Pinpointing Flakiness in Web Tests via DOM Event Analysis and LLM-based Reasoning

Yu Pei
SnT, University of Luxembourg
Luxembourg
yu.pei@uni.lu

Jeongju Sohn*
Kyungpook National University
Korea
jeongju.sohn@knu.ac.kr

Sarra Habchi
Cohere
Canada
sarra.habchi@cohere.com

Mike Papadakis
SnT, University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

***Abstract*—Flaky tests in web applications frequently arise from nondeterministic interactions between asynchronous events and dynamic DOM updates. Traditional debugging techniques, including fault localization, struggle to diagnose such failures due to the complex, event-driven, and continuously evolving nature of modern web interfaces. This paper introduces *DoeFL*, a novel approach that combines structured analysis of DOM event sequences with large language model (LLM) reasoning to identify the root causes of web flakiness. *DoeFL* models temporal causal relationships between DOM events and flaky test failures using Lamport logical clocks. It then uses LLMs to capture semantic inconsistencies and intent mismatches often missed by purely structural analyses. We evaluate *DoeFL* on 122 real-world flaky test cases collected from 47 web projects. Using only its Lamport-based structural component, *DoeFL* achieves 47.5% Top-1 and 82.0% Top-5 localization accuracy. When augmented with LLM-based reasoning, performance further improves, achieving Top-1 accuracy between 65.6% and 77.9% and Top-5 accuracy between 89.3% and 95.1%, depending on the selected LLM.**

***Index Terms*—Flaky Tests, Fault Localization, DOM and Events, Large Language Models**

I. Introduction

Flaky tests exhibit inconsistent pass and fail outcomes without any changes to the underlying code. This non-deterministic behavior complicates the development cycle, undermines developer confidence, and slows down debugging and maintenance [1]–[4]. Since such behavior lacks consistent triggers and observable patterns, flaky tests are inherently difficult to reproduce, understand, and diagnose. As a result, these flaky tests introduce fragility into the test suite and pose ongoing challenges to maintaining software quality [5]–[9].

Web applications are vulnerable to flakiness due to their asynchronous execution nature, dynamic DOM updates, and event-driven interactions [10]–[12]. In particular, interactions between DOM elements and related events, unpredictable timing, race conditions, and hidden dependencies between these elements make web testing scenarios unstable [13]. For instance, consider the case of a test that involves clicking a UI button and verifying the triggered reaction of the page under test. This test may occasionally fail as a result of rendering delays or missing synchronizations. To solve this issue, one needs to apply coordinated timing of the DOM events. Nevertheless, due to the numerous potential variables involved, including user-triggered events (e.g., clicks and form submissions) and state modifications in the DOM, determining the root cause of flakiness is challenging. Consequently, understanding the causal sequence of DOM interactions and their alignment with user or system intent, as reflected in test assertions and interaction context, is necessary for the diagnosis of these issues.

Modern web applications are making increasing use of complex asynchronous behaviors, which exacerbates the flakiness problem. Traditional debugging methods require manual inspection, reruns, and manual analysis, which are time-consuming and error-prone in such a dynamic environment [14]–[16]. Traditional fault localization techniques, such as coverage-based methods, have mainly been developed for back-end or general-purpose software systems. These methods typically target logic-level defects and assume deterministic behavior, relying on statistical correlations between observed failures and various program-level elements (e.g., code coverage, stack traces, or bug reports) [10], [17]–[21]. As a result, they are often inadequate or somewhat limited in terms of granularity and effectiveness in dynamic front-end scenarios where failures arise from subtle race conditions, event timing mismatches, or unobservable DOM state transitions [4], [22].

Recent advances in large language models (LLMs) have shown promise in fault localization by capturing complex patterns and contextual relationships in code and execution traces [23], [24]. However, LLMs alone lack structural guarantees, and their application to web-specific flakiness remains underexplored.

In this paper, we present **DO**M-**e**vent-based **F**ault **L**ocalization (*DoeFL*), a hybrid framework that combines Lamport logical clocks to model causal relationships among DOM events–including their interactions with specific DOM elements—with LLM-based reasoning to interpret event sequences in context. Our approach, *DoeFL*, constructs Lamport clocks to derive a logical timeline based on the captured happens-before relationships among DOM events. It then uses an LLM to analyze these event dependencies and assess whether they are likely to induce flakiness. Through this hybrid analysis, *DoeFL* identifies the most likely failure-inducing events and provides clear diagnostic cues, helping developers understand and fix flaky tests more efficiently.

In summary, we make the following contributions.

- We propose *DoeFL*, a novel tool that uses Lamport clocks

*Corresponding author

with LLM-based reasoning to localize DOM event-based flakiness. *DoeFL* uses Lamport clocks to track DOM event chains and reveals causal relationships in web test execution, helping to detect timing-sensitive behaviors that lead to flakiness.
- To further improve localization accuracy, *DoeFL* applies LLM-based analysis to interpret and reason about DOM event semantics and identify suspicious DOM event pairs. This enable the detection of contextual inconsistencies and subtle intent mismatches behind DOM events that are often overlooked by statistical or structural approaches.
- We evaluate *DoeFL* on a diverse set of real-world flaky test cases. The results show that *DoeFL* can reduce developer effort by ranking the most likely DOM event interactions that cause flakiness.

II. Background and Motivation

A. Flakiness Localization for Web Testing

Flakiness localization is a key challenge in web testing due to the inherently dynamic and asynchronous nature of modern web applications. Web tests often involve delayed responses, dynamic DOM updates, and complex event-driven interactions, all of which can lead to inconsistent test outcomes [25]. Identifying the cause locations of such nondeterministic behavior is challenging, as traditional debugging techniques typically focus on static code and fail to account for precise timing and ordering of DOM events. As a result, specialized techniques are needed to trace flaky behavior back to specific DOM elements, event interactions, or timing conditions, motivating approaches that systematically analyze test execution contexts to pinpoint flakiness more effectively [26], [27].

B. The Impact of DOM Event Sequences in Web Flakiness

DOM events, such as clicks, inputs, and navigation, are central to user interactions on the web, directly affecting application appearances. Their display order and timing can introduce non-deterministic behavior, leading to flaky tests that are difficult to debug [28]. Analyzing these interaction sequences can provide deeper insight into flaky behaviours and their patterns that traditional code analysis often misses [29]. As such, capturing and reasoning about such DOM event interaction sequences is essential for building effective flakiness localization tools in web applications [8], [30].

Diagnosing flakiness caused by asynchronous DOM event sequences requires more than observing the execution order; it requires understanding causal relationships between events. Lamport logical clocks provide a lightweight and effective means for capturing *happens-before* relationships in distributed or loosely ordered systems [31]. In the web testing context, where DOM events, listeners, and callbacks interact in non-deterministic and event-driven ways, Lamport clocks can help reveal latent causal structure behind observed behaviors [32].

C. LLM-Based Fault Localization

With the advancement of LLMs—like ChatGPT, Gemini, Llama, and DeepSeek, have proven to be exceptionally proficient in both natural language processing and code programming [33]–[36]. These models show great potential for automated fault localization and repair. For example, Qin et al. present AgentFL, a multi-agent system based on ChatGPT for automated fault localization [37]. Wu et al. provided ChatGPT with code snippets and error logs, instructing the model to identify buggy lines [38]. Likewise, Yang et al. introduced lightweight bidirectional adapters on top of LLMs to compute suspiciousness scores for each line to localize the bug line [24].

