11.4
M21	TooTallNate/Java-WebSocket	.	26	186.0	8220.3	8.8
M22	undertow-io/undertow	core	2	319.0	11556.0	19.8
M23	vmware/admiral	adapter/registry	1	206.0	7524.0	15.3
M24	vmware/admiral	compute	1	781.0	32455.0	13.3
M25	zxing/zxing	core	2	69.5	2868.5	3.0
Sum x 1 | Average x 3			**82**	**417.0**	**12370.2**	**16.9**

Isolated Rerun, which repeatedly reruns each test in isolation, serves as an upper bound and detects all 82 TD-test failures in our dataset by design, albeit at a high cost (Section 5.3). Overall, TDRepro detects failures for 60 out of 82 TD tests, achieving a detection rate of 73.2%. In contrast, FlakeRake detects 36 TD-test failures, corresponding to a detection rate of 43.9%. Concerning reproduced failures, we find that TDRepro reproduces all 60 TD-test failures detected, meaning that all detected failures match the corresponding failures from Isolated Rerun. In contrast, FlakeRake reproduces only 30 failures.

Answer to RQ1. TDRepro substantially outperforms FlakeRake, detecting 60 failures (73.2%) versus 36 (43.9%), and reproducing 60 failures compared to 30, representing a 100% improvement.

5.2 RQ2: Reliability of Reproduced Failures

We evaluate how reliably each test failure can be reproduced by counting how often the same failure occurs across 100 runs, using the configurations outputted by TDRepro and FlakeRake. We specifically count how often the same failure is reproduced when run under the configuration within 100 runs. We also compare against the number of times the failure is reproduced in the first 100 runs of the 10,000 Isolated Rerun runs, ensuring we use the same number of runs for comparison purposes.

Table 3 shows a comparison of failure reproduction rates between TDRepro, FlakeRake, and Isolated Rerun. The rows correspond to the failure reproduction rates achieved by TDRepro, grouped into multiple percentage intervals (from 0% to (99%,100%]), and the columns are divided into the same percentage intervals for FlakeRake and Isolated Rerun. Each cell in the table indicates the number of tests that fall into a specific combination of reproduction-rate ranges between TDRepro and the other two approaches. A higher failure reproduction rate enables more efficient debugging and root-cause analysis.

The table shows that TDRepro achieves high reliability in reproducing TD-test failures, with near-to-deterministic reproduction rate (99%,100%] for 55 tests. Interestingly, FlakeRake similarly achieves near-to-deterministic reproduction rate for all 30 TD tests whose failures it can reproduce, though it reproduces much fewer TD-test failures than TDRepro. Meanwhile, although Isolated Rerun reproduces all failures by our evaluation design (note that the table shows 39 tests reproduced at rate 0% because we only consider the first 100 runs for comparison purposes), the failure reproduction rate per test is lower and no tests fail at the highest (99%,100%] rate.

Answer to RQ2. Across 100 reruns, TDRepro delivers more stable failure reproductions, with 55 tests failing at a >99% rate. FlakeRake reaches that level for all 30 tests it can reproduce, and Isolated Rerun cannot achieve that level for any tests.

5.3 RQ3: Cost to Reproduce Failures

We measure the cost of each approach to reproduce failures multiple times, measured in terms of both time and monetary cost. For TDRepro and FlakeRake, we apply the configuration outputted by

