2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

FlakyGuard: Automatically Fixing Flaky Tests at Industry Scale

Chengpeng Li*, Farnaz Behrang†, August Shi*, Peng Liu†
*ECE Department, The University of Texas at Austin, USA {chengpengli, august}@utexas.edu
†Uber Technologies, USA {behrang, peng3141}@uber.com

***Abstract*—Flaky tests that non-deterministically pass or fail waste developer time and slow release cycles. While large language models (LLMs) show promise for automatically repairing flaky tests, existing approaches like FlakyDoctor fail in industrial settings due to the *context problem*: providing either too little context (missing critical production code) or too much context (overwhelming the LLM with irrelevant information). We present FLAKYGUARD, which addresses this problem by treating code as a graph structure and using selective graph exploration to find only the most relevant context. Evaluation on real-world flaky tests from industrial repositories shows that FLAKYGUARD repairs 47.6% of reproducible flaky tests with 51.8% of the fixes accepted by developers. Besides it outperforms state-of-the-art approaches by at least 22% in repair success rate. Developer surveys confirm that 100% find FLAKYGUARD's root cause explanations useful.**

I. INTRODUCTION

Flaky tests non-deterministically pass or fail when rerun on identical code, creating significant challenges in software development. When tests fail, developers cannot distinguish between those failures indicating real bugs or flakiness, forcing them to spend time investigating false alarms. In many industrial settings, teams may block code deployment until all test failures are resolved, slowing down release cycles. The prevalence of flaky tests wastes both developer time and computational resources [1, 2].

Recent advances in large language models (LLMs) [3, 4, 5], particularly in code generation and reasoning capabilities, show great promise for automatically repairing flaky tests. Prior work has demonstrated this potential: FlakyDoctor [6] successfully repaired more flaky tests than traditional non-LLM approaches [7, 8, 9] in open-source projects. Importantly, non-LLM approaches require prior tools to classify flaky test types before applying pattern-specific fixes, limiting their applicability when such classification is unavailable, as is often the case in industrial settings where only test failure information is readily accessible.

However, our experiments with FlakyDoctor in an industrial setting at Uber revealed significant limitations, as it frequently failed to repair real-world flaky tests. Our analysis identified the primary cause as the *context problem*, which manifests in two ways. On one hand, the LLM may receive *too little* context: FlakyDoctor [6] provides only the test code as context, causing the LLM to miss critical production code information necessary for proper reasoning and root cause analysis. On the other hand, the LLM may receive *too much* context: including all production code from a service (e.g., medium-sized services at Uber typically exceed 100K+ LOCs) overwhelms the LLM, diffusing its attention [10], and degrading performance on complex reasoning tasks such as root cause analysis. Additionally, developers usually need clear explanations of root causes and repair rationale to accept proposed fixes. Therefore, the goal is to provide the right balance of context that is neither too much nor too little, enabling the LLM to not only repair the flaky test but also explain the root cause to developers.

Recent LLM-based approaches [11, 12] designed for other software engineering tasks have proposed context collection methods that can be adapted to flaky test repair with slight modifications (Section IV). While directly applying these context collection techniques to FlakyDoctor [6] improves repair rates, significant limitations remain in our domain. Specifically, Agentless [11] and AutoCodeRover [12] construct context by treating the codebase as a flat text corpus, missing opportunities to leverage code structure for more targeted context selection.

Treating the codebase as a graph structure enables more effective context collection by using the call graph among functions to guide the search, entirely avoiding irrelevant functions through structured graph traversal. However, analyzing the codebase as a graph requires a novel *graph search* design, as naive BFS/DFS approaches would include the entire graph, overwhelming the LLM. Existing graph-based work RepoGraph [13] uses ego-graphs [14] that include k-hop nodes around central nodes to avoid exponentially growing context, but this approach is severely limited by depth constraints. Our preliminary study revealed that critical repair information often resides in leaf nodes deeply embedded in the call graph. To address these limitations, we develop a *selective graph exploration* strategy that identifies and traverses only the most relevant paths in the call graph. Rather than being constrained by fixed depth limits, our approach uses the LLM itself to guide the graph exploration process, enabling it to reach critical information regardless of its depth while maintaining manageable context size.

To further boost the effectiveness of the graph search, we also use a dynamic call graph (DCG) collected from test runs. A DCG captures the precise scope of functions actually executed at runtime, enabling our search to follow only the executed paths. In contrast, static call graphs may include functions that are never called by the specific test case. In addition, static call graphs struggle with overloaded

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00179

functions, anonymous functions, and reflection-based calls, creating imprecise linkages that can misguide the LLM to either include irrelevant functions or miss relevant ones. The use of a DCG resolves these issues.

We develop FLAKYGUARD, which addresses the context problem through LLM-guided exploration of dynamic call graphs. Our approach constructs dynamic call graphs from test execution traces, then employs an iterative LLM-based selection process to expand only those paths deemed relevant for understanding flaky behavior. This selective traversal enables FLAKYGUARD to reach critical information at arbitrary depths while maintaining manageable context sizes for effective repair and clear root cause explanations.

Additionally, Go codebases in industrial settings predominantly use table-driven testing [15], where test functions contain multiple similar test cases organized in tables. This paradigm creates challenges for existing approaches [6, 11, 12, 13], which frequently analyze the wrong test cases or encounter patching failures due to ambiguity in search-and-replacement operations [11]. We develop AST-based *test simplification* and *patch transplantation* techniques (Section III-D) to address these challenges and apply them to all baseline approaches to ensure they produce meaningful results.

We evaluate FLAKYGUARD on real-world flaky tests of various types from industrial repositories at Uber. Over six months of deployment, FLAKYGUARD produced fixes for 47.6% of reproducible flaky tests, with 51.8% of generated fixes accepted by developers. Compared to state-of-the-art LLM-based repair methods [6, 11, 12, 13], FLAKYGUARD outperforms them by at least 22% in repair success rate. In anonymous developer survey responses, 100% of the developers found FLAKYGUARD's root cause explanations useful. We also conduct comprehensive ablation studies, case studies, and breakdown analyses to address key research questions.

Beyond the experimental validation, FLAKYGUARD has been deployed as a fully autonomous system at Uber, integrating with the existing ticketing system to automatically process flaky test reports and deliver fixes to developers daily.

In this paper, we make the following contributions:

- We identify and characterize the *context problem* in LLM-based flaky test repair, where existing approaches provide either too little or too much context.
- We develop FLAKYGUARD, a novel approach that uses LLM-guided exploration of dynamic call graphs to reach critical information at arbitrary depths while maintaining manageable context sizes.
- We conduct a large-scale evaluation in industrial settings, showing that FLAKYGUARD outperforms state-of-the-art approaches and produces useful root cause explanations.