LLMs provide a promising opportunity for automating flakiness diagnosis [37], [39]. They can process natural language test descriptions, execution traces, and the intended semantics of tests to identify potential flaky patterns. By leveraging their contextual understanding, LLMs may assist in analyzing flakiness causes and issues without relying solely on manual inspection [40], [41]. However, applying LLMs to web test flakiness remains underexplored, especially in capturing the dynamic interplay between DOM elements and event sequences that leads to inconsistent test outcomes of pass and fail and hidden causal dependencies. To bridge this gap, we take a step-by-step approach: first, identify the causal relationships between DOM elements and events, and then apply an LLM to further pinpoint the sources of flakiness by reasoning over the semantic context of the interactions.

D. Motivation Example

```
it("should add same account addresses",
    async() {
  // ...

  await driver.clickElement(
    '[data-testid="account-options-
    menu__account-details"]');
  const detailsModal = await driver.
    findVisibleElement("span .modal")

  await driver.waitForSelector(".qr-code-address")
  const secondAccountAddress = await
    driver.findElement(".qr-code-address"
    )
  const secondAccountPublicAddress =
    await secondAccountAddress.getText()

  await driver.clickElement(".account-
    modal__close")
  await detailsModal.waitForElementState(
    "hidden")

  assert.equal(
    await secondAccountAddress.getText(),
    secondAccountPublicAddress
  )
})
```

Fig. 1: Motivating example

Consider the example shown in Figure 1[1]. This test verifies whether the correct account addresses are added from a modal dialog that appears after a user-click action. While the test appears to be logically correct, it occasionally fails in practice. The problem is caused by a missing synchronization point, as the test tries to access an element (`.qr-code-address`) before it is reliably rendered in the DOM structure. In

[1]https://github.com/oxgersa/metamask-extension/commit/a3d232ed

later developers' repair, an explicit wait statement (`await driver.waitForSelector("qr-code-address"))` was added to fix this problem. While this particular case appears straightforward, identifying all such missing waits—common in web applications—often demands substantial manual effort and domain expertise. Moreover, pinpointing the correct fix location (i.e., the cause of flakiness) becomes increasingly challenging under diverse interaction scenarios and UI timing conditions.

Given these challenges, this example illustrates the complex dynamics between the DOM elements structure, asynchronous behavior, and event-driven UI updates in flaky test cases. It requires a methodical approach to identify the sources of flakiness by examining DOM-event interactions and enhancing the observability of test execution. To achieve this goal, our approach integrates DOM event diagnostics with large language models (LLMs), enabling more automated, precise, and scalable flakiness localization.

III. The Approach

A. Overview

Our approach localizes DOM event-related flakiness by combining structural analysis of DOM events with LLM-based reasoning. As shown in Figure 2, *DoeFL* consists of two analysis phases: (1) *DOM event structure analysis* and (2) *Semantic interpretation of DOM interactions*.

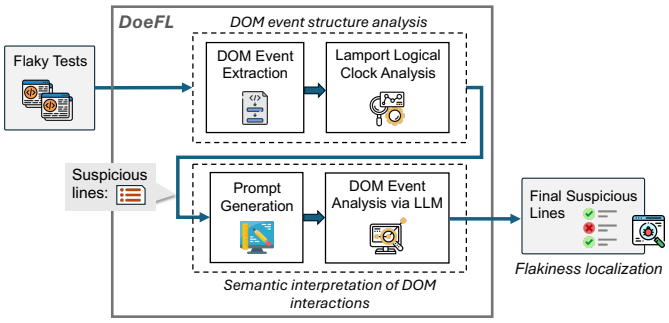


Fig. 2: Overall workflow of *DoeFL*

The structural analysis module parses the test source code to extract DOM–event interactions. These interactions consist of pairs of event operations, such as `clickElement`, and their corresponding DOM element targets; for example, Line 12 in Figure 1 captures the interaction between the `clickElement` operation and the DOM element `".account-model__close"`. By analyzing the structure of the test code, *DoeFL* reconstructs a logical sequence of DOM-event interactions. Lamport logical clocks are then applied to reason about the partial ordering of these events to capture potential temporal causal relationships between DOM-event interactions. While Lamport clocks do not directly expose low-level dependencies or race conditions, they provide a systematic way to reason about the partial orderings of events to uncover potential concurrency-related issues.

In the semantic interpretation phase, *DoeFL* generates prompts that incorporate the results of the prior Lamport-based structural analysis. This prompt guides an LLM-powered module to reason semantically about DOM-event interactions in the test code, with particular attention to code locations identified as structurally suspicious. The LLM interprets the interactions within the context (i.e., the surrounding test code and interaction sequence), identifying issues that may contribute to flaky behavior, such as unexpected event semantics, mismatches between observed behavior and intended UI flow, and unusual execution sequences. By integrating Lamport-based structural analysis with LLM-based semantic reasoning, *DoeFL* produces a ranked list of potentially flaky locations, aiding developers to focus on the most relevant interaction points.

B. DOM Event Structure Analysis

Building on the DOM event analysis, we apply Lamport logical clock analysis to capture candidate temporal causal relationships among DOM events. Lamport logical clocks, widely used in distributed systems, establish a partial ordering of events based on the *happened-before* relation. Each event/-DOM element entity is associated with a logical counter, which is incremented according to its appearance order in test code.

DoeFL maintains a local counter C_k for each context k (event or DOM element context), represented as a vertical line in Figure 3. Within a context, entities are ordered by their appearance in the test code; for each subsequent entity, C_k is incremented by one. When the analysis switches between event context and DOM element context—reflecting an interaction between these, the Lamport rule is applied to update counters. In this process, the transmitted value carries the sender's clock C_i. Upon receiving it, the recipient updates its local counter C_j, which may differ from C_i, as follows:

$$C_j = \max(C_j, C_i) + 1$$

Even when actual execution is interleaved, this mechanism guarantees that *causally dependent* entities ($e_i \rightarrow e_j$) are assigned non-decreasing logical timestamps ($C_i < C_j$).[2] In summary, *DoeFL* treats DOM and event interactions as distributed events, enabling causal reasoning through logical timestamps and constructing a partially ordered timeline of interactions in the test.

Figure 3 illustrates this analysis using the example case (in Figure 1), showing how events interact with DOM elements. Events $e1$ to $e6$ represent sequential test actions in the code, such as `click`, `find`, and `get`. The DOM elements (*DOMs*) $d1$ to $d4$ correspond to the elements referenced by these events, while C_{ei} and C_{dj} denote the local clock counters of events and DOM elements, respectively. In this example, event $e3$ at Line 9 retrieves the QR code address (`".qr-code-address"` = $d3$) immediately after the modal ($d2$) is opened. As a result, $e3$ and $d3$ are assigned Lamport clock values 3 and 4, respectively.

[2]The reverse implication "$C_i < C_j$" means that the logical order flows from e_i to e_j. As our focus is on capturing appearance-based causal relationships, Lamport clocks are sufficiently effective and cost-efficient for our purpose.

This suggests that the interaction between $e3$ and $d3$ falls between clock 3 and clock 4, as shown in orange in Figure 3. Meanwhile, the rendering process from $d2$ to $d3$ (i.e., rendering of the address element) also occurs between clock 3 and clock 4 ($C_{d2} = 3$ and $C_{d3} = 4$), also highlighted in orange in Figure 3. Combined, these indicate that the test may attempt to retrieve the QR address before the DOM is ready, resulting in a potential causal mismatch, since both interactions fall within the same Lamport clock interval, i.e., no strict *happens-before* relationship between rendering and access in this example.

The absence of a strict *happens-before* relationship in the Lamport clock analysis of *DoeFL* highlights potential flakiness, as non-ordered event-DOM interactions can lead to non-deterministic results. Therefore, *DoeFL* treats any line containing events or DOM elements that *share a Lamport clock value with other interacting entities as structurally suspicious*, since no strict happens-before relationship can be inferred.