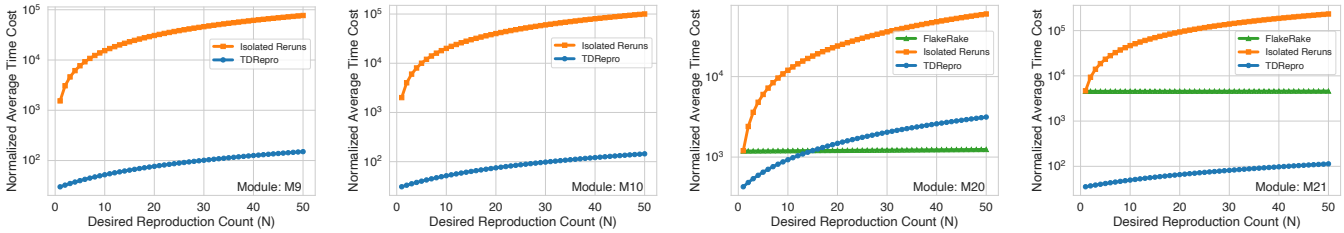

Figure 9: Average expected time (sec., log scale) across all tests per module required to observe N failures by the approaches per module. The time includes both the per-run normalized cost and, for TDRepro, the one-time configuration search cost.

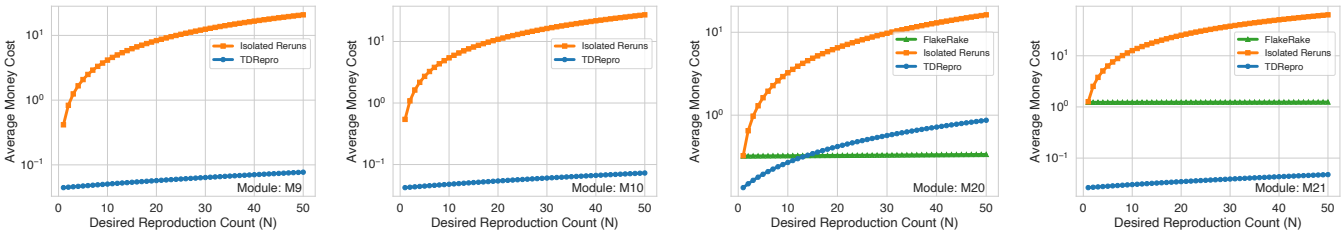

Figure 10: Average expected monetary cost (USD, log scale) across all tests per module required to observe N failures by the approaches per module. The cost includes both the per-run machine cost and, for TDRepro, the one-time LLM prompt cost.

Table 2: Test failures detected and reproduced by FlakeRake and TDRepro.

ID	# Tests Failure Detected		#Tests Failure Reproduced	
	FlakeRake	TDRepro	FlakeRake	TDRepro
M1	3	3	2	3
M2	2	0	2	0
M3	3	3	3	3
M4	2	1	2	1
M5	1	0	1	0
M6	0	0	0	0
M7	1	1	1	1
M8	0	0	0	0
M9	0	6	0	6
M10	0	6	0	6
M11	0	2	0	2
M12	0	2	0	2
M13	1	0	0	0
M14	1	2	0	2
M15	1	1	1	1
M16	1	0	1	0
M17	1	1	1	1
M18	2	0	2	0
M19	1	1	1	1
M20	9	4	7	4
M21	4	25	4	25
M22	2	2	2	2
M23	1	0	0	0
M24	0	0	0	0
M25	0	0	0	0
Total	**36**	**60**	**30**	**60**

each approach for a given TD test and run the test 100 times. For Isolated Rerun, we run the test 10,000 times. Using the observed failure rate, we estimate the expected number of reruns required to observe N failures using: $E[\text{runs for } N \text{ failures}] = N/p$, where p is the empirical failure probability.

For TDRepro, the total time cost is the sum of the time required to search for a failure-inducing configuration and the time required to run the test with that configuration enough times to observe N failures. The monetary cost for TDRepro accounts for both the cost of LLM API calls during configuration search (we estimate the cost using OpenAI pricing[2]) and the cost of compute resources for both searching and test execution (we estimate using AWS on-demand pricing[3] for the m8a.4xlarge instance type that closely matches the specification of the machine we use in our experiments). For FlakeRake, the total time cost is the sum of the time required to search for failure-inducing configurations and the time required to run the test with that configuration enough times to observe N failures, based on the empirical failure rate. The monetary cost for FlakeRake accounts for the cost of compute resources for both configuration search and test execution. For Isolated Rerun, there is no one-time configuration search or LLM cost. Therefore, the total time and monetary cost is simply the amount of time and money needed to run the test enough times to observe N failures, based on the observed empirical failure rate. To enable comparison across tests with varying runtimes, we normalize the reproduction cost for each failure by dividing by the time and money required to run the corresponding test once in isolation.

