UDK 004.415.53

PIPELINE-NATIVE MODELS FOR DETECTING AND MINIMISING FLAKY TESTS

ANTONKIN V, KHOSHABA O (pzmag2022@gmail.com)

Vinnitsia National Technical University (Ukraine)

Abstract. Flaky tests - non-deterministic pass/fail outcomes on unchanged code—undermine CI/CD reliability. We present a pipeline-native framework that combines Bayesian outcome modelling, controlled test re-ordering, coverage- and change-aware triage, and a light-weight machine-learning predictor to identify likely flakiness early and cheaply. A cost-aware policy allocates re-runs and restricts quarantine to time-boxed, auto-revalidated cases. Minimisation emphasises hermetic builds, deterministic time/network abstractions, concurrency isolation, and automated fixes (e.g., seeding, teardown guards). The design is grounded in empirical evidence on dominant causes (async waits, concurrency, order dependency) and industrial practice at scale. The result is reduced noise, preserved bug-finding power, and faster feedback cycles in CI/CD.

Keywords: flaky tests; CI/CD; order dependency; deterministic testing; Bayesian modelling; test quarantine.

Problem Statement. In modern CI/CD, developers expect each failing test to indicate a genuine regression. Flaky tests violate this assumption, creating false alarms that waste engineering time, consume compute resources, and obscure real defects [1]. Empirical studies show that flakiness is both prevalent and multi-causal—commonly due to asynchronous timing, concurrency, or test-order dependency [1]. Industrial reports and datasets further indicate that naïve re-run policies and indiscriminate quarantine lists are palliative at best, masking rather than removing root causes [2–4]. The research problem is to devise detection models and pipeline-integrated procedures that (a) accurately identify likely flakiness early; (b) minimise its operational impact; and (c) guide durable fixes without suppressing tests that do reveal faults.

Objectives and Solved Tasks.

We address seven tasks:

1. Taxonomise flakiness mechanisms relevant to pipelines. We adopt a practical taxonomy—order-dependent (OD) vs non-order-dependent (NOD)—and map concrete manifestations (async waits, race conditions, resource leaks, time/network non-determinism) to pipeline controls [1–3].

2. Formulate a probabilistic detection model. We define a Bayesian Flakiness Score (BFS) for each test using Bernoulli outcome histories under stable code, applying hierarchical shrinkage across projects to remain data-efficient on small suites. BFS drives triage, alerts, and re-run budgeting.

3. Detect order dependencies. We integrate targeted re-ordering (class/method permutations) and repetition strategies to surface OD failures with minimal wall-clock cost, inspired by iDFlakies configurations [2].

4. Exploit coverage and change information. We incorporate change-vs-coverage reasoning—if a newly failing test did not cover changed code, prefer the flakiness hypothesis (DeFlaker principle) and route to OD/NOD checks [3].

5. Predict flakiness before re-runs. We train a light-weight classifier on readily available artefacts (recent pass/fail streaks, duration variability, historical environment signals, basic static metrics) to prioritise suspect tests; this complements BFS and reduces confirmation runs [5].

6. Optimise re-run and quarantine policy. We cast re-run decisions as a cost-minimisation problem, balancing build latency, developer time, and the probability of masking a genuine fault. A constrained “quarantine-with-expiry” is used sparingly, with automatic re-admission gates.

7. Codify minimisation and automatic fixes. We provide a remediation playbook that enforces hermeticity, fixed seeds, virtual clocks, strict timeouts, deterministic mocks, resource cleanup, and concurrency isolation. For OD, we add state-reset checks and teardown guards; for async timing, we replace sleeps with condition waits; for network/IO, we virtualise endpoints.

Essence of the Proposed Approach.

1. Statistical Backbone. For a test t with a stream of outcomes on unchanged code, we model pass/fail as Bernoulli(pt); the posterior $P(pt>\theta \mid data)$ (e.g., with a Beta prior and project-level hyper-prior) yields BFS. A high BFS indicates likely flakiness; a low BFS suggests a fault. This unified score governs: (i) when to re-run; (ii) when to schedule OD re-ordering; and (iii) when to open an automated remediation ticket. (No formulas shown per requirements.)

2. Order-Dependency Discovery. We apply bounded randomisation of class/method orderings and an original-order repetition budget. OD likelihood rises if a failure toggles with order under constant code; otherwise, the test proceeds to NOD diagnostics (timing, concurrency, resources). This mirrors the effectiveness of controlled re-ordering in prior work [2].

3. Coverage- and Change-Aware Heuristics. If a failing test covers no modified code regions, a flakiness hypothesis gains prior weight [3]. We cache coverage snapshots and intersect with diffs to gate re-runs cheaply before expensive complete suites.