In Figure 3, `Line 4`, `Line 6`, and `Line 9` (marked in blue) are identified as structurally suspicious, as they participate in DOM-event interactions whose ordering cannot be strictly determined by Lamport clocks. Specifically, `Line 4` involves $d1$, which shares the same Lamport clock value (2) with $e2$, while `Line 6` and `Line 9` form an interacting pair where $d2$ and $e3$ share the same clock value (3). These suspicious lines, flagged by the Lamport analysis-based structural module of *DoeFL*, are passed to the LLM-based module for further semantic reasoning. In this example, the test outcome depends on when the rendering of d_3 completes (marked in red). **Notably**, the directly interacting lines responsible for this flakiness (`Line 6` and `Line 9`) are included among those marked by the Lamport-based analysis (blue). As shown by the green part in Figure 1 (`Line 8`), this issue is resolved by adding a `waitFor` statement, enforcing synchronization to ensure that the rendering of d_3 completes before it is accessed by e_3; this shows how violations of temporal causal ordering identified through Lamport-based analysis can be mitigated by explicit synchronization between events.

C. Semantic Interpretation of DOM Interactions

Beyond constructing structured DOM event dependencies, we analyze the semantic meaning and inferred intent of DOM event interactions using large language models (LLMs). While Lamport clocks effectively identify structurally suspicious DOM-event interactions by revealing the absence of strict *happens-before* relationships, they cannot infer user intent or higher-level test context. LLMs bridge this gap by interpreting semantic aspects of DOM events, such as the expected user flow or anticipated sequence of interactions. The RESPONSE line in Figure 3 represents the semantic interpretation stage associated with DOM–event interactions (i.e., the LLM-based analysis stage).

To enable semantic interpretation, we construct prompts that combine the test code with the code locations flagged by Lamport analysis as structurally suspicious (i.e., ambiguous) (as shown in Figure 4). The test code provides functional and behavioral context for DOM interactions, while the Lamport analysis highlights interaction points where ordering cannot be strictly inferred. Together, these inputs allow the LLM to assess whether the observed interaction patterns reveal potential causal ambiguities, race conditions, or suspicious timing between DOM event sequences.

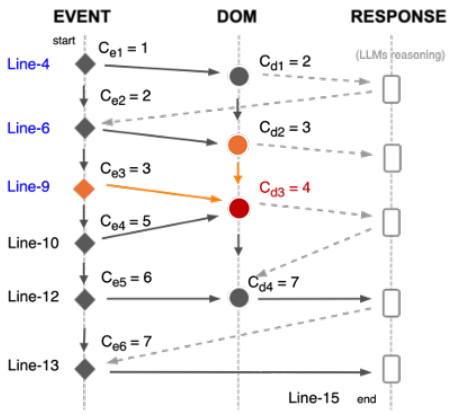


Fig. 3: Example of Lamport-based analysis of DOM–event interactions. The blue lines denote potential candidates for flakiness *marked by Lamport analysis* due to interacting with entities that share the same clock value with others. The gray dashed arrows denote potential relationships that may require further inspection by LLMs. $e1$ to $e6$ refer to `click`, `find`, `find`, `get`, `click`, and `get`, respectively. For $d1$ to $d4$, each corresponds to the account options menu button, account details modal, QR code address element, and the modal close button. The RESPONSE line represents downstream LLM-based semantic reasoning and is not part of the Lamport analysis.

The LLM returns a ranked list of suspicious DOM–event lines with natural-language explanations (**Task** in Figure 4). This has two purposes. First, it further narrows down suspicious locations by integrating structural signals from Lamport analysis with semantic reasoning, reporting only the top-k. Second, it improves the output interpretability by explaining why a particular interaction may lead to flaky behavior. By combining Lamport-based structural analysis with LLM-based semantic interpretation, *DoeFL* shows how subtle DOM–event interactions can cause non-deterministic test failures.

Prompt Template

As an AI web agent that can figure out debugging problems from given code information, especially when it comes to web flaky failures, which are types of code that make failures occur non-deterministically. Note the interactions of DOM event sequences and responses to pinpoint the suspicious DOM-event pairs or flakiness-inducing locations to cause the flakiness.
Code: [*test code snippet*]
Lamport analysis results: [*the lines flagged by the prior Lamport analysis as suspicious*]
Task: Understand the intent and recognize the recurring patterns in DOM event interactions. Provide the top 5 potential suspicious DOM event lines that are responsible for the flakiness.

Fig. 4: Prompt for identifying flaky DOM event sequences

IV. Experiment Setup

A. Interaction Rules Reflecting DOM-Event Dependencies

To better understand the localization behavior of *DoeFL*, we characterize patterns of DOM-event interactions that contribute to flaky behavior. Specifically, we introduce four rules (R1–R4) based on interaction styles. Building on the statement classification proposed in prior work on web flakiness [28], these rules describe recurring patterns, such as event-before-DOM or DOM-before-event, that may expose synchronization or timing issues. Because the notation used in [28] (E, D, R) refers to statement types, while in our paper E and D denote concrete events and DOM elements and *RESPONSE* denotes an LLM-based semantic reasoning stage, we adapt the statement-type notation to avoid ambiguity. In particular, we rename response statements (R) as assertion statements, denoted by A. The resulting statement (S) types (T) are defined as follows:

- T_{SE}: Event-triggering statements (e.g., `click`).
- T_{SD}: DOM access or structure-modification statements without events (e.g., adding elements, updating styles).
- T_{SA}: Response or assertion statements.
- T_{SED} and T_{SDE}: Combined event–DOM interaction statements, distinguished by whether the event occurs before or after the DOM action (e.g., `clickElement`).

Using these statement categories, we define four rules (R1–R4) that capture the *dominant DOM–event interaction style* associated with flaky behavior; here, "dominant" refers to the interaction style observed in the key DOM-event sequence linked to a flaky behavior. The notation T_*-T_* indicates the types of two consecutive statements that interact, where the key interaction that determines the rule is emphasized (red).

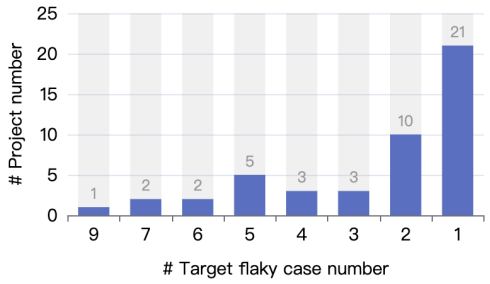
- *R1 (Event-before-DOM):* Captures interactions where an event action precedes a DOM-related operation. Example cases include T_{SE}-T_{SD}, T_{SDE}-T_{SD}, T_{SED}, etc. These cases reflect scenarios where a user action is expected to trigger or lead to DOM updates.
- *R2 (DOM-before-Event):* Captures interactions where a DOM modification occurs before an event action. Examples include T_{SD}-T_{SE}, T_{SDE}, T_{SED}-T_{SE}, etc. These patterns may indicate potential flakiness if the DOM rendering is incomplete when the event is triggered.
- *R3 (DOM/Event-before-Response):* Captures interactions where the response or assertion depends on preceding DOM or event actions. R3 includes T_{SE}-T_{SA}, T_{SD}-T_{SA}, T_{SED}-T_{SA}, T_{SDE}-T_{SA}, etc.
- *R4:* Captures remaining cases that do not fall into R1-R3, including homogeneous pairs (e.g., T_{SE}–T_{SE}, T_{SD}–T_{SD}, T_{SA}–T_{SA}) and assertion-led pairs (e.g., T_{SA}–T_{SE}, T_{SA}–T_{SD}).

