© AUG 2026 | IRE Journals | Volume 10 Issue 2 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV10I2-1722278

# Displaced and Suppressed: A Root-Cause Taxonomy of Test Failures in a Large Industrial Playwright/BDD Suite

SONIYA V
*Quality Assurance Engineer*

***Abstract-*** ***End-to-end (E2E) user-interface test suites at industrial scale are widely reported to make flakiness their dominant maintenance cost. Less widely examined is a second problem that compounds it: the site at which a failure is reported often diverges from the site of its true defect, and a further, under-examined subset of defects suppresses the failure signal entirely rather than merely displacing it. This paper reports an industrial case study of a 48-feature, approximately 852-scenario Playwright and Cucumber behaviour-driven test suite exercising a commercial multi-tenant software-as-a-service platform across three environments. Through practitioner-embedded root-cause analysis, latency instrumentation of the application under test, and codebase-wide pattern search, a taxonomy of seven recurring defect classes is derived and organized along a displacement/suppression axis. Five classes cause a correct expectation about test behaviour to fail at a location distant from the true cause; two classes cause a broken test to report a pass while verifying nothing. Measured latencies that motivate specific fixes are reported, the propagation of one defect class across eighteen page-object classes is quantified, and a dependency-partitioned parallel execution architecture adopted as a mitigating intervention for suite run time is described. The taxonomy is positioned against the flaky-test and test-smell literature, noting where the suppression classes are structurally analogous to previously reported "rotten green test" phenomena in unit testing, and the threats to validity that follow from a single-suite, single-team study are set out.***

***Index Terms-*** ***Behaviour-Driven Development, End-To-End Testing, Flaky Tests, Silent Test Failure, Test Smells.***

## I. INTRODUCTION

Automated end-to-end (E2E) testing of web applications through the browser is now standard practice for verifying user-facing behaviour that unit and integration tests cannot reach. It is also a well-documented source of maintenance cost: tests that pass and fail intermittently against an unchanged code base — flaky tests — have been the subject of a substantial empirical literature since Luo et al.'s foundational study of flaky-test fixes in open-source projects [1]. That literature, and the tooling built on it, is organized around an implicit assumption: that when a test fails, the failure is at least a usable signal, and the practitioner's task is to trace it back to a true cause that may be several architectural layers away.

This paper is concerned with two problems that sit either side of that assumption. The first, displacement, is well represented in the literature: a test fails, correctly, but the reported failure site — a timeout on a particular locator, an assertion on a particular field — is architecturally distant from the defect that caused it. The second, suppression, is comparatively under-examined in end-to-end and behaviour-driven-development (BDD) contexts specifically: a test that has been broken by a defect nonetheless reports a pass, because the step or assertion that would have caught the defect never meaningfully executed. Suppression is the more dangerous of the two, because it removes a coverage signal invisibly; by construction it cannot be found by reading the test report that it corrupts.

Both problems are studied here in a single artifact: an industrial Playwright and Cucumber test suite of 48 feature files and approximately 852 scenarios, automating a commercial multi-tenant recruitment SaaS platform across three environments. Over the course of routine maintenance of this suite, seven recurring defect classes were diagnosed, each with an identified mechanism, a validated fix, and, for several classes, a measured latency or blast radius. The contribution of this paper is threefold: (1) a seven-class taxonomy of E2E/BDD test defects organized along a displacement/suppression axis; (2) a case study of suppression arising structurally from BDD step-definition architecture rather than from an isolated coding mistake; and (3) a dependency-

partitioned parallel execution architecture adopted to reduce suite run time, presented as a design contribution with its timing evaluation identified as ongoing work.