Figures 9 and 10 show the expected normalized time and monetary cost, respectively, to reproduce up to 50 failures using TDRepro (blue line), Isolated Rerun (orange line), and FlakeRake (green line). We show results for the four modules that contain more than five TD tests (Table 1). In each subplot, the X-axis represents the number of desired reproduced failures (N) and the Y-axis (log scale) shows the normalized cost (either time in seconds or money in USD) relative to the cost of a single isolated test run. As FlakeRake cannot reproduce a failure for the M9 and M10 modules, the corresponding subplots

[2]https://openai.com/pricing
[3]https://aws.amazon.com/ec2/pricing/on-demand

Table 3: Comparison of failure reproduction rates for TDRepro, FlakeRake, and Isolated Rerun.

	FlakeRake							Isolated Rerun							
TDRepro	**0%**	**(0%,10%]**	**(10%,25%]**	**(25%,50%]**	**(50%,75%]**	**(75%,99%]**	**(99%,100%]**	**0%**	**(0%,10%]**	**(10%,25%]**	**(25%,50%]**	**(50%,75%]**	**(75%,99%]**	**(99%,100%]**	**Total**
0%	12	0	0	0	0	0	10	9	12	1	0	0	0	0	22
(0%,10%]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(10%,25%]	0	0	0	0	0	0	3	3	0	0	0	0	0	0	3
(25%,50%]	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1
(50%,75%]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(75%,99%]	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1
(99%,100%]	39	0	0	0	0	0	16	27	5	1	0	4	18	0	55
Total	52	0	0	0	0	0	30	39	18	3	0	4	18	0	82

contain only two lines, representing TDRepro and Isolated Rerun. The results show that the cost of TDRepro and FlakeRake flattens as N increases, because they search for a configuration only once, while the cost of Isolated Rerun keeps growing with N. To observe 50 failures, both TDRepro and FlakeRake cost less than Isolated Rerun in both time and money. When comparing TDRepro against FlakeRake, we observe that TDRepro outperforms FlakeRake. On average, to reproduce a failure 50 times, TDRepro is 1482.1× faster and 888.4× cheaper than Isolated Rerun, and 10.4× faster and 9.6× cheaper than FlakeRake across all modules. TDRepro's machine cost averages $0.22 per test, compared with $2.55 for FlakeRake. Although FlakeRake does not incur any LLM cost, its machine cost is much higher than TDRepro. TDRepro adds a small one-time LLM search cost of $0.018 per test on average, for a total cost of $0.24 per test. The LLM cost accounts for 7.5% of TDRepro's total cost. FlakeRake performs better than TDRepro (1.5× faster and 1.6× cheaper than TDRepro to reproduce the same failures for 50 times) only for module M20 as shown in Figures 9 and 10. The reason is that, for module M20, TDRepro has many unsuccessful attempts, for which configuration search requires more time. Results for the remaining modules are available on our website [7].

Answer to RQ3. TDRepro takes substantially less time and costs less money to use than Isolated Rerun, with an average of 1482.1× less time and 888.4× less money. Compared to FlakeRake, TDRepro, on average, takes 10.4× less time and costs 9.6× less money.

5.4 RQ4: Component Analysis of TDRepro

We show how our design choices behind each component of TDRepro contribute to its overall performance. We begin by examining four different ablations that remove key capabilities: (i) Method Ranking, where we evaluate the importance of TDRepro's method-ranking strategy by comparing it with an LLM-only approach that does not perform method ranking and instead provides all executed methods directly to the LLM to rank delay locations (Section 3.3); (ii) Embedding Model Comparison, where we compare GPT-2-Medium embedding with other popular embedding models for ranking candidate methods and evaluate their effectiveness in reproducing failures and their runtime; (iii) Reasoning Model Comparison, where we compare GPT-5.6 with popular open-source reasoning models for reproducing failures using the same ranked methods; and (iv) Hyperparameter Sensitivity, where we evaluate the impact of varying the number of candidate methods (K) and the number of GPT-suggested delay locations (L).