II. RUNNING EXAMPLE

We illustrate the challenge of repairing flaky tests using the example in Listing 1. The test in the source file backend_test.go non-deterministically fails with the error "Not all calls expected by the mock for UpdateInfo were made". This error originates from the mock library, which verifies that all expected mock calls have been executed when the test completes, but it is not a standard assertion failure. Notably, this error provides no stacktrace, and even when stacktraces are available, they typically only show what happened rather than why the error occurred.

```
1  // backend_test.go
2  ...
3    t.Run("validation and call UpdateInfo", func(t *testing.T) {
4      ...
5  +   var wg sync.WaitGroup
6  +   wg.Add(1)
7  -   doc.Let().UpdateInfo().Return(nil)
8  +   doc.Let().UpdateInfo().Run(func(ctx context.Context, data *entity.Info) {
9  +       defer wg.Done()
10 +   }).Return(nil)
11 
12     ...
13     err := h.AddProgram(context.Background(), r)
14     ...
15 +   wg.Wait()
16     assert.NoError(t, err)
17   })
18 
19 // deep call chain leading to this is omitted.
20 // validator.go
21 func ValidateIdentity(ctx context.Context, program *entity.Program, c *controller) map[string]string {
22   ...
23   data, err := c.db.GetInfo(ctx, _a, email)
24   go c.db.UpdateInfo(ctx, data)
25   ...
26 }
27 ...
```

Listing 1: Flaky Test that needs the Deep Calling Context

The function `UpdateInfo` is invoked through a deep call chain: `AddProgram` (backend.go) → `AddProgram` (controller.go) → `ValidateIdentity` (validator.go) → `UpdateInfo`, with each function residing in a different source file. Critically, `UpdateInfo` is invoked as a goroutine without synchronization with the test execution. The goroutine may not complete when the test finishes, especially in the heavily loaded CI environments, thereby causing the mock call to be missed and triggering the error.

The fix produced by FLAKYGUARD adds a `WaitGroup` to synchronize the test with the goroutine: the test waits (line 15) until the goroutine signals completion (line 9) after making the mock call (line 24). Discovering this fix automatically is challenging because root cause analysis requires the LLM to trace through the call chain across multiple files to identify that `UpdateInfo` runs in an unsynchronized goroutine. This scenario exemplifies the fundamental *context problem*: without sufficient context, the LLM fails to understand why the test is flaky, yet too much context overwhelms the LLM with irrelevant information. We examine how different context collection strategies perform on this example.

No Production Code Without production code context, the LLM initially tries to remove the mock entirely, which fails to pass the validations, then settles on relaxing the mock expectation with `.MaxTimes(1)`. While this eliminates flakiness, it weakens the test by allowing the mock call to be skipped, which is precisely what the test should detect as a bug.

Text Search–based Approaches We tested Agentless [11] and AutoCodeRover [12]. Agentless uses hierarchical search (files by names, then functions by signatures, then

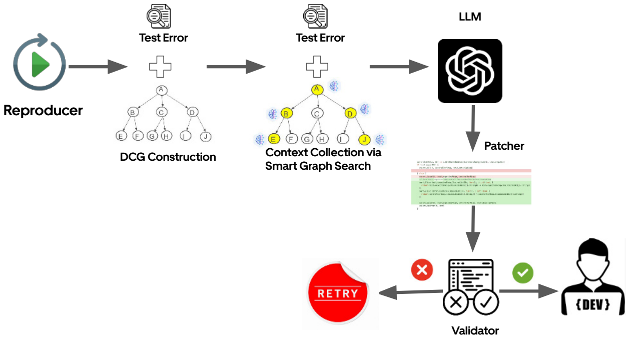

Fig. 1: Overview of the FLAKYGUARD Workflow.

code blocks), while AutoCodeRover provides APIs like `search_method_in_file` and lets the LLM decide which APIs to call.

Both failed to repair the flaky test, collecting irrelevant context while missing the critical `ValidateIdentity` function. These agents rely on natural language understanding to guide search, but `ValidateIdentity` provides no hint that it calls `UpdateInfo`. With 35+ functions per file and nearly 50 calls in `AddProgram` alone—including five functions with the "Validate" prefix—the noise overwhelms context selection.

Graph based Approaches We attempted using RepoGraph [13, 16], which extends Agentless [11] by building static call graphs and retrieving k-hop ego-graphs around relevant functions. RepoGraph keeps k small (1-2 hops) to avoid exponential context growth. However, it fails to help in this example, because RepoGraph avoids deep traversal to leaf nodes such as `ValidateIdentity`, precisely where the root cause resides. Moreover, its static call graph includes unexecuted branches, adding noise that confuses the LLM.

Our Work FLAKYGUARD addresses these limitations using LLM-guided selective graph search on top of dynamic call graphs. First, the dynamic call graph contains only 40% of the nodes compared to static call graphs by pruning unexecuted branches and precisely resolving overloaded functions. This filtering bounds the search space to actually executed function chains, reducing noise. Second, our selective graph search can traverse deeply to leaf nodes like `ValidateIdentity`, enabling the LLM to generate not only correct fixes but also reasonable root cause explanations.

III. FLAKYGUARD METHODOLOGY

Figure 1 shows the high-level overview. Given a flaky test reported by our ticketing system, FLAKYGUARD reproduces flaky test failures, builds a dynamic call graph (Section III-A), gathers context via LLM-guided graph traversal (Section III-B), and uses prompts to request the fixes from the LLM. After receiving a suggested fix, FLAKYGUARD applies and validates it, and automatically submits it for developer review if validation succeeds. Otherwise, it enters the iterative fixing loop (Section III-C). Lastly, Section III-D describes FLAKYGUARD's test simplification and patch transplantation techniques, which are generally applicable to existing approaches [11, 12, 13] for addressing key challenges in table-driven testing.

Before we dive into each component, we first introduce the inputs of FLAKYGUARD and how it reproduces the flaky test failures to collect the error details. These steps are crucial, because the error details are not always available in the ticketing system, or they may be stale due to recent code changes.

Inputs of FLAKYGUARD FLAKYGUARD receives the flaky tests reported by our ticketing system (out of scope), e.g., in the conventional *Bazel* form of test_target/test_func/test_case, where `test_target` consists of multiple test functions, and each `test_func` contains multiple test cases following the predominantly used table-driven testing pattern [15].

Failure reproduction. To collect failure information, we execute the test case N times (default: 1000). If no failures occur, indicating potential interference from other test cases, we execute the entire test target N times again and filter results for the targeted test case. The workflow terminates if reproduction fails.

Information extraction. From reproduced failures, we extract: error message, stack trace, assertion file path and line number, and test function file path. These paths may differ when tests invoke external assertion utilities. We also extract the assertion statement using the file path and line number. We use regular expressions to parse this information, using LLM fallback for rare corner-cases.

Prompting FLAKYGUARD uses a general prompt design not customized for specific flakiness types. The prompt combines a system message, extracted failure information, and context from graph traversal (Sections III-A and III-B). In the system prompt, we use "you are an expert in fixing flaky tests", and we specify that the LLM should only make changes to the test code, not the production code, which is beyond the scope of FLAKYGUARD.

A. Construction of Dynamic Call Graphs

To construct the dynamic call graph, FLAKYGUARD instruments the source code at compile time to inject the logging code [1], which collects call edges at runtime.

The instrumentation modifies the original source code. To ensure the call graph accurately reflects execution, we collect the call graphs specific to the failed runs. The instrumentation is removed before requesting fixes from the LLM, with fixes later transplanted back to the original source (Section III-D).

At runtime, the logging code collects call edges in the format `MethodEntry: file, line, method Caller: file, line, method`. It also dedups to log only the new edges, and it flushes the file upon every new edge to ensure all edges are written before test completion. With the collected logs, FLAKYGUARD builds a dynamic call graph by mapping each method to a unique AST node using the file path and line number, with each file mapped to a unique AST tree.

[1] We follow techniques from bazel/rules_go [17].