B. Dataset

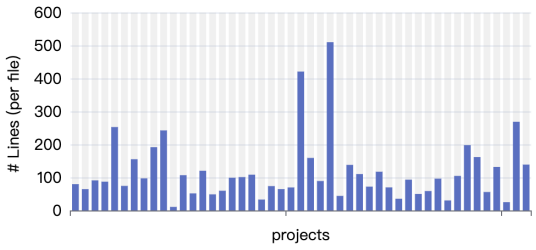
Previous studies have explored various web UI flaky tests to understand and mitigate flakiness. In this work, we focus on flaky behavior caused by DOM event issues, which are a common source of flakiness. We leverage the dataset from prior studies that specifically target DOM-related flakiness issues [28], [42]. This dataset provides commits where DOM-related flakiness was identified, forming the basis for localizing flakiness. The dataset comprises 122 flaky test cases from 47 projects across diverse domains, including browser extensions, web component libraries, monitoring and observability platforms, front-end web applications, educational ERP systems, WordPress-based systems, and online stores.

To ensure the validity of the identified flaky test cases, we conducted a two-round verification process. First, we reviewed the commit messages to ensure that the issues were related to flaky tests and DOM event interactions. Then, two authors independently reviewed the associated code changes to verify they addressed DOM event issues. This verification ensures that the dataset captures the flakiness patterns, reflecting both intentional and unintentional nondeterminism in web tests. Overall, Table I presents the collected flaky tests, along with their distribution across the four dependency rules (R1–R4), which are discussed in the following section.

1) Statistical analysis of the dataset As shown in Figure 5(a), most projects contain only a small number of flaky cases, with more than half having just one or two failures. Compared to the reported frequency of flakiness in larger, heavily maintained industrial applications such as Chromium [43], this suggests that developers may fail to capture many flaky tests due to their nondeterministic nature. Given its impact on reliability, this observation highlights the need for more effective and early flaky detection and localization. Figure 5(b) shows how the lengths of front-end test files vary across projects. Most projects have a single test file averaging around 116 lines of code, indicating that the complexity of these tests is manageable and suitable for reasoning-based localization methods.



((a)) Project distribution according to the number of target flaky cases, 122 flaky cases in total. The *x-axis* indicates the number of flaky cases per project, and *y-axis* represents the frequency of projects at each count level.



((b)) Distribution of test length across front-end test files, showing the average lines of front-end test per file (*y-axis*) for each project (*x-axis*). The total average number of lines in each file in the dataset is 116.2.

Fig. 5: Statistics about the dataset

2) Distribution of DOM-Event Dependency Rules To better understand DOM event sequences in the context of flakiness localization, we examine the distribution of DOM event dependency rules in the dataset. These rules reveal how ordering and dependency issues manifest in web flaky tests. As shown in Figure 6 and the second column of Table I, the most common rule is R1 (*Event-before-DOM*), accounting for 39.3% of cases (48 out of 122), where an event occurs before a DOM update. The second most common pattern is R3 (*DOM/Event-before-Response*), with 27.0% of cases (33 out of 122); R3 involves a DOM or event operation preceding a response, where timing or execution mismatches can cause interactions to occur before the page is fully adjusted. R2 (*DOM-before-Event*) accounts for 23.0% of cases, where a DOM update precedes an event trigger, creating risk if the element is not yet ready. R4 accounts for the remaining cases and is the least frequent at 10.7%, suggesting that most flaky behavior stems from complex DOM event interactions Overall, these results highlight that DOM event dependencies are particularly prone to subtle timing and ordering issues, confirming the value of rule-based analysis for uncovering common flakiness patterns.

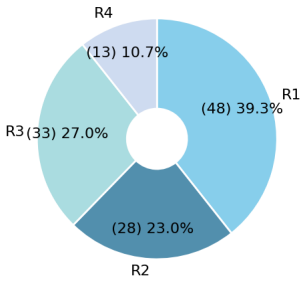


Fig. 6: Distribution of dependency rules (R1–R4) for DOM event analysis

TABLE I: Dataset. $\#_{FlakeyTests}$ and $\#_{Project}$ refer to the number of flaky tests and unique projects in each rule type. Flaky test size reports Lines Of Code (LOC), and mean_{proj} denotes the average flaky test size across projects.

Rule	$\#_{FlakeyTests}$	$\#_{Project}$	Flaky Test Size (*LOC*)		
			min	max	mean (mean_{proj})
R1	48 (39.3%)	23	4	32	15.44 (14.37)
R2	28 (23.0%)	18	6	28	16.00 (15.00)
R3	33 (27.0%)	21	6	21	13.70 (13.90)
R4	13 (10.7%)	13	5	27	14.23 (14.23)
All	122 (100.0%)	47	4	32	14.97 (14.37)

C. Ground-truth

To evaluate the effectiveness of our approach, we use developer-written fixes as the ground-truth for flaky locations. Specifically, we identify the original commit in each project that resolved the flaky test case and extract the corresponding code modifications. By comparing the predicted suspicious locations against developer-written fixes, we quantify the localization accuracy of *DoeFL* using standard Top-N ranking metrics introduced in the following section.

D. Metrics

We adopt the widely used Top-N metric (N = 1, 3, 5) to assess the localization effectiveness as shown in earlier studies [21], [37], [44]–[46]. The localization is considered successful by this metric if any of the actual flaky lines are present within the top N most suspicious locations identified by the approach. We focus primarily on Top-1 and Top-5. Top-1 represents the optimal situation, in which developer effort is minimized by prioritizing the more likely location of flakiness. The Top-5 offers a more practical balance by offering a limited set of candidates that significantly reduces the manual inspection; we include Top-3 as a supplement to further illustrate localization performance across varying thresholds.

E. Research Questions

RQ1: *How effective is the Lamport-based structural analysis in localizing DOM event-based flakiness issues?*

We first evaluate the structural analysis component of *DoeFL*, which leverages Lamport logical clocks to identify DOM elements and event sequences most likely responsible for flakiness across diverse web test failures; we call this version of *DoeFL* as $DoeFL_{woLLM}$. We use Top-N accuracy metrics to measure whether the actual flakiness-inducing locations appear within the top-ranked results, providing a quantitative view of localization effectiveness. To further understand the root causes of flakiness, i.e., the flaky DOM contexts arising from event sequences, we analyze how the DOM–event interaction styles (as defined by the four rules R1–R4 in (Section IV-A) relate to localization accuracy.

RQ2: *To what extent can LLMs enhance the effectiveness of flaky localization?*

In this question, we evaluate the complete *DoeFL*, which combines Lamport-based structural reasoning with LLM-based semantic analysis. We investigate how LLMs contribute to identifying subtle flaky patterns by leveraging contextual, intent-based information that pure structural reasoning may miss. To investigate the impact of model capability on localization results, we further compare the performance of *DoeFL* when using two different LLMs, (i.e., Llama3.2 and GPT4o).

RQ3: *What is the impact of fine-grained DOM event interaction types on flakiness localization?*

To answer this question, we move beyond the rule-level categories (R1–R4) examined in RQ1 and RQ2 and investigate localization performance at a finer granularity by examining the types of statements in the interaction (i.e., T_{SA}, T_{SE}, T_{SD}, T_{SDE}, T_{SDE}). While previous research has broadly categorized flaky tests, little is known about how specific DOM-event sequences—at the granularity of individual interactions between statements—affect localization success. By analyzing both the distribution of these interaction patterns and their correlation with Top-N accuracy, we aim to provide empirical insights into which fine-grained patterns are more effectively localized and which remain challenging.