Method Ranking. Table 4 shows the effectiveness of method ranking in reproducing TD-test failures. The table shows the LLM-only baseline reproduces 9 test failures (avg. 345.9 seconds), whereas the embedding-guided method ranking (i.e., TDRepro's method ranking using GPT-2-Medium embedding) reproduces 60 test failures (avg. 13335.0 seconds) across 25 modules. This result indicates that ranking methods using embedding similarities substantially improves failure reproduction. Furthermore, when we sequentially introduce delays before each line, we identify the correct location for 60 tests. These results highlight the effectiveness of our ranking in identifying the most relevant methods, including cases where the LLM-only baseline fails.

Table 4: Effectiveness of method ranking for reproducing test failures.

	LLM-only		Embedding-only	
ID	**Reproduced**	**Runtime**	**Reproduced**	**Runtime**
M1	0	184.0	3	320.4
M2	0	131.8	0	1032.5
M3	0	0.0	3	323.9
M4	0	132.6	1	36067.6
M5	0	76.2	0	1375.3
M6	0	155.5	0	65751.9
M7	1	56.0	1	237.2
M8	0	165.9	0	2484.4
M9	0	246.2	6	2276.3
M10	0	254.3	6	1608.6
M11	0	34.0	2	611.2
M12	0	275.0	2	2524.4
M13	0	0.0	0	3332.5
M14	1	85.3	2	390.1
M15	1	71.1	1	146.5
M16	0	0.0	0	1115.0
M17	0	114.1	1	686.2
M18	0	224.7	0	14200.0
M19	1	96.2	1	80.2
M20	1	92.8	4	8396.7
M21	4	27.4	25	377.5
M22	0	240.6	2	815.9
M23	0	318.4	0	2975.7
M24	0	0.0	0	4671.8
M25	0	0.0	0	12373.6
Total/	9		60	
Avg.		345.9		13335.0

Table 5: Experiments with different embedding choices and open-source inference models. Runtime represents the percentage difference in runtime relative to TDRepro. A positive value indicates that the technique is that percentage slower than TDRepro, whereas a negative value indicates that it is faster.

	Ranking Quality (Embedding Model)				Replacing GPT-5.6 with Open-source Models			
	Qwen/Qwen2-7B embedding		meta-llama/Meta-Llama-3-8B embedding		Qwen/Qwen2.5-Coder-14B-Instruct		meta-llama/Llama-3.3-70B-Instruct	
ID	Reproduced	Runtime	Reproduced	Runtime	Reproduced	Runtime	Reproduced	Runtime
M1	2	227.9	2	425.0	0	332.2	0	3740.7
M2	2	-41.5	1	-34.7	0	0.0	0	0.0
M3	3	108.9	3	123.8	3	589.8	2	3055.1
M4	1	-87.9	1	-87.5	1	8.3	0	768.1
M5	0	6.9	0	-1.9	0	0.0	0	0.0
M6	0	-14.1	0	-42.2	0	0.0	0	0.0
M7	1	-8.2	1	29.1	1	33.8	1	1110.1
M8	0	0.0	0	0.0	0	0.0	0	0.0
M9	6	152.6	6	104.9	3	1053.7	5	1206.8
M10	6	437.9	6	274.5	5	427.6	5	1205.0
M11	2	2857.6	2	270.1	0	1109.0	2	1214.6
M12	2	40.4	2	18.1	1	403.4	2	596.4
M13	0	38.2	0	-20.5	0	0.0	0	0.0
M14	2	-8.2	2	2.8	2	29.5	2	2029.4
M15	1	22.2	1	-34.0	1	119.4	1	1643.3
M16	0	22.6	0	-25.1	0	0.0	0	0.0
M17	1	80.2	1	-11.9	1	20.6	1	1588.3
M18	0	8.4	0	81.4	0	0.0	0	0.0
M19	1	2.9	1	44.7	1	32.3	1	1146.5
M20	2	59.9	2	-41.9	2	664.2	1	303.5
M21	23	-0.6	24	29.2	24	49.8	21	2030.5
M22	1	-3.5	1	17.8	2	54.7	1	1247.3
M23	0	-18.7	0	155.4	0	0.0	0	0.0
M24	0	33.6	0	1.8	0	0.0	0	0.0
M25	0	0.0	0	0.0	0	0.0	0	0.0
Total/ Avg.	56	431.2	56	197.4	47	893.7	45	4303.3