Encoding Function Calls via Goroutines Consider this minimal example of commonly used goroutine calls: `func A(){go B()}`. The call edge from `A` to `B` is not captured in the log because `B` is launched in a new goroutine, and the runtime stack does not include the enclosing function `A`.

We address this issue by analyzing the AST of the function `A` that launches the goroutine, identifying the functions being called, e.g., `B`. We then link the nodes of `A` and `B`.

This name-based linkage is not perfect, as it may introduce spurious edges due to interface call resolution ambiguity. Also, if goroutines originate from external libraries without source code, the call edge is lost, and those functions like `B` become root nodes in the graph traversal. We found those issues do not impact FLAKYGUARD much in practice, as only around 0.2% of code starts goroutines from external libraries.

B. Context Collection with LLM-guided Graph Traversal

Algorithm 1 performs a breadth-first search (BFS) where a LLM selects which nodes to explore rather than exploring all neighbors. Starting from root nodes, the algorithm queries the LLM at each step to select at most k children per node based on their relevance to the flaky test error, continuing to depth d (configurable, set to infinite by default). This step produces a list of selected nodes in BFS order.

As a final post-processing step, the LLM selects up to F most relevant functions from this result to reduce context size, in addition to the root nodes that are always included. Unlike existing approaches that use depth-limited search [13], this method is more effective since it identifies globally the relevant functions rather than imposing arbitrary depth constraints.

Algorithm 1: Smart Graph Traversal with GenAI

```
Input: Call graph $G$, depth limit $d$, children per node $k$
Output: List of selected nodes $L$
Function CollectContext(G, d, k):
    $L \leftarrow$ GetRootNodes(G);
    $Q \leftarrow$ queue of tuples $(n, 0)$ for nodes $n \in L$ ;
     // each tuple: (node, depth)
    while $Q$ is not empty do
        $(n, h) \leftarrow Q$.pop();
        if $h \geq d$ then
            continue;
        $C \leftarrow$ GetChildren($n$);
        $S \leftarrow$ GenAI_Select($C, k$);
        foreach $s \in S$ do
            $L \leftarrow L \cup \{s\}$;
            $Q$.append($s, h + 1$);
    return $L$;
```

C. Fixing Loops

At a high level, FLAKYGUARD operates through three nested loops: the outer loop (M iterations) collects new contexts, the middle loop (P iterations) generates high-level thoughts by prompting the LLM, and the inner loop (N iterations) produces, applies, and validates fixes for each thought. Each loop uses progressively enriched prompts: the outer loop uses the flakiness information, the middle loop adds context from graph traversal, and the inner loop incorporates the proposed thought. We explain the configurable parameters in Section IV-A.

In the middle loop, each thought contains root cause category, root cause explanation, and fixing plan. Following prior work [18], we provide the LLM with summarized categories from past fixes while allowing the LLM to propose new categories. Failed thoughts are recorded to help the LLM avoid ineffective strategies, following the Reflection paradigm [19].

In the inner loop, FLAKYGUARD applies and validates LLM-suggested fixes. Fix application is detailed in Section III-D. Validation consists of build validation (ensuring compilation) and test validation (rerunning tests the same number of times as failure reproduction). For compilation failures, we attempt best-effort repair by providing the LLM with the original code, the modified code, and the compilation errors. Test validation requires all runs to pass; any error or timeout triggers a revert and iterating to the next fix. FLAKYGUARD sends the fix to developers for review upon successful validation.

D. Test Simplification and Patch Transplantation

Following the table-driven testing practice [15], each test function contains multiple similar test cases organized into tables, which creates challenges for fixing: the LLM may focus on the wrong test case, or search-and-replacement patching may fail due to ambiguity. We develop AST-based *test simplification* and *patch transplantation* techniques to address these challenges.

Before flakiness reproduction, FLAKYGUARD simplifies the test function T_{orig} to keep only the targeted test case t while removing all other sibling test cases, yielding a simplified test function T_{simp}. We disable simplification for tests that need interference with others to manifest. This simplification process traverses the AST to record the byte offsets of all changes in an editing tracker. To preserve offset validity during modification, the changes are applied in reverse order [17]. After simplification, some variables become unused, causing compilation errors. We handle these errors iteratively by commenting them out, adding annotations to them for later restoration.

We send T_{simp} to the LLM to produce the fix T'_{simp}. The challenge is then transplanting these edits back to the full context of T_{orig}, yielding a patched test function T'_{orig}:

$$\texttt{Patch}(T_{\text{simp}}, T'_{\text{simp}}, T_{\text{orig}}) \rightarrow T'_{\text{orig}}$$

We use a two-step AST-based transplantation approach. First, we extract the AST from T'_{simp} and replace the enclosing test table with its counterpart from T_{orig}. This process preserves the fixes outside the test table, which is crucial because the LLM may suggest changes to the surrounding code, not just the test case itself. Second, we reload the merged AST and precisely replace the specific test case node from T_{orig} with the corresponding fixed node from T'_{simp}. This approach ensures that fixes are correctly propagated while maintaining the original table structure.

IV. Evaluation

To evaluate the effectiveness and usefulness of FlakyGuard in producing fixes for flaky tests, we conducted a series of experiments designed to answer the following research questions:

1) **RQ1.** How effective is FlakyGuard at fixing flaky tests?
2) **RQ2.** What is the contribution of each component of the technique to the overall results?
3) **RQ3.** How does FlakyGuard compare with the state-of-the-art approaches?
4) **RQ4.** How much do the results change when utilizing a different model?
5) **RQ5.** How do developers perceive the usefulness of FlakyGuard?

A. Experiments Setup

We use the Go monorepo and its real-world flaky tests in Uber to conduct our evaluation. This monorepo consists of over 100 million lines of Go code across more than 100 distinct projects, developed by over 6000 engineers from 100+ teams spanning diverse domains including storage systems, machine learning applications, and distributed services. Our evaluation dataset provides comprehensive coverage of different types of flaky tests since it systematically captures every flaky test occurrence rather than being constrained to predetermined categories. We choose ChatGPT o1 [20] and Claude 3.7 Sonnet [4] as the LLM models, because they are the state-of-the-art models available at the time of this evaluation.

For our experiments, we set a time limit of 2 hours for each fixing process. We run each test 1000 times to collect a failing run. To increase the chance of detecting scheduling-related flaky failures, we enabled the race detection flag [21] to introduce randomness at scheduling points.

For the parameters described in Sections III-B and III-C, we control the number of attempts for collecting context M to be 3, the number of high-level thoughts P per context to be 2, and the number of fix attempts per thought N to be 3. The depth limit d is set to infinite (i.e., max integer). The number of functions F that the postprocessing retains is maximally 5.

B. RQ1: How effective is FlakyGuard *at fixing flaky tests?*

We analyzed a total of 1115 flaky tests that FlakyGuard attempted to fix, over a period of six months. Among these flaky tests:

- FlakyGuard reproduced $\mathbf{71.6}\%$ of the tests (i.e., $\mathbf{798}$ tests),
- FlakyGuard produced fixes for $\mathbf{47.6}\%$ (i.e., $\mathbf{380}$) of the reproducible tests
- $\mathbf{51.8}\%$ of the fixes (i.e., $\mathbf{197}$ fixes) were accepted by developers and successfully landed.

Table I shows the breakdown of accepted fixes by root cause category. Standard benchmarks are limited to few simple categories, while our industrial setting challenges us with many more complex categories. FlakyGuard addresses a broader range of flakiness types compared to existing automated tools.