## II. RELATED WORK

### A. Flaky-Test Taxonomies

Luo et al. manually analyzed 201 commits that fixed flaky tests across 51 Apache projects and produced the first widely used taxonomy of flaky-test root causes, finding asynchronous waits, concurrency, and test-order dependency to be the dominant categories [1]. Eck et al. extended this with a developer-centred study of 200 flaky tests drawn from Mozilla's issue tracker [2]. Zhang et al. specifically revisited the assumption of test independence and characterized order-dependent failures as a distinct phenomenon [3]. Gruber et al. replicated this line of work in Python and found order dependency, rather than asynchronous waiting, responsible for 59% of flaky tests in their sample, arguing that root-cause prevalence is ecosystem-dependent [4]. Industrial replications reinforce this variability: Lam et al. reported on root-causing flaky tests inside Microsoft's own infrastructure [5], and a recent study of SAP HANA's fixed flakiness issue reports found concurrency to be the leading category in that system [9]. The suite examined here adds a data point at a scale unusual for E2E/BDD-specific flakiness studies, in a stack (Playwright driving a Cucumber step-definition layer) that differs from the unit- and integration-test corpora on which most existing taxonomies were built.

### B. Test Smells and Assertion Quality

A parallel literature, originating with van Deursen et al.'s catalogue of test smells [6], treats poor test design as a maintainability and reliability problem distinct from non-determinism. Assertion Roulette — a test with multiple unexplained assertions such that a failure does not indicate which one failed — is among the most studied smells in this catalogue [10]. Several mechanisms reported in Section IV (particularly C3 and C4) are best understood as this literature's fixture- and assertion-quality concerns manifesting in a browser-driven, multi-step wizard context.

### C. Silent False Passes

The suppression side of this taxonomy is closest to what Delplanque et al. term Rotten Green Tests: tests that report as passing but contain at least one assertion that is never executed — "a case worse than a broken test", since it falsely certifies validity rather than merely failing to detect a defect [7]. Delplanque et al. detected 294 such tests among roughly 19,900 in the Pharo ecosystem; Martinez et al. built a comparable framework for Java and found 427 rotten tests across 26 open-source projects [8]. That body of work is confined to unit tests written against xUnit-style frameworks. The C6 and C7 classes reported here are a distinct manifestation of the same underlying failure mode — a test that passes without having verified anything — arising instead from step-definition-scoped state loss and misdirected preconditions in a Cucumber/Gherkin architecture. No prior work connecting the rotten-green-test phenomenon specifically to BDD step-definition lifecycle was found in the course of preparing this paper; this is treated as an open question for peer review rather than an established claim, since the underlying literature search was not exhaustive.

### D. Parallel and Dependency-Aware Test Execution

Reducing E2E suite run time through parallel execution is common practice but is complicated whenever scenarios are not independent — for instance when a later scenario consumes a record created by an earlier one, itself a documented anti-pattern in BDD test design [3]. Dependency- and similarity-aware scheduling has been studied for manual and integration test suites, notably by Landin et al., who cluster test cases by semantic similarity to construct parallel schedules for a Telecom use case [11]. The intervention in Section VI applies the same underlying idea to an automated, BDD-structured E2E suite.

## III. STUDY SETTING AND METHOD

The artifact under study is a Playwright and Cucumber (BDD, TypeScript) test suite automating a commercial multi-tenant recruitment SaaS platform: 48 feature files, approximately 852 scenarios, 116 TypeScript source files, 18 page-object classes, 48 step-definition files, 5 CI workflows, and 3 target

environments (development, pre-production, QA), organized into smoke, sanity, and regression tiers by Gherkin tag. Following the platform's confidentiality requirements, the product name, hostnames, and tenant identifiers are withheld throughout; environments are referred to generically and any account-level detail is referred to by role only. Framework-level detail — the behaviour of a searchable-dropdown component, the Cucumber step lifecycle, Playwright's auto-waiting semantics — is reported precisely, since it describes the testing framework rather than client business data.

The taxonomy in Section IV was derived through practitioner-embedded root-cause analysis conducted during ordinary maintenance of the suite, not a systematic mining study of a fixed historical corpus. The triage sequence for each investigated failure was: reproduce in isolation; instrument the relevant step to distinguish an application defect from a test-side defect; identify the mechanism by which the test's expectation and the application's actual behaviour diverged; implement and verify a fix; and, where the mechanism appeared reusable, search the remaining codebase for the same pattern. Each diagnosed defect was then classified onto the displacement/suppression axis according to a single criterion: whether the test, once broken, reported a failure (displacement) or a pass (suppression). This classification was performed by the engineer who diagnosed the defects; independent re-classification to report inter-rater agreement is identified in Section VII as outstanding work.