RQ4: *How does the size of the test function lines affect the performance of DoeFL?*

To explore this question, we examine how the number of lines in a test file influences *DoeFL*'s ability to localize flakiness. Test code lines range from short scripts to longer tests with multiple DOM event interactions, each posing different challenges. Longer tests may introduce more sources of flakiness, making structural reasoning more complex. However, they also provide richer context for LLM-based semantic analysis. By evaluating *DoeFL* across varying test lengths, we aim to understand its consistency and assess the complementary benefits of combining structured analysis with LLM-driven reasoning.

V. Results

A. RQ1: Lamport Analysis for DOM Event-based Flakiness Localization

We evaluated the Lamport-based analysis component of *DoeFL* (referred to as $DoeFL_{woLLM}$) on 122 real-world flaky test cases to assess its baseline effectiveness in localizing DOM event-related flakiness. The results are categorized according to the four dependency rules (R1–R4), each representing a distinct pattern of DOM–event interaction (Table II). R1 (*Event-before-DOM*) achieves the highest localization performance, with a Top-1 rate of 50.0% and Top-3 and Top-5 rates of 79.2% and 85.4%. *DoeFL* achieves similar localization performance on R2 (*DOM-before-Event*), where flakiness occurs due to the event taking place before the DOM rendering, with Top-1 rate being 46.4% and Top-3 and Top-5 close to or over 80% (78.6% and 85.7%). R3 (*DOM/Event-before-Response*) follows, with Top-1 rates of 39.4% and both Top-3 and Top-5 rates of 78.8%. R4, which is the least common rule type in our dataset, shows the lowest performance in Top-3 and Top-5, despite having the highest value for Top-1 (61.5%).

This observation might be related to the size of the R4 category: R4 contains only 13 flaky test cases, making its Top-k statistics more sensitive to individual cases. Additionally, R4 primarily comprises remaining interaction patterns, including homogeneous and assertion-led sequences, which have fewer explicit DOM-event dependencies. As a result, such structural cues may suffice in cases with strong structural signals, but are less effective at maintaining high Top-k accuracy when structural signals are weaker. Combined, the Lamport-based analysis achieves an overall Top-1 accuracy of 47.5% and improves to 82.0% at Top-5, demonstrating that order and timing information alone can bring flaky locations close to the top ranking positions.

Answer to RQ1. The structural component of *DoeFL* achieved an overall Top-1 accuracy of 47.5% and Top-5 accuracy of 82.0% on 122 real-world flaky test cases. Among the four interaction rules (R1–R4), R1 (Event-before-DOM) showed the highest performance at Top-5: 85.4%, while R4 had the lowest at Top-5: 69.2%.

B. RQ2: LLM-Augmented Localization

To evaluate the impact of large language models (LLMs) on DOM event-related flakiness localization, we compare the baseline performance of *DoeFL* without LLM ($DoeFL_{woLLM}$) against two LLM-augmented configurations, Llama3.2 and GPT4o. As shown in Figure 7, both LLM models significantly improve Top-1, Top-3, and Top-5 accuracy, with GPT4o yielding higher gains from using LLMs. When integrated with Llama3.2, *DoeFL* achieves 65.6% Top-1 accuracy and 89.3% Top-5 accuracy. This suggests that LLMs can complement dependency-based structural analysis by capturing contextual or behavioral cues. GPT4o further improves the localization, reaching 77.9% Top-1 and 95.1% Top-5 accuracy, highlighting the impact of model capability on localization performance.

TABLE II: Effectiveness of *DoeFL* with only the structural analysis component ($DoeFL_{woLLM}$) in localizing flakiness across dependency rules (R1–R4). $\#_{FlakTst}$ refers to the number of flaky tests in each rule group, and Top-N (N=1,3,5) rankings are reported.

Rule	$\#_{FlakTst}$	Top1	Top3	Top5
R1	48	24 (50.0%)	38 (79.2%)	41 (85.4%)
R2	28	13 (46.4%)	22 (78.6%)	24 (85.7%)
R3	33	13 (39.4%)	26 (78.8%)	26 (78.8%)
R4	13	8 (61.5%)	9 (69.2%)	9 (69.2%)
All	122	58 (47.5%)	95 (77.9%)	100 (82.0%)

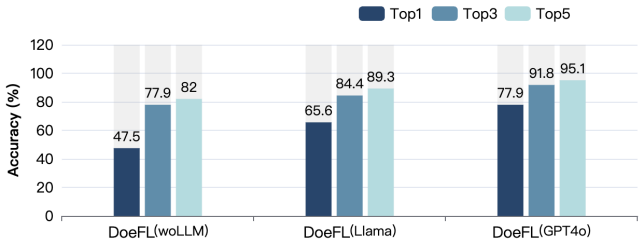


Fig. 7: Comparing the localization of *DoeFL* using Llama and GPT4o.

Overall, these results indicate that incorporating LLM-based reasoning helps improve the effectiveness of Lamport-based structural analysis by leveraging contextual and intent-level information. The results further suggest that localization performance varies with LLM capability, as shown by the additional gains observed when using a stronger model.

TABLE III: Compare *DoeFL* with Llama3.2 and GPT4o's accuracy across DOM event rules (R1–R4) and Top-N metrics

Rule ($\#_{FlakTst}$)	LLM	Top1	Top3	Top5
R1 (48)	Llama3.2	36 (75%)	42 (87.5%)	45 (93.8%)
	GPT4o	42 (87.5%)	45 (93.8%)	46 (95.8%)
R2 (28)	Llama3.2	17 (60.7%)	23 (82.1%)	24 (85.7%)
	GPT4o	19 (67.9%)	26 (92.9%)	27 (96.4%)
R3 (33)	Llama3.2	19 (57.6%)	28 (84.8%)	29 (87.9%)
	GPT4o	26 (78.8%)	30 (90.9%)	31 (93.9%)
R4 (13)	Llama3.2	8 (61.5%)	10 (76.9%)	11 (84.6%)
	GPT4o	8 (61.5%)	11 (84.6%)	12 (92.3%)
All (122)	Llama3.2	80 (65.6%)	103 (84.4%)	109 (89.3%)
	GPT4o	95 (77.9%)	112 (91.8%)	116 (95.1%)

Table III provides a detailed comparison of *DoeFL* integrated with Llama3.2 and GPT4o across the four dependency

rule categories (R1–R4). Overall, both models show great improvements in Top-1 to Top-5 accuracy, except for R4. In R4, neither model improves the Top-1 rate; although Top-3 and Top-5 rates improve slightly, the gains are minor compared to the other rules. This suggests that loosely structured or repetitive single-type DOM or event operations offer limited context for effective reasoning. For R1–R3, both Llama3.2 and GPT4o show the greatest improvement in terms of Top-1 accuracy for the R1 group. With Llama3.2, R1's Top-1 rate increases from 50.0% to 75.0%, representing a 50% relative improvement. For GPT4o, the pattern is similar, but in R3 the model achieves a 100% relative improvement (from 39.4% to 78.8%). Nevertheless, *DoeFL* still achieves its best overall performance in R1, benefiting from the semantic reasoning provided by the LLM-based approach. For Top-3 and Top-5 rankings, more than 80% and even over 90% of flaky tests are localized within the top three or five statements, demonstrating the combined effectiveness of Lamport-based structural analysis and LLM-based semantic reasoning.

Across all rule types and Top-N metrics, GPT4o consistently outperforms Llama3.2, confirming that LLM capacity significantly influences *DoeFL*'s localization accuracy. While some rule types, such as R4, remain challenging, LLM integration clearly enhances contextual reasoning and ranking performance.