TABLE I: Breakdown of the accepted fixes

Category	Frequency	
	Count	Percentage
Schedule randomness	72	37%
Random iteration of unordered collections	65	33%
Timestamp discrepancy	24	12%
State pollution	16	8%
Time-dependent flakiness	13	7%
Others	7	3%

Schedule randomness. Schedule randomness is the most common reason for flaky tests. Go provides extensive support for concurrency, which introduces inherent non-determinism due to thread scheduling. We observe several examples of flaky tests in this category similar to those described in prior work, such as due to asynchronous waits [22, 23].

We show a Go-specific example in Listing 2. The test function executes `emitLoop`, which has a `select` clause. In Go, the language construct `select` is non-deterministic: when multiple cases are ready, the runtime picks one randomly. In this example, if the runtime picks the first case, it executes only one iteration of the loop. Otherwise, it executes two iterations of the loop, which will fail at the `Get` mock call, since it is set up to run only once. FlakyGuard proposed a fix to increase the `timerPeriod` so that `ticker.C` becomes ready later than `ctx.Done()`, ensuring the first case is reliably selected. Developers accepted this fix.

We observed around 45% of the flaky tests with schedule randomness are due to use of `select` like the one shown above. Other reasons for schedule randomness flaky tests include asynchronous wait (42%) and atomicity violations, e.g., threads interleave in a way that causes undesired program state (13%). It is important to note that the LLM needed context from the production code, not just the test code, to determine a fix and produce the reasonable root cause explanation.

Random iteration of unordered collections. These flaky tests stem from iterating over unordered data structures, such as Go maps, which do not guarantee consistent traversal order across executions, but the tests implicitly assume a fixed element order [24]. Typical mitigation strategies include using relaxed assertions like `ElementsMatch(...)` or explicitly sorting elements prior to comparison. In some complex cases, the results are serialized into strings in the assertion, for which FlakyGuard deserializes them first.

Listing 3 shows an example of such a flaky test. The test sets up the mock so that it accepts an expected input string. However, in production code, the input string is computed from iterating over a map, as shown in function `format`

```
// metrics_test.go
t.Run("FailoverCompleted", func(t *testing.T) {
    emitted := runTest(t,
        func(mockK8sClient *clientmock.MockClient) {
            mockK8sClient.EXPECT().
                Get(...).
                DoAndReturn(...)
        },
        func(e *Emitter, ctx context.Context) {
-       e.emit(ctx, time.Millisecond)
+       e.emit(ctx, 200 * time.Millisecond)
        },
    )
// metrics.go
func (emitter *Emitter) emit(ctx context.Context,
      timerPeriod time.Duration) {
   ...
   emitLoop(ctx, timerPeriod)
   ...
}
func (emitter *Emitter) emitLoop(ctx context.Context
      , timerPeriod time.Duration) {
    ticker := time.NewTicker(timerPeriod)
    defer ticker.Stop()
    for {
        err := emitter.K8sClient.Get(...)

        select {
        case <-ctx.Done():
            return
        case <-ticker.C:
        }
    }
}
```

Listing 2: A Go-specific flaky test due to the schedule randomness and the select construct.

```
// decider_test.go
expected := "-9..-1=node/backup,node/backup,node/
      backup\n-1..10=node/backup,node/backup,node/
      backup\n"

mockStorage.EXPECT().Write(
    gomock.Any(),
-   gomock.Eq(strings.NewReader(expected))
+   hdfsContentMatcher{expected, expected2}
).Return(nil)

// production code
func format(input map[string][]string) string {
    var sb strings.Builder
    for token, nodes := range input {
        sb.WriteString(token)
        sb.WriteString("=")
        sb.WriteString(strings.Join(nodes, ","))
        sb.WriteString("\n")
    }
    return sb.String()
}
```

Listing 3: A flaky test due to non-deterministic map iteration order that leads to an unexpected mock input.

embedded inside some deep function call chain, which may return different input strings depending on the iteration order. If the input string is not expected, the mock would not be called, thereby leading to an error. FlakyGuard addresses this problem by providing customized matcher logic for the mock setup that accepts alternative input strings (expected2 is omitted for simplicity).

Timestamp discrepancy. If a data structure references a timestamp field transitively, that field can get different time.Now() values at different sites, e.g., at the expected value initialization vs the actual initialization. FlakyGuard typically fixes this type of flaky test by setting the timestamp fields to a constant value or using a comparison logic that ignores those fields. We find that FlakyGuard needs to extract context from the production code so it can provide the detailed root cause explanation about where the timestamp difference comes from.

```
// test code:
-os.Setenv("APP_ENV", tc.envVar)
+t.Setenv("APP_ENV", tc.envVar)
populateContextMetadata(ctx)

assert.Equal(t, expectedValue, getSpanBaggage(ctx))

// production code:
func populateContextMetadata(ctx context.Context) {
    span := opentracing.SpanFromContext(ctx)
    group := "default"
    if env := os.Getenv("APP_ENV"); env != "" {
        group = "group_" + strings.ToLower(env)
    }
    span.SetBaggageItem(group)
}
```

Listing 4: A flaky test due to the pollution.

State pollution. When multiple tests run in parallel, the state can be modified by other tests [25, 26]. The pollution can come from shared global state, file system, or databases. FlakyGuard generates fixes for flaky tests whose pollution comes from all of these sources.

Listing 4 shows an example of such a flaky test. Each test sets the APP_ENV environment variable (e.g., "Staging" or "Production") and expects a specific baggage value like "group_staging" or "group_production" to be set on the span in the context. However, since environment variables are global to the process, concurrent tests may overwrite each others' APP_ENV values. As a result, one test may see baggage derived from another test's environment setting, causing unexpected assertion failures. The sharing of data through these environment variables leads to nondeterministic behavior when tests are run in parallel. FlakyGuard fixed this test by using the t.Setenv() API, following best practice [27], which ensures each test sets and restores the environment properly, and it disallows the tests to run in parallel [27].

Time-dependent flakiness. The flaky tests in this category depend heavily on the wall clock time. Listing 5 shows an example flaky test where there is a deep call chain, i.e., GetSupplyChanges → validateTimeRange → validateTimes → validateAge, which leads to the leaf node function validateAge that explains the root cause. The function computes a new cutoff, in the same way as the test computes the global variable cutoff. However, the two time.Now() calls return different values at runtime. Depending on whether they are truncated to the same second interval, the branch condition at line 18 may be true or false, thereby making the function return the error non-deterministically.

FlakyGuard fixes the test by shifting beginTime by 1 second to offset the time elapsed between the test's start and the invocation of validateAge, so that beginTime is not before the dynamically computed cutoff, avoiding false failures due to the time-dependent flakiness. Such flakiness is hard to fix or root cause without the production code context.