Embedding Model Comparison. We replace GPT-2-Medium embedding with Qwen/Qwen2-7B embedding and meta-llama/Meta-Llama-3-8B embedding to rank executed methods. From each model, we select the top-10 methods and inject delays line-by-line (in ranked order) to determine how many lines must be traversed before the first test failure occurs. Table 5 shows that both Qwen/Qwen2-7B embedding and meta-llama/Meta-Llama-3-8B embedding reproduce 56 failures, requiring 431.2 seconds and 197.4 seconds, respectively. Although both Qwen/Qwen2-7B embedding and meta-llama/Meta-Llama-3-8B embedding achieve comparable effectiveness to GPT-2-Medium embedding in terms of reproduced failures, they require substantially more runtime than GPT-2-Medium embedding, by 431.2% and 197.4%, respectively.

Reasoning Model Comparison. We evaluate the choice of reasoning model GPT-5.6 by swapping it with open-source LLMs, namely, meta-llama/Llama-3.3-70B-Instruct and Qwen/Qwen2.5-Coder-14B-Instruct. As shown on the right-side of Table 5, Qwen reproduces 47 failures in 893.7 seconds, while Llama reproduces 45 failures in 4303.3 seconds. However, when compared with our GPT-5.6 configuration (Table 2), both open-source alternatives reproduce fewer failures. In summary, large open-source models can serve as faster, usable substitutes when budgets dominate, but GPT-5.6 remains preferable when maximizing failure reproduction.

Hyperparameter Sensitivity. Figure 11 presents the success rate versus average runtime for different values of K and L used to achieve that success. For both parameters, increasing the search space initially improves reproduction effectiveness, but the benefit peaks at 10 and then declines. With K, the success rate peaks at 10 candidate methods; considering 15 methods increases execution time by approximately 5% while slightly reducing effectiveness. Likewise, increasing the number of GPT-suggested delay locations beyond 10 incurs additional runtime without improving reproduction performance. Conversely, using only 3 or 5 candidates reduces runtime but noticeably lowers the success rate because relevant delay locations are more likely to be excluded from the search space. Consequently, $K = 10$ and $L = 10$ lie on the best effectiveness and efficiency trade-off, achieving the highest reproduction rate while also costing less time than the larger $K = 15$ and $L = 15$ configurations. We therefore use $K = L = 10$ throughout the paper.

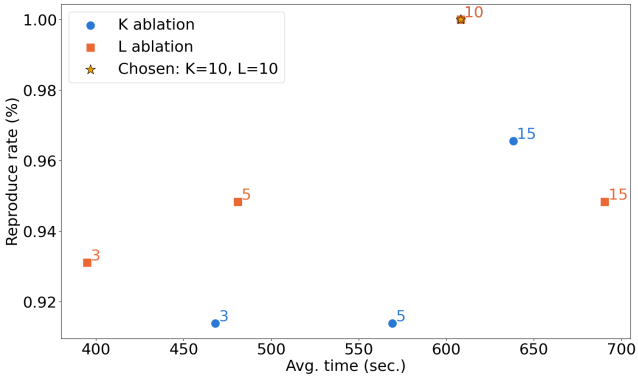

Figure 11: Comparison between different values of K and L.