4. Learning-Based Prediction. A pipeline-friendly classifier (e.g., gradient-boosted trees) uses: short-horizon pass/fail windows, execution time variance, retry history, environment jitter (CPU pressure, clock skew, network), and basic static test metrics. Unlike heavyweight static analysis, features are inexpensive to collect and align with ICSE-grade predictors [5].

5. Cost-Aware Re-run Policy. We model the expected utility of re-runs vs immediate fail/quarantine. The policy allocates a flakiness budget per build, preferring re-runs on tests with high BFS but high failure impact (broad coverage, gate tests). Quarantine, if used, is time-boxed with automatic re-evaluation to avoid coverage erosion.

6. Minimisation and Automated Fixes. We treat flakiness as an environmental non-determinism leak. Pipeline measures include: fully hermetic containers; pinned dependencies; fixed seeds; virtualised time; deterministic network stubs; file-system sandboxes; aggressive resource cleanup; per-test process isolation for concurrency; and teardown checks to reset shared state. For OD tests, we add isolation of global/static state and enforce idempotent setup/teardown hooks. These align with industrial experience, which shows that focusing on async waits, concurrency, and order yields the highest payoff [1,4].

Results in Context (What the Literature Implies).

Empirical evidence shows (i) async waits, concurrency, and order dependency dominate root causes [1]; (ii) OD/NOD rates are near-even in large Java ecosystems, validating the need for re-ordering checks [2]; (iii) change-vs-coverage reasoning is practical for triage [3]; and (iv) at Google-scale, dedicated tooling can reliably localise root causes and improve adoption when integrated into developer workflows [4]. ML-based predictors reduce re-run costs and can prioritise suspect tests before execution churn [5], while surveys and multivocal reviews catalogue effective responses and pitfalls [6,7]. These insights motivated our combined statistical–structural learning design.

Conclusions.

We contribute a pipeline-native framework to detect and minimise flakiness without losing the defect-revealing value of unreliable tests. The BFS-driven triage, OD discovery, coverage-aware heuristics, and ML prediction form a coherent decision layer that curbs needless re-runs and guides automated fixes. Future work targets (a) failure-focused prediction at the build level (reducing missed faults), (b) generalisation beyond JVM ecosystems, and (c) continuous learning loops that adapt policies as projects evolve [2–7].

To ground our study in empirically established baselines, we reference three widely used anchor figures. First, within the iDFlakies corpus of 422 flaky tests, order-dependent and non-order-dependent cases appear in approximately equal proportions (50.5% versus 49.5% [2]), which motivates treating both classes symmetrically in our data collection and model design. Second, industrial root-cause localisation at Google reports 82% accuracy across large-scale codebases [4], providing a realistic upper bound for diagnostic precision in practice. Third, live deployment of DeFlaker on 96 open-source Java projects surfaced 87 previously unknown flaky tests at an approximate false-positive rate of 1.5% [3], illustrating the practical trade-off between coverage and noise in CI settings. In our work, these anchors serve strictly as external reference points rather than as claims of novelty; we use them to calibrate thresholds (e.g., for order-sensitivity detection), to select evaluation metrics (precision and false-positive rate), and to frame any observed improvements in relative terms. When reporting results, we therefore contrast our measured outcomes against these baselines and, in the Conclusions, include a concise statement that qualifies any gains with explicit citations to [2], [4] and [3].

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 (ESEC/FSE), 2014, pp. 643–653. doi:10.1145/2635868.2635920.

[2] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, “iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests,” in 2019 IEEE 12th Int. Conf. Software Testing, Verification and Validation (ICST), Xi’an, China, 2019, pp. 312–322. doi:10.1109/ICST.2019.00038.

[3] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “DeFlaker: Automatically Detecting Flaky Tests,” in Proc. 40th Int. Conf. Software Engineering (ICSE), 2018, pp. 433–444. doi:10.1145/3180155.3180164.

[4] C. Ziftci and D. Cavalcanti, “De-Flake Your Tests: Automatically Locating Root Causes of Flaky Tests in Code at Google,” in 2020 IEEE Int. Conf. Software Maintenance and Evolution (ICSME), 2020, pp. 736–745.

[5] A. Alshammari, C. Morris, M. Hilton, and J. Bell, “FlakeFlagger: Predicting Flakiness Without Rerunning Tests,” in 2021 IEEE/ACM 43rd Int. Conf. on Software Engineering (ICSE), 2021, pp. 1572–1584. doi:10.1109/ICSE43902.2021.00140.

[6] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, “A Survey of Flaky Tests,” ACM Trans. Softw. Eng. Methodol., vol. 31, no. 1, 2022. doi:10.1145/3476105.

[7] A. Tahir, S. Rasheed, J. Dietrich, N. Hashemi, and L. Zhang, “Test flakiness’ causes, detection, impact and responses: A multivocal review,” Journal of Systems and Software, vol. 206, art. 111837, 2023. doi:10.1016/j.jss.2023.111837.