Others. There are other miscellaneous reasons for flaky tests that FlakyGuard can fix. One example test draws an insufficient number of samples before asserting over statistical

```
// handler_test.go
var cutoff = time.Now().UTC().AddDate(0, -Age, 0).
     Truncate(time.Second)
func TestGetSupplyChanges(t *testing.T) {
-   beginTime := common.TimeToMS(cutoff)
+   beginTime := common.TimeToMS(cutoff.Add(1 *
     time.Second))

    request := &GetSupplyChangesRequest{
        StartTime: beginTime,
        ...
    }
    ...
    _, err := handler.GetSupplyChanges(ctx, request)
    assert.NoError(t, err)
}
// a deep call chain leading here.
func validateAge(...) {
    cutoff := time.Now().UTC().AddDate(0, -Age, 0).
         Truncate(time.Second)
    if beginTime.Before(cutoff) {
       ...
       return err
    }
    return nil
}
```

Listing 5: A time-dependent flaky test

properties such as variance. FLAKYGUARD fixes the test by having it draw more samples. Another example test builds random inputs, where some of the inputs are invalid and break assumptions in the production code. FLAKYGUARD generated a fix that restricts the space of random inputs to comply with the assumptions.

Analysis of the fixes that were not accepted. 48.2% of the proposed fixes were not accepted by the developers. One major reason is that the generated fix did not reuse the test helper APIs. Some organizations build such APIs to enforce best practices to ease testing. The LLM we use is not aware of these custom APIs and hence does not use them. Interestingly, through our context collection, the LLM can read the API implementations and copy them over into the fixes. We plan to investigate how to guide LLMs to reuse the APIs directly, which needs to resolve the Bazel build dependencies [28].

Another major reason is that each flaky test may have multiple fixes, where the easier ones are more likely to be produced and sent to developers, but the more complex fixes are more proper and what developers want. In Listing 1, an easy fix would be to relax the mock so that it can be called maximally once. However, developers expressed to us that they do not want such fixes.

Beyond these reasons, a handful of fixes changed the test semantics, e.g., removing some assertions, adding irrelevant test assertions. We also saw fixes got rejected because of the readability issues, e.g., the fixes are too long or too complex.

Last but not least, we observed that developers manually fixed some flaky tests in parallel with FLAKYGUARD's fixing pipeline, and they prefer to submit their own fixes.

C. RQ2: What is the contribution of each component of the technique to the overall results?

We conducted an ablation study to study the contribution of each component of FLAKYGUARD. To conduct a controlled experiment (used hereafter up to Section IV-E), we selected a single commit of the monorepo (at the time we started the experiment) and performed the evaluations against the 295 flaky tests that are reproducible in this commit. We used ChatGPT o1 [20] and set a two-hour time limit for each test case to fix.

For this ablation study, we did not consider developer feedback, as it would have introduced extensive back-and-forth processes beyond the scope of the automated setup. Instead, we relied on test validation and spot checking to achieve large-scale results. Although this approach may over-count the impact of certain components, it ensures consistent evaluation criteria across all experimental configurations, thus maintaining comparative validity.

We break the evaluation of this research question into two parts:

RQ2.1 How effective are different context collection strategies at fixing the flaky tests?

RQ2.2 How effective is the test simplification logic at fixing the flaky tests?

How effective are different context collection strategies at fixing flaky tests? Table II presents an ablation study addressing RQ2.1. We evaluate four configurations: (1) ***FlakyDoctor***, which is our re-implementation of FlakyDoctor [6] that provides only the test code context; (2) ***BFSRepoGraph***, which performs BFS traversal of static RepoGraphs [13]; (3) ***BFSDCG***, which performs BFS traversal of dynamic call graphs; and (4) **FLAKYGUARD**, which applies selective traversal of dynamic call graphs (Section III-B). Note that *BFSRepoGraph* serves as another baseline approach similar to FlakyDoctor and does not strictly belong to the ablation study of our technique's components. Configurations (2)-(4) collect relevant production code context starting from the test. Furthermore, for all four configurations, we applied the test simplification techniques described in Section III-D to help them deal with the challenges imposed by the table-driven testing practice.

FlakyDoctor generates 134 fixes, while *BFSRepoGraph* generates 149 fixes. The improvement over *FlakyDoctor* demonstrates the usefulness of the production code context. On the other hand, *BFSRepoGraph* does not significantly outperform *FlakyDoctor*. One contributing factor is that the code context, i.e., the transitive closure over the static graph, can get so big that it confuses the LLM.

BFSDCG generates 161 fixes. The improvement over *BFSRepoGraph* highlights the value of the dynamic call graph, which helps produce more focused and precise context by excluding the unexecuted branches and precisely resolving the dynamic call dispatch. In addition, we observed cases where *BFSDCG* collects the relevant context hidden behind the reflection calls or anonymous functions, while *BFSRepoGraph* is unable to.

We would like to mention that the performance gap between *BFSDCG* and *BFSRepoGraph* would be substantially larger without test simplification. Most test functions follow table-driven testing practices [15], containing typically 10+ test cases. Without simplification, *BFSRepoGraph* presents the

TABLE II: Comparison of different context collection strategies.

Approach	Fixed Tests	Success Rate (%)
FlakyDoctor	134	45.42
BFSRepoGraph	149	50.51
BFSDCG	157	53.22
FlakyGuard	194	65.76

TABLE III: Comparison of FLAKYGUARD with *Agentless+RepoGraph* and *AutoCodeRover*.

Approach	Fixed Tests	Success Rate (%)
Agentless+RepoGraph	159	53.90
AutoCodeRover	158	53.56
FlakyGuard	194	65.76

complete test function to the LLM, causing it to often focus on unrelated test cases (in around 60% of cases).

FLAKYGUARD can successfully generate 194 fixes, which outperforms *BFSDCG* by 20.5%, thanks to the selective context collection. Dynamic call graphs can be substantial, with the node count reaching hundreds and the graph depth extending to seven levels in extreme cases. In the cases that can be fixed by FLAKYGUARD but not by *BFSDCG*, FLAKYGUARD helped greatly in selecting a smaller set of nodes as context and letting the LLM focus on a smaller relevant scope.

To quantify the effectiveness of our core contribution of intelligent context pruning, we analyzed the pruning mechanism in detail. In Listing 1, FLAKYGUARD selected 27 nodes from the DCG with 264 nodes, then further limited it to 5 nodes via the global filtering. This dramatic reduction from hundreds of nodes to approximately 5 maximally relevant functions enables the LLM to focus on semantically important code while eliminating extraneous information. Manual inspection of cases fixed by *BFSDCG* revealed that the root cause nodes are selected by FLAKYGUARD in all but two cases, confirming the precision of our selection mechanism.

How effective is the test simplification logic at fixing the flaky tests? We conduct an ablation study to assess the effectiveness of the test simplification (Section III-D). When comparing FLAKYGUARD before and after applying simplification, the generated fixes increase from 160 to 194, demonstrating that simplification significantly improves performance in addressing the complexity introduced by table-driven testing practices, which result in test functions with multiple similar-looking tests. Sometimes, the tests only differ with a few words in the description and the test body differs only on few arguments, which imposes a great challenge to the LLM and often misleads it to look into some other irrelevant tests. The test simplification helps the LLM focus only on the test case of interest and the functions actually called by it.

D. RQ3: How does FLAKYGUARD *compare with the state-of-the-art approaches?*

We compare FLAKYGUARD with *Agentless+RepoGraph* [11, 13] and *AutoCodeRover* [12]. *Agentless+RepoGraph* and *AutoCodeRover* are the state-of-the-art approaches designed to fix Python bugs (evaluated on SWE-Bench). We extracted the context collection components from the publicly available code, which we believe is the major factor for the success of the LLM.