Answer to RQ4. Ranking executed methods before passing them to the LLM achieves better reproduction effectiveness than directly passing all executed methods. Among the embedding models evaluated for method ranking, the model used by TDRepro more effectively ranks the true method (i.e., the method containing the actual delay location) than the other open-source embedding models. Similarly, GPT-5.6, which TDRepro uses to identify delay locations, reproduces more failures than the evaluated open-source LLMs. Finally, using $K = L = 10$ achieves the best balance between reproduction effectiveness and runtime.

6 Threats to Validity

Our study focuses on open-source Java projects, which may limit the generalizability of our findings to other programming languages and projects. While the high-level algorithm behind TDRepro is language-agnostic, differences in syntax and testing frameworks (e.g., pytest [5]) may affect its effectiveness.

Certain design choices may affect our results. For instance, we fix the feedback count to five, though higher values may improve performance. TDRepro currently injects a single five second delay at one location, similar to prior work [35]. Some failures may require injecting multiple delays or having different amounts of delays.

TDRepro restricts delay injection to the project code under test. It may be possible to reproduce more TD-test failures by expanding the search to consider locations outside the project code, e.g., in library code, so our current results may still be underestimating the effectiveness of injecting delays to reproduce failures.

7 Related Work

Luo et al. performed the first empirical study on flaky tests in open-source projects [26]. They categorized flaky tests by manually inspecting developer-fixed flaky tests. They found that the most prevalent types of flaky tests are asynchronous wait and concurrency flaky tests, where both types similarly fail due to differences in timing when a test executes multiple threads concurrently. Rahman et al. termed both types of flaky tests together as timing-dependent (TD) tests [35]. Researchers would develop approaches to automatically detect different types of flaky tests based on the results from Luo et al.'s study, including rerun, dynamic analysis, static analysis, and machine-learning approaches [10, 11, 13, 20–23, 30, 31, 38–42, 44].

Rahman et al. developed an approach, FlakeRake [35], to reproduce the failures from TD tests. FlakeRake targets a set of known Java APIs related to multithreading by injecting delays at locations that use these APIs during test execution to see whether the test fails more reliably. Similarly, Leesatapornwongsa et al. proposed FlakeRepro [24], which uses static and dynamic program analysis to determine where to inject delays to reproduce specific thread interleavings that can lead to previously observed failures. TDRepro is similar to both prior work in that we also aim to inject delays to induce TD-test failures. The difference is that TDRepro relies on code similarity and LLMs to rank likely delay locations, with the goal to more effectively and efficiently determine the correct delay locations that lead to the failures. We directly compare against FlakeRake given our focus on Java projects, finding that TDRepro can reproduce TD-test failures better than FlakeRake.

Rahman et al. investigated repairing TD tests with FlakeSync [36], in contrast to approaches that focus on reproducing TD-test failures. FlakeSync identifies points during test execution that need to be synchronized with one another to force out the expected behavior, preventing the test from flakily failing. Jayanthi et al. improved upon the original implementation of FlakeSync [19], outputting patches that can be applied to the project code to repair the TD test. Our work is complementary. By finding more efficient ways to reproduce TD-test failures, TDRepro can guide repair strategies towards generating the patch as well as validating patch correctness.

Recently, researchers have increasingly explored the use of LLMs to address various challenges related to flaky tests. Fatima et al. proposed Flakify [15], which fine-tunes CodeBERT [16], a pre-trained language model, to predict whether a test is flaky or not using only its source code. Flakify does not require repeated test executions or access to the production code. Akil et al. proposed FlakyCat [8], which uses CodeBERT and few-shot learning to classify flaky tests according to their root causes, such as classifying a flaky test as a TD test. Rahman et al. proposed FlakyQ [33] to further explore LLM-based flaky-test classification, making the predictions faster through quantization. Rahman et al. also proposed FlakyLens [34] to improve flaky-test classification by addressing limitations in prior evaluations, including data leakage and class imbalance. Chen and Jabbarvand proposed FlakyDoctor [12], which uses a neurosymbolic approach that combines LLMs with program analysis to automatically repair different types of flaky tests. Li et al. proposed FlakyGuard [25], which extends upon this direction of leveraging LLMs to repair flaky tests, but in an industrial setting and using selective graph exploration to provide LLMs with relevant code context for flaky-test repair when there is a large codebase to analyze. Although these approaches demonstrate the potential of LLMs for addressing problems with flaky tests, they primarily focus on detecting, classifying, or repairing flaky tests. In contrast, TDRepro focuses on reproducing flaky-test failures, specifically on TD tests, by using LLMs and code similarity to rank candidate locations to inject delays and more easily reproduce failures. Essentially, techniques to reproduce flaky-test failures are complementary to these prior flaky-test approaches.