## IV. A SEVEN-CLASS TAXONOMY OF TEST DEFECTS

Each class is presented with a uniform structure — mechanism, why it misleads, and fix pattern — so the set reads as a taxonomy rather than a sequence of anecdotes.

### A. C1 — Asynchronous List Placeholder Selected Instead of the Real Option

The application's searchable dropdown renders a placeholder reading “Loading…” while its option list is fetched. Instrumented measurement showed the placeholder still displayed at t = 1500 ms, with the real option list appearing only at t = 2500 ms. The prevailing pattern was a fixed 1000 ms wait followed by a click on the first positional option, which lands on the placeholder. The click leaves the selection silently unset; every downstream assertion depending on it fails several steps later, non-deterministically, and reads as an unrelated slow dependency. The fix is to wait for the option whose text matches the intended value, falling back to the first option not containing the placeholder string, and to log which option text was actually clicked. This pattern was found, by codebase-wide search, in approximately 16 further page objects beyond the one first diagnosed — spanning reports, account creation and editing, job orders, candidates, tearsheets, and bulk upload — demonstrating a defect class propagating by copy-paste across a codebase.

### B. C2 — Assertion Window Narrower Than the Response Latency

A record-update request was measured taking 13.4 seconds to return HTTP 200 on the pre-production environment. The success toast appeared at t + 13.5 s and disappeared by t + 19.3 s — about 6 seconds of visibility. The step's wait budget was 10 seconds, which expired before the toast rendered, so the step read an empty string. An empty toast reads identically to a genuine save failure; here the save had, in fact, succeeded. The fix is to size the wait budget against measured API latency (about 30 seconds here) and to read the toast text from the handle already returned by the wait, since the element can vanish before a second lookup. Verifying measured API timing before concluding that a save silently failed is the operative lesson.

### C. C3 — Form State Toggled Back Off

Consent-style checkboxes were clicked unconditionally. Where already checked, the click toggled the control off, leaving the form invalid and the submit button disabled. The reported failure is a click timeout on the Save button — entirely downstream of the real defect, a required input elsewhere on the form. The fix is to check presence, then read the checked state, then click only if unchecked.

D. C4 — Premature Commit in a Multi-Step Wizard
The record-creation wizard's tab set varies with the selected parent record; only the final step, identifiable as the one with no "Continue" control, persists. Clicking Save on an intermediate step validates and advances without persisting. Every fill step reports a pass; nothing is created; the failure surfaces later at list verification, reading as a listing defect. The fix walks Continue by accessible role and label until it disappears, then saves.

E. C5 — Environment-Divergent Locators
On the same wizard, one environment's footer buttons carried predictable sequential identifiers while another environment's final Save button did not, so an identifier-prefix locator passed on one environment and timed out on the other — reading as environment instability rather than a locator-strategy defect. Preferring accessible role and visible label over generated identifiers resolved this across both environments, and is offered as an argument for accessibility-first locators on reliability grounds rather than the more commonly cited accessibility-compliance grounds.

F. C6 — Cross-Step State Loss Producing a Silent False Pass
Nearly every step definition constructs a fresh page-object instance at the top of the step, so instance fields do not persist between Gherkin steps. A verification step reading such a field observes its default (typically an empty string), and an empty-string locator query can still resolve and match — producing a false pass. Classes C1–C5 make a correct expectation fail in a confusing place; C6 makes a broken expectation look healthy, removing the coverage signal entirely without any visible symptom. The fix carries values across steps explicitly — as Gherkin parameters where possible, otherwise on the shared World object or a module-scoped variable — never on page-object instance fields.

G. C7 — Guard on the Wrong Context
A bulk-upload step was guarded on a URL belonging to a different screen; the guard never matched, so the step silently did nothing and reported a pass. The structural cause differs from C6 — the assertion is never reached at all, rather than reached against stale state — but the observable outcome (a green step that tested nothing) is the same. A guard that fails to match its intended context must fail loudly or log, never no-op silently.