For *Agentless+RepoGraph*, we combine Agentless [11] and RepoGraph [13] with modifications to the RepoGraph component. Since the original approach searches for callers to identify the failure-inducing usage at the caller side, we adapted it to search for callees to understand implementation details that cause flakiness. We use one-hop traversal following the default configuration [13, 16]. We exclude the embedding-based file search from Agentless [11] as it requires natural language descriptions from developers, which are not available in our problem setting.

For *AutoCodeRover*, we directly use the existing technique without additional customizations, like we did for *Agentless+RepoGraph*. It iteratively searches for relevant functions starting from the flaky test and stops when it believes it has found sufficient context for fixing.

We enhanced *Agentless+RepoGraph* and *AutoCodeRover* by applying the techniques in Section III-D to tackle the challenges imposed by the table-driven testing in order to produce meaningful results. Without these enhancements, they would constantly get confused working on some irrelevant test cases or encounter patching errors due to the matching ambiguity.

Table III shows the results of the comparison. *Agentless+RepoGraph* generates 159 fixes, each taking 1528.60 seconds on average. *AutoCodeRover* generates 158 fixes, each taking 1799.74 seconds on average. In contrast, FLAKYGUARD fixes 194, each taking 1978.73 seconds on average. FLAKYGUARD outperforms *Agentless+RepoGraph* or *AutoCodeRover* by 22%, which quantitatively confirms the effectiveness of our technique. In terms of overall efficiency, although FLAKYGUARD incurs additional overhead for code instrumentation and runtime call graph collection, we observed FLAKYGUARD is usually more efficient in the LLM reasoning and fixing phases than the other two because (1) with shorter context, each LLM response is faster; and (2) with more relevant context, the LLM succeeds within fewer iterations. The initial instrumentation overhead is justified by the subsequent reduction in patch generation time.

Figure 2 shows a Venn diagram illustrating the overlap of the fixes produced by each approach. No approach's fixed tests are a subset of any other, suggesting that each one has its pros and cons. We manually inspected the tests that each approach cannot fix while the others can fix, during which we paid special attention to the contexts collected.

Flaky tests that *Agentless+RepoGraph* cannot fix. We manually analyzed 35 flaky tests that were fixed by FLAKYGUARD but not by *Agentless+RepoGraph*, identifying three

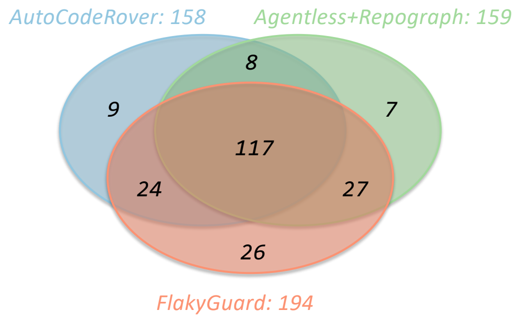

Fig. 2: Venn diagram for the fixes generated by different approaches

main reasons *Agentless+RepoGraph* could not fix tests. First, due to its lack of dynamic information, *Agentless+RepoGraph* often picks the wrong function or includes code branches not executed by the test, resulting in large and noisy prompts that confuse the LLM (13 tests). Second, *Agentless+RepoGraph* relies on static name-based analysis, which causes it to miss relevant anonymous functions or functions invoked via reflection (9 cases). Third, the Repograph in *Agentless+RepoGraph* employs a shallow graph traversal strategy: it uses a one-hop call graph by default [13] to avoid exponential context growth, which leads to missing the root causes deeply nested in the leaf nodes (9 cases).

Flaky tests that *AutoCodeRover* cannot fix. *AutoCodeRover* failed to fix 36 flaky tests due to three main reasons. First, *AutoCodeRover* often terminates early after identifying a potential bug location, but in some cases, additional surrounding context is required for successful fixing (15 cases). Second, like *Agentless+RepoGraph*, *AutoCodeRover* can include excessive irrelevant context (11 cases). Third, due to the limitations of static analysis, *AutoCodeRover* also misses the relevant anonymous functions or reflection-based functions (9 cases).

Flaky tests that FlakyGuard cannot fix. Among the flaky tests we analyzed, 24 were fixed by either *Agentless+RepoGraph* or *AutoCodeRover* but not by FlakyGuard. These failures fall into three categories. First, FlakyGuard either missed relevant functions or included irrelevant ones, despite traversing the dynamic call graph (9 cases). However, compared to other approaches, the graph-guided traversal greatly reduced the number of such cases overall. Second, the global filtering FlakyGuard uses during postprocessing removed relevant functions, causing the LLM to fail in the fixing (8 cases). Third, FlakyGuard provided similar context as the other tools did but failed to generate a fix, which may be due to differences in the function ordering within the prompt or inherent randomness in using the LLM (7 cases).

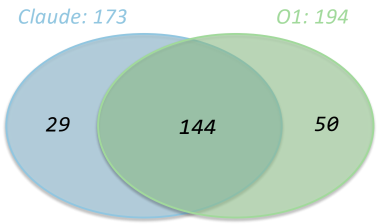

Fig. 3: Venn diagram for the fixes generated with different models

E. RQ4: How much do the results change when utilizing a different model?

We compared ChatGPT o1 [20] and Claude 3.7 Sonnet [4] in this study. Figure 3 shows a Venn diagram illustrating the overlap of the fixes produced with each model. No model's fixed tests are a subset of the other, suggesting that each one has its pros and cons.

Flaky tests that Claude 3.7 Sonnet cannot fix. Claude 3.7 Sonnet failed to fix 50 flaky tests that ChatGPT o1 successfully addressed. In 38 tests, Claude 3.7 Sonnet selected different contexts during the graph traversal, which yielded incorrect fixes. In 12 tests, Claude 3.7 Sonnet selected the same context as ChatGPT o1 but still produced incorrect fixes. In 4 out of these 12 cases, Claude 3.7 Sonnet derived the wrong high-level root causes. Due to the black box nature, it is beyond our capability to explain what happened. When we manually inspected these cases, we found that the root causes given by Claude 3.7 Sonnet were also plausible, but the fixes failed the automatic validations.

Flaky tests that ChatGPT o1 cannot fix ChatGPT o1 failed to fix 29 flaky tests that Claude 3.7 Sonnet successfully addressed. In 22 tests, the two models selected different context during graph traversal, which yielded incorrect fixes. In 7 tests, ChatGPT o1 shared the same context with Claude 3.7 Sonnet, yet still produced incorrect fixes. In 3 out of these 7 cases, ChatGPT o1 derived the wrong high-level root causes. Like the previous case of Claude 3.7 Sonnet, ChatGPT o1 gave plausible but subtly wrong explanations. The fixes did not pass the validations.

F. RQ5: How do developers perceive the usefulness of FlakyGuard?

We sent out surveys to developers to understand their perceptions to FlakyGuard, receiving 19 responses. Table IV shows the results. All 19 respondents (100%) reported that FlakyGuard's root-cause explanations are useful.

On a five-point scale, developers rated the quality of the fixes produced by FlakyGuard very highly (mean 4.42, standard deviation 0.77), while the complexity of the underlying root causes was judged to be moderate (mean 2.73, standard deviation 1.10).