Answer to RQ2. With LLM integration, *DoeFL*'s localization accuracy improved substantially. Across R1–R4, the largest Top-1 gains often occurred in R1 (50.0% to 75.0% with Llama3.2 and 87.5% with GPT4o), while R3 saw a 100% relative improvement under GPT4o (39.4% to 78.8%). Relative improvements for Top-3 and Top-5 were moderate, still in total 80% and 90% of flaky tests localized within the top three or five statements. R4 remained challenging due to limited contextual cues. Overall, GPT4o consistently outperformed Llama3.2, confirming that LLM capacity strongly influences localization accuracy.

C. RQ3: The impact of fine-grained DOM event interaction types on localization

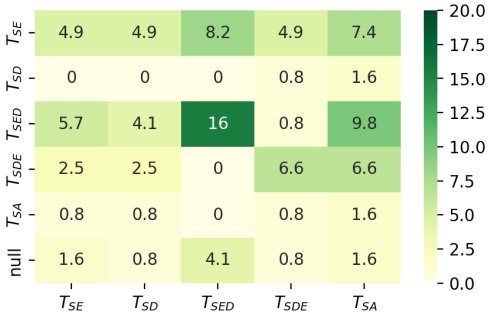


Fig. 8: Heatmap of detailed interaction behind DOM event dependency rules. Each cell contains the frequency value in percentage. Two consecutive statement types are indicated by *y-axis* for the former and *x-axis* for the latter.

Figure 8 presents a heatmap illustrating the frequency of sequence patterns across DOM event dependency rule types, providing a detailed view of how specific statement sequences relate to flakiness. The percentage of flaky test cases for each consecutive statement pair is shown in the heatmap cells, where both axes represent the interaction categories of T_{SE}, T_{SD}, T_{SA}, T_{SED}, and T_{SDE}. The statement sequence T_{SED}-T_{SED}, which belongs to R1, is the most prevalent, accounting for 16% of the studied flaky cases. This prevalence suggests that consecutive event-before-DOM interactions are particularly prone to flakiness due to compounded timing sensitivity and asynchronous DOM updates. Other recurring patterns include T_{SED}-T_{SA} (9.8%) and T_{SE}-T_{SED} (8.2%), reinforcing the importance of event-driven DOM updates and their cascading effects in dynamic web environments. In addition, event-first sequences (Y-axis on T_{SE}) tend to be less stable, while interactions involving DOM-rendering before events, such as T_{SDE}-T_{SDE} (6.6%), also show flakiness, especially when chained DOM–event interactions are present.

Overall, the distribution highlights that flakiness often stems from event-triggered DOM changes, especially those with chained or repetitive dependencies. These findings highlight the importance of tools like *DoeFL* in identifying and prioritizing high-risk interaction patterns. Furthermore, the presence of non-zero values across all cells suggests that comprehensive coverage is necessary to effectively handle diverse flaky behaviors in web tests.

TABLE IV: The top 5 localization of DOM event rule-based patterns for $DoeFL_{woLLM}$ and LLM-based approach ($DoeFL_{Llama}$ and $DoeFL_{GPT4o}$)

Type	Rules	Details	Top-5	Total
$DoeFL_{woLLM}$	R1	T_{SED}-T_{SED}	17	25
		T_{SE}-T_{SED}	8	
	R2	T_{SDE}-T_{SD}	7	7
	R3	T_{SED}-T_{SA}	11	25
		T_{SE}-T_{SA}	7	
		T_{SDE}-T_{SA}	7	
$DoeFL_{Llama}$	R1	T_{SED}-T_{SED}	19	27
		T_{SE}-T_{SED}	8	
	R2	T_{SDE}-T_{SD}	7	7
	R3	T_{SED}-T_{SA}	11	27
		T_{SE}-T_{SA}	9	
		T_{SDE}-T_{SA}	7	
$DoeFL_{GPT4o}$	R1	T_{SED}-T_{SED}	19	28
		T_{SE}-T_{SED}	9	
	R2	T_{SDE}-T_{SD}	8	8
	R3	T_{SED}-T_{SA}	12	29
		T_{SE}-T_{SA}	9	
		T_{SDE}-T_{SA}	8	

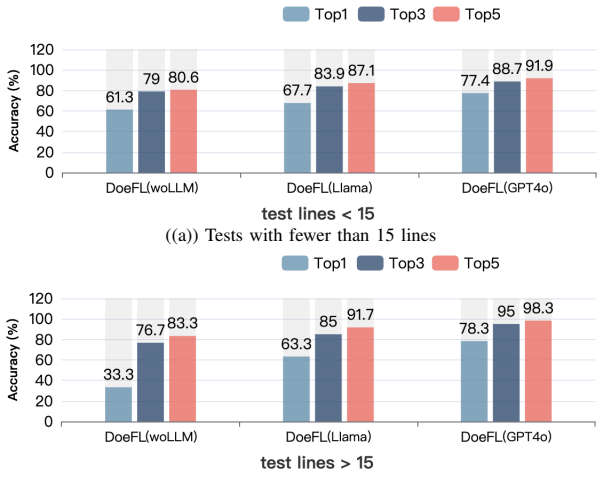
Table IV presents the Top-5 accuracy metrics for the best-performing cases based on finer-granularity interactions defined by consecutive statement types: T_{SED}-T_{SED}, T_{SE}-T_{SED}, T_{SDE}-T_{SD}, T_{SED}-T_{SA}, T_{SE}-T_{SA}, and T_{SDE}-T_{SA}. The last two combinations, T_{SE}-T_{SA} and T_{SDE}-T_{SA}, share the same Top-5 accuracy and are therefore both included, resulting in six finer-granularity cases for analysis.

The combination most effectively localized by *DoeFL* is T_{SED}-T_{SED} under R1, which achieves high detection counts across all three configurations: 17 ($DoeFL_{woLLM}$), 19 ($DoeFL_{Llama}$), and 19 ($DoeFL_{GPT4o}$). This suggests that flaky

behavior from consecutive event-before-DOM interactions is not only frequent but also easier to localize due to its well-defined execution flow and clear causal ordering. Similarly, T_{SED}-T_{SA} demonstrates strong localization results within R3, with case counts increasing from 11 to 12 across models. The consistent performance of T_{SED}-T_{SED} and T_{SED}-T_{SA} indicates that both rule-based and LLM-augmented analysis effectively handle flakiness associated with tightly coupled, sequential event–DOM chains interactions. These findings suggest that interaction combinations with evident structural dependencies—particularly those involving event-driven DOM updates with direct sequencing (e.g., T_{SED}-T_{SED}, T_{SE}-T_{SED}, T_{SED}-T_{SA})—are relatively simpler to localize. Such patterns provide strong temporal and contextual signals, enabling both structural and LLM-enhanced systems to consistently rank the root cause within the top five.

Answer to RQ3. Our results indicate that flakiness is easier to localize in DOM event combinations that exhibit clear, sequential patterns—particularly T_{SED}-T_{SED} and T_{SE}-T_{SED} under R1, T_{SDE}-T_{SD} under R2, and T_{SED}-T_{SA} under R3. These sequences achieve a higher Top-5 localization rate by providing robust contextual signals that benefit both structural analysis and LLM-based reasoning.