H. The Organizing Axis
Displacement (C1–C5) is defined as a test failing at a site architecturally distant from the true cause, at the cost of diagnosis time and misattributed product defects. Suppression (C6, C7) is defined as a test passing while verifying nothing, at the cost of a silent, undetectable loss of coverage. The flaky-test literature reviewed in Section II-A is overwhelmingly concerned with displacement; suppression is comparatively thin ground, and the closest existing work (rotten green tests) is framed around unit-test assertion call sites rather than BDD step-definition lifecycle. If this paper contains a claim to novelty, it is here, offered with the caveat that a broader literature search may narrow or reframe it.

## V. SUPPRESSION AS A DISTINCT THREAT

Suppression is given its own section because its epistemics differ from displacement's in a way that matters for how the evidence in this paper should be weighed. A displaced failure is, eventually, a true signal: the cost is the engineer-hours spent tracing it through several architectural layers. A suppressed failure produces no signal at all. C6 and C7 both belong to a broader category that might be called "the step that never verified": the runner records a pass, but the specific check that would have caught the regression either operated on stale or default data (C6) or was never reached because a precondition silently excluded it (C7).

This has a direct methodological consequence. By definition, a silent false pass cannot be found by reading the test report it corrupts — it must be discovered by some other means, most plausibly because a person noticed a feature was broken in production or manual testing while its automated test remained green. In this suite, both C6 and C7 were found this way: incidentally, while investigating a defect the suite itself had not flagged, rather than through a systematic audit of guard conditions and

cross-step state usage. The count of suppressed cases reported here is a lower bound; a systematic audit of every guard clause and cross-step data dependency in the suite — not completed at the time of this draft — is identified as the natural next step, and one whose method of discovery is itself worth reporting in detail, since it cannot be reconstructed from run history alone.

What is, and is not, being claimed as new should be stated precisely. A passing test that verifies nothing is documented for unit tests as rotten green tests [7][8] and, more loosely, in the test-smell literature's treatment of assertion quality [6][10]. Not found documented is the specific architectural mechanism by which a BDD framework's step-definition lifecycle — a fresh page-object instance per step, by design, to avoid cross-step coupling of behaviour — can itself become the vector for cross-step state loss that produces a false pass. Whether this is a genuinely new mechanism or an already-known phenomenon appearing in a framework the existing literature has not yet examined is exactly the question a more exhaustive search, or peer review, should settle.

## VI. INTERVENTION: DEPENDENCY-PARTITIONED PARALLEL EXECUTION

Free parallelization of this suite is unavailable because a substantial number of scenarios consume records created by earlier scenarios in the same feature — a data-dependency chain rather than an accidental coupling. Naively parallelizing across all scenarios would reproduce the order-dependency failures documented broadly in the flaky-test literature [1][3][4]. As a mitigation, the suite's execution architecture was restructured into two concurrently running lanes: an ordered lane that preserves the original data-dependency chain, running alongside an independent-feature lane that is itself parallelized three-wide for scenarios with no such dependency. Per-lane JSON reports and rerun files are merged automatically after execution, so downstream reporting and flake-rerun tooling require no change.

This architecture is implemented and in routine use. A controlled before/after wall-clock comparison — the same commit, the same environment, run serially and then under the two-lane architecture, across each suite tier — has not yet been completed, and no speed-up figure is reported here in order to avoid presenting an unmeasured quantity as a result. The architecture and its motivating constraint are the contribution of this section; the timing evaluation is flagged as the paper's most direct piece of unfinished work, requiring two regression runs on one evening per suite tier.

A related question, surfaced rather than resolved here, is whether the data-dependency chain between scenarios is itself best understood as a test-design smell in its own right — one that imposes a direct maintenance cost [3] and blocks parallelization as a secondary consequence. If so, that coupling, rather than the parallel-lane mechanism built to work around it, may be the more durable contribution of this part of the study.