8 Conclusions

We present TDRepro, a neurosymbolic approach for detecting and reproducing TD-test failures via injecting delays at strategic locations to more easily induce the failures. TDRepro reduces the search space by first identifying methods executed by the TD test, ranking them using embedding-based similarity, and then leveraging an LLM to infer candidate delay locations. The approach validates these candidates through targeted delay injections and iteratively refines the search using feedback from unsuccessful attempts.

TDRepro reproduced 60 out of 82 test failures, while FlakeRake reproduced 30 failures. TDRepro also demonstrated high reliability, achieving a >99% failure reproduction rate for 55 tests. In comparison, only 30 tests reach this rate of reproduction with FlakeRake, and none with Isolated Rerun. TDRepro also required substantially less time and cost to reproduce failures than FlakeRake and Isolated Rerun. Such a high reproduction rate combined with lower reproduction cost can substantially improve developers' ability to debug flaky tests.

Data Availability

Our artifact is at https://doi.org/10.5281/zenodo.19342874.

Acknowledgments

We thank the anonymous reviewers for their comments and feedback. This work is partially supported by the US National Science Foundation grant numbers CCF-2145774 and CCF-2338287, as well as the Jarmon Innovation Fund.

References

[1] 2026. GPT-2. https://huggingface.co/openai-community/gpt2.
[2] 2026. IDoFT. https://github.com/TestingResearchIllinois/idoft.
[3] 2026. Introducing OpenAI. https://openai.com.
[4] 2026. JaCoCo Java Code Coverage Library. https://github.com/jacoco/jacoco.
[5] 2026. pytest. https://pypi.org/project/pytest.
[6] 2026. qos-ch/logback. https://github.com/qos-ch/logback.
[7] 2026. TDRepro. https://sites.google.com/view/tdrepro.
[8] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. 2023. FlakyCat: Predicting Flaky Tests Categories using Few-Shot Learning. In *International Conference on Automation of Software Test*. https://doi.org/10.1109/AST58925.2023.00018
[9] Abdulaziz Alaboudi and Thomas D. LaToza. 2023. What Constitutes Debugging? An Exploratory Study of Debugging Episodes. *Empirical Software Engineering* 28, 5 (2023). https://doi.org/10.1007/s10664-023-10352-5
[10] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *International Conference on Software Engineering*. 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 *International Conference on Software Engineering*. https://doi.org/10.1145/3180155.3180164
[12] Yang Chen and Reyhaneh Jabbarvand. 2024. Neurosymbolic Repair of Test Flakiness. In *International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3650212.3680369
[13] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Misailovic Sasa. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3395363.3397366
[14] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3338906.3338945
[15] Sakina Fatima, Taher A. Ghaleb, and Lionel Briand. 2023. Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests. *IEEE Transactions on Software Engineering* 49, 4 (2023). https://doi.org/10.1109/TSE.2022.3201209
[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Empirical Methods in Natural Language Processing*. https://doi.org/10.18653/v1/2020.findings-emnlp.139
[17] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3106237.3106270
[18] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *International Conference on Automated Software Engineering*. https://doi.org/10.1145/2970276.2970358
[19] Nandita Jayanthi, Shanto Rahman, and August Shi. 2026. FlakeSync: A Tool for Automatically Repairing Async Flaky Tests. In *International Conference on Software Engineering (Tool Demonstrations Track)*.
[20] Tariq M. King, Dionny Santiago, Justin Phillips, and Peter J. Clarke. 2018. Towards a Bayesian Network Model for Predicting Flaky Automated Tests. In *International Conference on Software Quality, Reliability and Security Companion*. https://doi.org/10.1109/QRS-C.2018.00031
[21] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and Ranking Flaky Tests at Apple. In *International Conference on Software Engineering (Software Engineering in Practice Track)*. https://doi.org/10.1145/3377813.3381370