D. RQ4: The impact of test length on localization accuracy



((a)) Tests with fewer than 15 lines

((b)) Tests with 15 or more lines

Fig. 9: Localization accuracy across various test lengths

To examine the influence of test function size on localization performance, we partitioned the flaky cases into two balanced groups based on line count (flakiness-related code snippet): tests with fewer than 15 lines (62 cases) and those with 15 or more lines (60 cases). This threshold reflects the overall distribution of test lengths, which range from 4 to 32 lines. As shown in Figure 9, the performance of *DoeFL* (GPT4o) improves with longer tests. Top-1 accuracy increases from 77.4% to 78.3%, and Top-5 accuracy from 91.9% to 98.3%. This pattern indicates that longer test cases with richer context offer more semantic cues, which improves GPT4o's ability to infer intent and causality. In contrast, *DoeFL* (Llama) sees a slight drop in Top-1 accuracy, from 67.7% to 63.3%. This implies that *DoeFL* with Llama may have some limitations in handling the increased complexity of longer DOM event sequences, suggesting that model capacity and depth of contextual understanding are important factors in scalability.

The *DoeFL*$_{woLLM}$ baseline approach exhibits a more visible decline, with Top-1 accuracy decreasing from 61.3% to 33.3% as the test length increases. This highlights the difficulty of isolating flaky behavior through structural rules alone in longer, noisier event sequences. Overall, the findings emphasize the benefits of integrating structural analysis with LLM-based reasoning. While all LLMs offer improvements, more advanced models like GPT4o show greater robustness and adaptability in handling longer, more complex tests.

Answer to RQ4. The findings show that the accuracy of localization can be impacted by the size of the test file. *DoeFL* (GPT4o) improves localization even when faced with longer tests, demonstrating the advantages of an LLM-based approach that capitalizes on richer context and longer event traces. Furthermore, a slight decline in *DoeFL* (Llama) implies the influence of model capacity. In contrast, the traditional-based *DoeFL*$_{woLLM}$ performs worse as the test size increases, highlighting the limitations of purely structural analysis in larger, more intricate scenarios.

VI. Discussion

Practical support for developers in debugging DOM-event related flakiness. A key strength of *DoeFL* is its ability to assist developers in debugging DOM-event-related flakiness. By combining structural analysis of DOM events with LLM-based semantic reasoning, the approach effectively narrows down likely sources of flakiness and explains unusual DOM event interactions. This enables developers to interpret lengthy and complicated execution traces without the need to manually examine each statement. When dealing with asynchronous behavior or subtle timing problems, the ranked list of suspicious locations and annotated DOM event pairs provides a good starting point for investigation. Even though developer input is still needed, the approach fits well with the actual program workflow, i.e., how UI interactions work, which makes the process efficient and relatively lightweight.

Broaden the scope to general asynchronous and order-sensitive flakiness. While our approach is specifically designed to address web-based DOM event flakiness, its fundamental principles, including the modeling of structured DOM event sequences and LLM-based semantic reasoning, are applicable to various contexts that involve asynchronous behavior and ordering issues. Many modern flaky failures stem from implicit timing assumptions, such as race conditions, delayed responses, or loosely coupled logical handlers. By using Lamport logical clocks to infer partial execution order and leveraging LLMs to reason about intended execution flows, our approach can help detect order-sensitive bugs beyond the DOM layer. For example, it can support debugging situations that involve asynchronous test steps, callback sequencing, or client-server

synchronization. While adjustments would be needed to define non-DOM interactions, the combination of structural analysis and context-aware reasoning remains broadly applicable to a wide range of asynchronous debugging challenges.

VII. Threats to Validity

Construct validity threats. A potential threat to construct validity lies in whether the captured DOM event sequences fully represent all factors contributing to flakiness. The trace data may not reflect external influences, such as network latency or environment-specific behaviors, which could result in the exclusion of these causes. In addition, our dependence on LLMs is predicated on the assumption that the models are trained on datasets that are representative of real-world test behaviors. The model's predictions may overlook context-specific flakiness patterns if the training data is not diverse or covered, which would restrict generalizability.

Internal validity threats. A potential threat to internal validity lies in the accuracy of the LLM's reasoning about the main causes of flakiness. While the model may detect correlations between specific DOM event sequences and flaky behavior, these patterns do not entirely imply the causal dependencies. Hidden factors, including external service dependencies or asynchronous timing issues, may also contribute to flakiness. Additionally, non-deterministic behaviors in web applications, including subtle timing variations or user-driven interactions, may lead to inconsistent outcomes.

External validity threats. The external validity is primarily threatened by the representativeness of the test cases that were evaluated. Our study focuses on DOM-related flakiness in web applications, using a set of real-world flaky tests. However, the results may not be applicable to all types of web tests, particularly those that involve complex third-party integrations, non-DOM-related failures (e.g., server-side flakiness), or highly complex dynamic content. Moreover, the LLMs employed in our methodology may exhibit varying performance across languages, frameworks, or test styles that are not included in our dataset. As a result, our findings might not be as applicable to projects with different architectures and testing frameworks or to more general web testing scenarios.

VIII. Related work

Flaky tests have been widely studied due to their impact on software reliability and developer productivity [1], [47]–[50]. Prior studies have characterized flakiness in various contexts, ranging from backend logic to UI and integration testing, and developed tools to detect, classify, and mitigate flakiness. However, most studies treat flakiness as a statistical or code-level issue, often overlooking the dynamic and asynchronous nature of web-based interactions. Romano et al. conducted an empirical study of flaky web tests by analyzing flaky UI tests, identifying common causes across web projects [42]. Pei et al. highlighted the critical role of DOM-event sequences in web test flakiness via an in-depth investigation of DOM event interactions [28]. While these studies show how subtle differences in event ordering or rendering delays can trigger flakiness, they do not provide systematic methods for localizing flakiness within complex test workflows.

In the domain of fault localization, several approaches have been proposed to help developers diagnose the root cause of test failures, including spectrum-based fault localization (SBFL) and statistical debugging [17]. However, these techniques often struggle with flaky tests because they assume a deterministic behavior, which makes it difficult to handle failures that are not consistently reproducible. Flaky tests detection and localization have been the subject of research in unit and integration testing. Tools such as DeFlaker [51], iDFlakies [52], Flakify [53], iFixFlakies [54], and so on [55], [56] have been employed to identify non-deterministic patterns in test outcomes. Nevertheless, these methods frequently do not account for the dynamics of web applications, which are frequently the result of asynchronous UI interactions and DOM-state dependencies, which contribute to flakiness. Additionally, the structure and evolution of the Document Object Model (DOM) have been demonstrated to significantly impact test stability, particularly in event-driven interfaces. Empirical investigations have established a correlation between high rates of nondeterminism in test outcomes and unpredictable DOM event sequences, such as those set off by user interaction simulations or rendering delays [9], [28]–[30]. Nevertheless, the majority of current research examines DOM behavior in a static or isolated manner, failing to investigate the sequential nature of event propagation and mutation.

More recently, large language models (LLMs) have shown promise in supporting fault localization and repair. Tools such as AgentFL [37] use LLMs to rank suspicious lines or suggest fixes based on code and logs, and tools like Testgeneval [57] demonstrate the ability of LLMs to generate test code. Some work has explored LLMs in flaky contexts, but their application to web-specific flakiness—particularly DOM-event sequence reasoning—remains limited [58].

Our work fills this gap by combining DOM event structure analysis with LLM-based reasoning. Unlike prior efforts that treat DOM issues statically or generically, we explicitly model the causality between DOM events using Lamport clocks and supplement this with LLM-based interpretation of semantic intent. This hybrid approach makes it possible to locate DOM-induced flakiness more accurately and interpretably.