In terms of time savings per individual flaky test case, 3 participants (15.8%) estimated saving between one and eight hours, 8 (42.1%) estimated saving less than one day, 6 (31.6%)

TABLE IV: Survey Results on Developers' Perceptions of FlakyGuard

Usefulness of the root cause explanation	
Usefulness	**Count(%)**
Uesful	19 (100%)
Quality and Complexity of flaky tests fixed by FlakyGuard	
Metric	**Average Rating (stddev)**
Quality of Fixes (1-5)	4.42 ±0.77
Complexity of Root Causes (1-5)	2.73 ±1.10
Estimated Time Saved by FlakyGuard	
Time Saved	**Count(%)**
1 to 8 hours	3 (15.8%)
less than 1 day	8 (42.1%)
1 to 2 days	6 (31.6%)
2 to 4 days	2 (10.5%)

estimated saving one to two days, and 2 (10.5%) estimated saving two to four days when resolving flaky tests with FlakyGuard. Together, these results suggest that not only do developers find FlakyGuard's diagnostic information valuable, but they also perceive FlakyGuard's solutions to be of high quality, as well as saving them time.

G. Limitations and Threat to Validity

LLM reasoning and fixing are naturally affected by non-determinism, where the LLM may fail to repair a flaky test in one attempt but succeed in a later attempt even with identical context. LLM non-determinism represents a potential threat to evaluation validity [29]. To explicitly mitigate this issue, our experimental design incorporates multiple retry mechanisms with $M = 3$ context collection attempts, $P = 2$ parallel reasoning paths, and $N = 3$ fix generation attempts per path (as detailed in Section IV-A), providing 18 total repair opportunities per test case. These retry mechanisms directly help reduce the impact of non-deterministic LLM behavior by providing multiple chances for successful repair. Additionally, our concrete and detailed prompts are designed to minimize output variability, which according to Ouyang et al. [29] can significantly reduce non-determinism in LLM responses.

H. Discussion

FlakyGuard's methodology is generally applicable to other languages. We implemented a version for Java and confirmed it works on Java flaky tests at Uber. The approach requires minimal language-specific adaptation, where only the flakiness pattern prompts need to be adjusted. The patterns are shown in Table I, where each pattern is described by a single sentence. One can add prompts for other languages easily.

We chose Go as our evaluation target because it exhibits a rich variety of flakiness patterns in our industrial setting. Our dataset spans more than 100 projects with diverse business logic, revealing flakiness types not extensively studied in prior work. This comprehensive coverage motivated the development of FlakyGuard.

The techniques we employ are standard software engineering practices. Table-driven testing follows official Go guidelines, and after simplification, these tests become conventional unit tests. The flakiness patterns we address represent universal challenges in concurrent software development across languages and organizations.

The main change we had to make was on the dynamic call graph collection part, which needs instrumentation, which we can do with most modern compilers or even with Tree-sitter [30] at the AST level, since we only need to insert one API call at the method entry. When dynamic call graphs are unavailable, one can still apply the LLM-guided graph exploration on the static call graph or other scalable graph format [31]. In addition, we made new implementations of the test simplification and patch transplantation logic.

Our open-source implementation includes dynamic call graph collection mechanisms, LLM-guided context selection algorithms, and automated patch generation capabilities. The implementation and evaluation artifacts are available at: https://sites.google.com/view/flakyguard.

V. Related Work

Luo et al. conducted the first empirical study on flaky tests in open-source projects, categorizing the reasons they are flaky as well as how developers fixed them [22]. Following this study, researchers developed techniques to detect and repair specific types of flaky tests automatically. For example, to detect order-dependent flaky tests, which are tests whose pass/fail outcome depend on which tests run before it due to pollution in shared state, researchers proposed techniques to run tests in different orders [32, 33, 34, 35, 36, 37, 38, 39] or to track shared state between tests to see what could be polluted [40, 41]. Researchers later developed techniques to automatically repair these flaky tests by identifying code within the application that can be used as a patch into the test code to reset the shared state [7, 8]. Shi et al. [24] developed NonDex to detect tests that fail due to assuming an ordering on unordered collections by exploring how tests behave when controlling over the iteration order on those collections [24, 42]. Zhang et al. later developed DexFix to repair these flaky tests [9]. For timing-dependent flaky tests, researchers developed techniques that efficiently explore where to insert delays that can reliably reproduce their failures [43, 44]. Techniques for repairing timing-dependent flaky tests include adjusting existing wait times in the code [45, 46] or synchronizing the critical points [23]. We found examples of all of these flaky tests in Uber, and we leverage our LLM-based approach to automatically repair them.

Recently, researchers are using LLMs to solve flaky test-related problems, such as predicting which tests are flaky [47, 48, 49] or debugging flaky tests [48, 50, 51]. Our work is most similar to FlakyDoctor [6], an LLM-based approach to automatically repair flaky tests. FlakyDoctor is effective at repairing order-dependent flaky tests and flaky tests that assume deterministic ordering of unordered collections, repairing more tests than non-LLM approaches [7, 8, 9]. Our approach also collects context from test code and executions to prompt an LLM for a solution. However, we leverage a more generic way of collecting context based on searching over a dynamic call graph, allowing us to generally handle more types of flaky tests, including those that FlakyDoctor handles. Fatima et al. evaluated the effectiveness of LLM-based flaky test repair if the model also received the fix category along with some examples, showing this extra information helps generate better fixes [18]. We also prompt the LLM by summarizing the pattern from previous examples to the model to help predict the category and generate a fix, along with extra context.

Recent work leveraged LLMs with different context extraction methods for various software engineering tasks. Agentless [11] uses hierarchical localization from repository overview to specific code elements to obtain the context. AutoCodeRover [12] combines LLM with program analysis, using stratified retrieval and iterative API calls to gather relevant code snippets. RepoGraph [13] creates a code graph using Tree-sitter, enabling k-hop ego-graph retrieval for related semantic context. We adapt all three of these past approaches for our goal of repairing flaky tests, to use as comparison against FLAKYGUARD. FLAKYGUARD leverages a dynamic call graph and a selective LLM-based graph search to more effectively and efficiently repair flaky tests.

VI. CONCLUSIONS

We presented FLAKYGUARD, a novel approach that addresses the context problem in LLM-based flaky test repair through selective exploration of dynamic call graphs. Our evaluation on real-world industrial flaky tests shows that FLAKYGUARD achieves high developer acceptance, outperforming existing methods by at least 22%. The approach effectively balances providing sufficient context for accurate repairs while avoiding information overload, and developers find the root cause explanations universally useful. FLAKYGUARD represents a significant advancement in automated flaky test repair for industrial software development.

In addition, FLAKYGUARD has been deployed as a fully autonomous system at Uber, integrating with the existing ticketing system to automatically process flaky test reports and deliver fixes to developers daily.

REFERENCES

[1] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, “Modeling and ranking flaky tests at apple,” in *International Conference on Software Engineering, Software Engineering in Practice*, 2020, pp. 110–119.

[2] J. Micco, “The state of continuous integration testing@google,” https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html, 2017.

[3] OpenAI, “Gpt-4 technical report,” OpenAI, Tech. Rep., 2023, arXiv:2303.08774. [Online]. Available: https://arxiv.org/abs/2303.08774

[4] Anthropic, “The claude 3 model family: Opus, sonnet, haiku,” Anthropic, Tech. Rep., 2024. [Online]. Available: https://www.anthropic.com/news/claude-3-family

[5] Anthropic, “Tracing the thoughts of a large language model,” *Anthropic Research*, 2025. [Online]. Available: https://www.anthropic.com/research/tracing-thoughts-language-model

[6] Y. Chen and R. Jabbarvand, “Neurosymbolic repair of test flakiness,” in *International Symposium on Software Testing and Analysis*, 2024, pp. 1402–1414.