[22] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and partially Classifying Flaky Tests. In *International Conference on Software Testing, Verification, and Validation*. https://doi.org/10.1109/ICST.2019.00038
[23] 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 *International Symposium on Software Reliability Engineering*. https://doi.org/10.1109/ISSRE5003.2020.00045
[24] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. 2022. FlakeRepro: Automated and Efficient Reproduction of Concurrency-related Flaky Tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3540250.3558956
[25] Chengpeng Li, Farnaz Behrang, August Shi, and Peng Liu. 2025. FlakyGuard: Automatically Fixing Flaky Tests at Industry Scale. In *International Conference on Automated Software Engineering*. https://doi.org/10.1109/ASE63991.2025.00179
[26] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/2635868.2635920
[27] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. 2023. Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation. In *International Conference on Software Engineering*. https://doi.org/10.1109/ICSE48619.2023.00063
[28] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *International Conference on Software Engineering (Software Engineering in Practice Track)*. https://doi.org/10.1109/ICSE-SEIP.2017.16
[29] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding Bugs by Isolating Unit Tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/2025113.2025202
[30] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael D. Ernst. 2021. Verifying Determinism in Sequential Programs. In *International Conference on Software Engineering*. https://doi.org/10.1109/ICSE43902.2021.00017
[31] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the Vocabulary of Flaky Tests?. In *Mining Software Repositories*. https://doi.org/10.1145/3379597.3387482
[32] Michael Pradel and Thomas R Gross. 2012. Fully Automatic and Precise Detection of Thread Safety Violations. In *Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/2254064.2254126
[33] Shanto Rahman, Abdelrahman Baz, Sasa Misailovic, and August Shi. 2024. Quantizing Large-language Models for Predicting Flaky Tests. In *International Conference on Software Testing, Verification, and Validation*. https://doi.org/10.1109/ICST60714.2024.00018
[34] Shanto Rahman, Saikat Dutta, and August Shi. 2025. Understanding and Improving Flaky Test Classification. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025). https://doi.org/10.1145/3763098
[35] Shanto Rahman, Aaron Massey, Wing Lam, August Shi, and Jonathan Bell. 2024. Automatically Reproducing Timing-Dependent Flaky-Test Failures. In *International Conference on Software Testing, Verification, and Validation*. https://doi.org/10.1109/ICST60714.2024.00032
[36] Shanto Rahman and August Shi. 2024. FlakeSync: Automatically Repairing Async Flaky Tests. In *International Conference on Software Engineering*. https://doi.org/10.1145/3597503.3639115
[37] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*. https://doi.org/10.18653/v1/D19-1410
[38] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *International Conference on Software Testing, Verification, and Validation*. https://doi.org/10.1109/ICST.2016.40
[39] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. 2020. Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In *International Conference on Software Maintenance and Evolution*. https://doi.org/10.1109/ICSME46990.2020.00037
[40] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. 2021. Know You Neighbor: Fast Static Prediction of Test Flakiness. *IEEE Access* 9 (2021). https://doi.org/10.1109/ACCESS.2021.3082424
[41] Ruixin Wang, Yang Chen, and Wing Lam. 2022. iPFlakies: A Framework for Detecting and Fixing Python Order-dependent Flaky Tests. In *International Conference on Software Engineering (Tool Demonstrations Track)*. https://doi.org/10.1145/3510454.3516846
[42] 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*. https://doi.org/10.1145/3510003.3510170
[43] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012). https://doi.org/10.1002/stvr.430
[44] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/2610384.2610404