IX. Conclusion

This paper introduces *DoeFL*, a hybrid approach that combines structural DOM-event modeling with LLM-based semantic reasoning to identify flakiness in web tests. By analyzing DOM event sequences using Lamport logical clocks and interpreting context through LLM-driven intent reasoning, *DoeFL* localizes likely sources of flaky behavior and ranks them to aid debugging. Our evaluation of 122 real-world flaky test cases shows that the Lamport-based structural component alone can achieve 47.5% Top-1 and 82% Top-5 accuracy, while integrating advanced LLMs such as GPT4o boosts performance to 77.9% (Top-1) and 95.1% (Top-5). Detailed analysis of DOM-event sequences shows that patterns

like T_{SED}-T_{SED}, T_{SE}-T_{SED}, T_{SDE}-T_{SDE}, T_{SED}-T_{SA}, are easier to localize. *DoeFL* also performs well on longer, complex tests that challenge traditional rule-based methods. Despite limitations such as reliance on LLMs and limited coverage of non-DOM factors, *DoeFL* provides a practical and extensible foundation for debugging flakiness in dynamic web environments. To support reproducibility and future research on flakiness localization, we release our data and replication package at https://zenodo.org/records/17804388.

X. Acknowledgments

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C20/IS/14761415/TestFlakes and partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2021-NR060080).

References

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering (FSE)*, 2014, pp. 643–653.

[2] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, "A replication study on the usability of code vocabulary in predicting flaky tests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 219–229.

[3] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–74, 2021.

[4] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, M. Cordy, and Y. Le Traon, "What made this test flake? pinpointing classes responsible for test flakiness," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 352–363.

[5] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root Causing Flaky Tests in a Large-Scale Industrial Setting," in *Proceedings ofthe 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019, pp. 101–111.

[6] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 244–255.

[7] M. Cordy, R. Rwemalika, A. Franci, M. Papadakis, and M. Harman, "Flakime: Laboratory-controlled test flakiness impact assessment," in *2021 IEEE/ACM 44rd International Conference on Software Engineering (ICSE)*. IEEE, 2022.

[8] Y. Pei, S. Habchi, R. Rwemalika, J. Sohn, and M. Papadakis, "An empirical study of async wait flakiness in front-end testing." in *BENEVOL*, 2022.

[9] 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 and Methodology (TOSEM)*, 2024.

[10] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[11] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 265–274.

[12] T. Li, C. Cui, L. Ma, D. Towey, Y. Xie, and R. Huang, "Leveraging large language models for automated web-form-test generation: An empirical study," *arXiv preprint arXiv:2405.09965*, 2024.

[13] A. Alameer, S. Mahajan, and W. G. Halfond, "Detecting and localizing internationalization presentation failures in web applications," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 202–212.

[14] Y. Zou, Z. Chen, Y. Zheng, X. Zhang, and Z. Gao, "Virtual DOM coverage for effective testing of dynamic web applications," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov, Eds. ACM, 2014, pp. 60–70. [Online]. Available: https://doi.org/10.1145/2610384.2610399

[15] S. Mahajan, K. B. Gadde, A. Pasala, and W. G. Halfond, "Detecting and localizing visual inconsistencies in web applications," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2016, pp. 361–364.

[16] R. Rwemalika, M. Kintis, M. Papadakis, Y. L. Traon, and P. Lorrach, "On the evolution of keyword-driven test suites," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 2019, pp. 335–345. [Online]. Available: https://doi.org/10.1109/ICST.2019.00040

[17] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[18] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[19] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 165–176.

[20] J. Sohn, Y. Kamei, S. McIntosh, and S. Yoo, "Leveraging fault localisation to enhance defect prediction," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 284–294.

[21] J. Sohn and M. Papadakis, "Cement: On the use of evolutionary coupling between tests and code units. a case study on fault localization," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 133–144.

[22] K. Barbosa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda, "Test flakiness across programming languages," *IEEE Transactions on Software Engineering (TSE)*, vol. 49, no. 4, pp. 2039–2052, 2022.

[23] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of llm-based explainable fault localization," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1424–1446, 2024.

[24] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[25] J. Morán, C. Augusto, A. Bertolino, C. De La Riva, and J. Tuya, "Flakyloc: flakiness localization for reliable test suites in web applications," *Journal of Web Engineering*, vol. 19, no. 2, pp. 267–296, 2020.

[26] B. Vancsics, T. Gergely, and A. Beszedes, "Simulating the effect of test flakiness on fault localization effectiveness," in *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. IEEE, 2020, pp. 28–35.

[27] J. Liu, J. Hao, C. Zhang, and Z. Hu, "Wepo: Web element preference optimization for llm-based web navigation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 25, 2025, pp. 26 614–26 622.

[28] Y. Pei, J. Sohn, and M. Papadakis, "An empirical study of web flaky tests: Understanding and unveiling dom event interaction challenges," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2025, pp. 92–102.

[29] M. Mirzaaghaei and A. Mesbah, "Dom-based test adequacy criteria for web applications," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 71–81.

[30] X. Liu, Z. Song, W. Fang, W. Yang, and W. Wang, "Wefix: Intelligent automatic generation of explicit waits for efficient web end-to-end flaky tests," in *Proceedings of the ACM on Web Conference (WWW)*, 2024, pp. 3043–3052.

[31] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[32] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 251–262, 2012.

[33] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models

are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[34] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[35] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[36] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[37] Y. Qin, S. Wang, Y. Lou, J. Dong, K. Wang, X. Li, and X. Mao, "Agentfl: Scaling llm-based fault localization to project-level context," *arXiv preprint arXiv:2403.16362*, 2024.

[38] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large language models in fault localisation," *arXiv preprint arXiv:2308.15276*, 2023.

[39] M. Nass, E. Alégroth, and R. Feldt, "Improving web element localization by using a large language model," *Software Testing, Verification and Reliability*, vol. 34, no. 7, p. e1893, 2024.

[40] S. Rahman, A. Baz, S. Misailovic, and A. Shi, "Quantizing large-language models for predicting flaky tests," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2024, pp. 93–104.

[41] 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*, 2024.

[42] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of ui-based flaky tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1585–1597.

[43] G. Haben, S. Habchi, J. Micco, M. Harman, M. Papadakis, M. Cordy, and Y. Le Traon, "The importance of accounting for execution failures when predicting test flakiness," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1979–1989.

[44] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 273–283.

[45] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 169–180.

[46] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 664–676.

[47] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1471–1482.

[48] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 403–413.

[49] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, 2020, pp. 492–502.

[50] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*, 2020, pp. 211–224.

[51] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the 40th international conference on software engineering (ICSE)*, 2018, pp. 433–444.

[52] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "IDFlakies: A framework for detecting and partially classifying flaky tests," *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 312–322, 2019.

[53] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," *IEEE Transactions on Software Engineering (TSE)*, vol. 49, no. 4, pp. 1912–1927, 2022.

[54] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019, pp. 545–555.

[55] S. Dutta, A. Shi, and S. Misailovic, "Flex: fixing flaky tests in machine learning projects by updating assertion bounds," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 603–614.

[56] X. Cai, Z. Dong, Y. Wang, A. Tiwari, and X. Peng, "Reproducing timing-dependent gui flaky tests in android apps via a single event delay," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1504–1515.

[57] K. Jain, G. Synnaeve, and B. Rozière, "Testgeneval: A real world unit test generation and test completion benchmark," *arXiv preprint arXiv:2410.00752*, 2024.

[58] C. Lee, C. S. Xia, L. Yang, J.-t. Huang, Z. Zhu, L. Zhang, and M. R. Lyu, "A unified debugging approach via llm-based multi-agent synergy," *arXiv preprint arXiv:2404.17153*, 2024.