[7] C. Li, C. Zhu, W. Wang, and A. Shi, “Repairing order-dependent flaky tests via test generation,” in *International Conference on Software Engineering*, 2022, pp. 1881–1892.

[8] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, “iFixFlakies: A framework for automatically fixing order-dependent flaky tests,” in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 545–555.

[9] 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 *International Conference on Software Engineering*, 2021, pp. 50–61.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *International Conference on Neural Information Processing Systems*, 2017, pp. 6000–6010.

[11] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, “Agentless: Demystifying llm-based software engineering agents,” *The Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 801–824, 2025.

[12] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, “AutoCodeRover: Autonomous program improvement,” in *International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[13] S. Ouyang, W. Yu, K. Ma, Z. Xiao, Z. Zhang, M. Jia, J. Han, H. Zhang, and D. Yu, “RepoGraph: Enhancing ai software engineering with repository-level code graph,” in *Internationl Conference on Learning Representations*, 2025, poster Presentation. [Online]. Available: https://openreview.net/forum?id=dw9VUsSHGB

[14] Y. Hu, Z. Lei, Z. Zhang, B. Pan, C. Ling, and L. Zhao, “GRAG: Graph retrieval-augmented generation,” *CoRR*, vol. abs/2405.16506, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2405.16506

[15] The Go Authors, “Table driven tests,” https://go.dev/wiki/TableDrivenTests, 2025, accessed: 2025-05-04.

[16] O. Yyshr, “Repograph: localize.py,” 2024, accessed: 2025-05-21. [Online]. Available: https://github.com/ozyyshr/RepoGraph/blob/6c3977d87845993bf2c0359b4ac752278d7f3c45/agentless/fl/localize.py

[17] The rules_go team, “Saving the nogo fixes,” https://github.com/bazel-contrib/rules_go/pull/4102, 2025, ac-

cessed: 2025-05-04.

[18] S. Fatima, H. Hemmati, and L. Briand, "FlakyFix: Using large language models for predicting flaky test fix categories and test code repair," *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3146–3171, 2024.

[19] Reflection, "Reflection agents," https://blog.langchain.dev/reflection-agents/, 2025, accessed: 2025-05-29.

[20] "Introducing openai o1 — openai," https://openai.com/index/introducing-openai-o1-preview/, (Accessed on 11/10/2024).

[21] B. Contributors, "Using the race detector," https://github.com/bazel-contrib/rules_go/blob/master/go/modes.rst#using-the-race-detector, 2025, accessed: 2025-05-30.

[22] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.

[23] S. Rahman and A. Shi, "FlakeSync: Automatically repairing async flaky tests," in *International Conference on Software Engineering*, 2024, p. 1673–1684.

[24] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *International Conference on Software Testing, Verification, and Validation*, 2016, pp. 80–90.

[25] J. Candido, L. Melo, and M. d'Amorim, "Test suite parallelization in open-source projects: a study on its usage and impact," in *International Conference on Automated Software Engineering*, 2017, pp. 838–848.

[26] F. Eder and S. Winter, "Efficient detection of test interference in c projects," in *International Conference on Automated Software Engineering*, 2024, pp. 166–178.

[27] Go Authors, "testing.b.setenv," https://pkg.go.dev/testing#B.Setenv, 2024, accessed: 2025-05-27.

[28] B. Contributors, "Dependencies," https://bazel.build/concepts/dependencies, 2025, accessed: 2025-05-30.

[29] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of ChatGPT in code generation," *ACM Transactions on Software Engineering Methodology*, vol. 34, no. 2, pp. 1–28, 2025.

[30] M. Brunsfeld, "Tree-sitter: A parser generator tool and incremental parsing library," https://tree-sitter.github.io/tree-sitter/, 2018, accessed: 2025-05-07.

[31] Sourcegraph, "Announcing SCIP: A new standard for precise code intelligence," https://sourcegraph.com/blog/announcing-scip, 2022, accessed: 2025-05-07.

[32] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *International Symposium on Software Testing and Analysis*, 2014, pp. 385–396.

[33] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *International Conference on Software Testing, Verification, and Validation*, 2019, pp. 312–322.

[34] C. Li and A. Shi, "Evolution-aware detection of order-dependent flaky tests," in *International Symposium on Software Testing and Analysis*, 2022, pp. 114–125.

[35] C. Li, M. Khosravi, W. Lam, and A. Shi, "Systematically producing test-orders to detect order-dependent flaky tests," in *International Symposium on Software Testing and Analysis*, 2023, pp. 627–638.

[36] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam, "Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2021, pp. 270–287.

[37] A. Wei, P. Yi, Z. Li, T. Xie, D. Marinov, and W. Lam, "Preempting flaky tests via non-idempotent-outcome tests," in *International Conference on Software Engineering*, 2022, pp. 1730–1742.

[38] R. Wang, Y. Chen, and W. Lam, "iPFlakies: A framework for detecting and fixing python order-dependent flaky tests," in *International Conference on Software Engineering (Tool Demonstrations Track)*, 2022, pp. 120–124.

[39] N. Hashemi, A. Tahir, S. Rasheed, A. Shi, and R. Blagojevic, "Detecting and evaluating order-dependent flaky tests in JavaScript," in *International Conference on Software Testing, Verification, and Validation*, 2025, pp. 13–24.

[40] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *International Symposium on Software Testing and Analysis*, 2015, pp. 223–233.

[41] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *International Conference on Software Testing, Verification, and Validation*, 2018, pp. 1–11.

[42] 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 *International Symposium on Foundations of Software Engineering (Tool Demonstrations Track)*, 2016, pp. 993–997.

[43] S. Rahman, A. Massey, W. Lam, A. Shi, and J. Bell, "Automatically reproducing timing-dependent flaky-test failures," in *International Conference on Software Testing, Verification, and Validation*, 2024, pp. 269–280.

[44] T. Leesatapornwongsa, X. Ren, and S. Nath, "FlakeRepro: Automated and efficient reproduction of concurrency-related flaky tests," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1509–1520.

[45] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *International Conference on Software Engineering*, 2020, pp. 1471–1482.

[46] Y. Pei, J. Sohn, S. Habchi, and M. Papadakis, "Non-flaky and nearly optimal time-based treatment of asynchronous wait web tests," *ACM Transactions on Software Engineering Methodology*, vol. 34, no. 2, pp. 45:1–45:29,

2025.

[47] S. Fatima, T. A. Ghaleb, and L. Briand, “Flakify: A black-box, language model-based predictor for flaky tests,” *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1912–1927, 2023.

[48] S. Rahman, A. Baz, S. Misailovic, and A. Shi, “Quantizing large-language models for predicting flaky tests,” in *International Conference on Software Testing, Verification, and Validation*, 2024, pp. 93–104.

[49] S. Rahman, S. Dutta, and A. Shi, “Understanding and improving flaky test classification,” *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA2, pp. 320:1–320:27, 2025.

[50] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. Le Traon, “FlakyCat: Predicting flaky tests categories using few-shot learning,” in *The Automation of Software Test (AST) conference*, 2023, pp. 140–151.

[51] S. Rahman, B. N. Chanumolu, S. Rafi, A. Shi, and W. Lam, “Ranking relevant tests for order-dependent flaky tests,” in *International Conference on Software Engineering*, 2025, pp. 1999–2011.