## VII. THREATS TO VALIDITY

Single application, single team. All seven classes were diagnosed in one product and one team's practices. This is mitigated by grounding each class in framework- and library-level behaviour rather than product-specific business logic, so mechanisms plausibly transfer even where relative frequencies would not.

Classification by a single analyst. The taxonomy and its axis assignment were constructed by the engineer who diagnosed the defects. Independent re-classification of a sample, with reported inter-rater agreement, would strengthen this claim and had not been completed at the time of this draft.

Selection bias toward diagnosed failures. Only investigated failures appear in this taxonomy; undiagnosed and still-latent failures, including any unfixed instances of C1 beyond the sixteen identified page objects, are under-represented by construction.
The suppression count is a lower bound. False passes that were never noticed cannot be counted; the true

figure is unknown and, without a systematic audit, unknowable from available report data.

Environment noise and unmeasured intervention. Latencies in Section IV were captured on shared pre-production infrastructure as single measurements rather than a distribution across repeated trials, and the Section VI intervention is reported without a wall-clock before/after measurement. Both should be treated as pending empirical validation rather than settled results.

## VIII. CONCLUSION

This paper has reported a seven-class taxonomy of test defects diagnosed in a large industrial Playwright/Cucumber end-to-end suite, organized along a displacement/suppression axis that separates failures which are merely hard to locate from failures that are never reported at all. Five classes displace the observable failure site away from the true defect; two classes — cross-step state loss and misdirected guards — suppress the failure signal entirely. The suppression classes are argued to be structurally analogous to, but architecturally distinct from, previously reported rotten-green-test phenomena in unit testing, and a dependency-partitioned parallel execution architecture adopted to reduce suite run time without reintroducing order-dependency failures is described. The unifying observation across all seven classes is that the observable failure — or, in the suppression cases, the observable absence of failure — is almost never located at the site of the underlying defect. Several quantitative gaps, most notably the wall-clock benefit of the parallel-execution intervention and the inter-rater agreement of the taxonomy's classification, are left unfilled rather than estimated, and closing them is identified as the immediate next step toward a submission-ready version of this work.

## REFERENCES

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE), 2014, pp. 643–653.

[2] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, “Understanding flaky tests: The developer's perspective,” in Proc. 27th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE), 2019, pp. 830–840.

[3] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, “Empirically revisiting the test independence assumption,” in Proc. Int. Symp. on Software Testing and Analysis (ISSTA), 2014, pp. 385–396.

[4] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, “An empirical study of flaky tests in Python,” in Proc. IEEE Conf. on Software Testing, Verification and Validation (ICST), 2021, pp. 148–158.

[5] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, “Root causing flaky tests in a large-scale industrial setting,” in Proc. Int. Symp. on Software Testing and Analysis (ISSTA), 2019, pp. 204–215.

[6] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, “Refactoring test code,” in Proc. 2nd Int. Conf. on Extreme Programming and Flexible Processes in Software Engineering (XP2001), 2001, pp. 92–95.

[7] J. Delplanque, A. Etien, S. Ducasse, and C. Fuhrman, “Rotten green tests,” in Proc. 41st Int. Conf. on Software Engineering (ICSE), 2019, pp. 500–511.

[8] M. Martinez, A. Etien, S. Ducasse, and C. Fuhrman, “RTj: A Java framework for detecting and refactoring rotten green test cases,” arXiv:1912.07322, 2019.

[9] A. Berndt, T. Bach, and S. Baltes, “Flaky tests in a large industrial database management system: An empirical study of fixed issue reports for SAP HANA,” arXiv:2602.03556, 2026.

[10] R. Santana, L. Martins, T. Virgínio, L. Soares, H. Costa, and I. Machado, “Refactoring Assertion Roulette and Duplicate Assert test smells: a controlled experiment,” arXiv:2207.05539, 2022.

[11] C. Landin, S. Tahvili, H. Haggren, M. Längkvist, A. Muhammad, and A. Loutfi, “Cluster-based parallel testing using semantic analysis,” in Proc. 2nd IEEE Int. Conf. on Artificial Intelligence Testing (AITest), 2020, pp. 99–106.