Shanto Rahman

RESEARCH STATEMENT

My research interests are in **Software Engineering**, with a focus on **Software Testing** and its application to improve the reliability of modern software systems. Software testing is essential for identifying defects early, preventing security breaches, and protecting user data. Yet reliability remains difficult because of the broken tests. Tests can be broken because of two main reasons: (1) **nondeterminism in tests execution**, which leads to tests that intermittently fail due to factors such as asynchrony, shared state, or unordered behavior; and (2) **change-induced test breakage**, which occurs when code or APIs evolve while tests remain unchanged. These challenges are pervasive across both open-source and industrial settings, including organizations such as Apple, Google, Meta, and Microsoft.

The goal of my research is to develop novel testing techniques and tools to make software systems more reliable. I achieve this goal by tackling three main problems: predicting, reproducing, and repairing test failures. First, I develop techniques to **predict test failures** by analyzing code changes and test behavior, which helps proactively identify flaky or fragile tests before they cause problems. Second, I build tools to **reproduce test failures** – especially nondeterministic (flaky) failures – in a reliable manner, enabling developers to diagnose and debug intermittent issues effectively. Third, I create automated approaches to **repair test failures** by fixing tests that have nondeterminism and updating tests when software evolves, thereby reducing manual maintenance effort.

To fulfill my research goal, I have designed, developed, and evaluated an integrated set of methods and tools that address flakiness and test brittleness in evolving software systems. My work is organized into three thrusts:

Thrust 1. I build a fine-tuned a large language model (LLM)-based classifier that identifies likely nondeterminism in tests (i.e., flaky tests) directly from code and predicts their root-cause, while attributing the most influential code tokens [10]. To make this usable at scale, I apply post-training quantization and a lightweight classifier to preserve accuracy while substantially reducing inference time and memory [7].

Thrust 2. I develop a method to deterministically reproduce nondeterminstic tests (i.e., flaky tests) by controlling execution schedules, turning sporadic failures into debuggable instances [8]. For order-dependent flakiness, I introduce a ranking approach that rapidly pinpoints the most likely culprit tests responsible for a given failure [9].

Thrust 3. I create a technique that eliminates flakiness by inserting the necessary synchronization into test and code [4]. Complementing this, I develop a change-aware, LLM-guided repair approach that updates tests when software evolves, keeping suites aligned with current behavior [11].

Collectively, these thrusts minimize false alarms from flaky tests, streamline failure diagnosis, and ensure that test suites remain dependable as software grows and changes. Some of these techniques have also been evaluated or adopted in industrial workflows like Amazon Web Service (AWS), underlining their potential impact on large-scale systems.

Interpretable and Efficient Flaky-Test Prediction

Flaky tests pass or fail unpredictably on the same code, so their failures cannot be trusted as clear signs of a real bug. Developers often rerun the same tests many times to determine whether a failure is flaky, but this practice is costly and wastes developer time. A more effective strategy is to **predict flakiness from code before it manifests**. But a binary "flaky/not flaky" label is not enough: remediation depends on the **root cause** (e.g., async-wait, shared state, order dependence, timing). We therefore need methods that both identify likely flaky tests and **explain why**, so engineers can apply the appropriate fix, **ultimately repairing**

the test. In this thrust, I **develop interpretable and efficient methods to identify flaky tests directly from code and to explain their causes**. I introduced **FlakyLens**, a fine-tuned LLM classifier that predicts a flaky test's **root-cause category** from its source (e.g., async-wait, order-dependent, timing-sensitive, unordered-collection), leveraging structural signals such as sleep/wait calls, shared state, assertions, and iteration over unordered sets. To make predictions actionable, FlakyLens provides **token-level attributions** that highlight the code regions most responsible for the decision, consistently surfacing patterns like sleeps, nondeterministic iteration, or global-state mutation. These explanations guide developers toward concrete fixes (e.g., adding synchronization, isolating state, or avoiding unordered traversals). To support realistic evaluation, we curated FlakeBench, a dataset of 280 flaky and 8,000+ non-flaky tests from real projects, **balanced by root cause** and designed for reproducible benchmarking.

While LLM-based classifiers such as FlakyLens has proved highly accurate at predicting flakiness, their inference cost can hinder practical use in CI. I address this issue with FlakyQ, an approach to make **flaky-test prediction efficient** without sacrificing accuracy. FlakyQ decouples feature extraction from prediction: FlakyQ makes flaky-test prediction faster and lighter without losing accuracy. It compresses the fine-tuned CodeBERT model to 8-bit (a common model-compression technique) and swaps its final neural layer with a simpler machine learning classifier (such as a random forest) that works on the model's feature outputs. This design preserves the rich predictive signals of LLM embeddings while substantially reducing runtime and memory use. In our experiments, FlakyQ retained or slightly improved classification accuracy relative to the full LLM, while **reducing prediction time by 25–35% and memory saving by approximately 48%**. Importantly, we found that a small ML classifier on top of LLM features outperformed models that rely only on simpler code features, confirming that the LLM captures meaningful signals. Together, FlakyLens and FlakyQ enable proactive, interpretable, and low-overhead flakiness screening at scale, allowing developers to identify fragile tests earlier and reduce CI costs.

Reproducing and Diagnosing Flaky Test Failures

Even after identifying the root causes of flaky tests, developers face a key challenge: debugging toward a correct repair, because flaky failures are difficult to reproduce reliably. My second research thrust therefore focuses on making flaky failures reproducible on demand and pinpointing their causes in complex test suites. Prior work highlights two dominant patterns of flakiness: (a) asynchronous/timing flakiness, where tests intermittently fail due to async-wait (e.g., missing waits for async events or thread scheduling issues), and (b) order-dependent flakiness, where outcomes change with test execution order (often due to shared mutable state or missing cleanup). These categories call for different strategies—synchronization for async/timing issues and isolation or order control for order-dependent cases—and my work develops targeted solutions for both.

For async-wait flakiness, I develop FlakeRake, a technique that **systematically reproduces** these hard-to-capture failures. Rather than relying on brute-force reruns, FlakeRake **searches for an execution** schedule that consistently triggers the known flaky test's failure. The key idea is to **inject targeted delays** into program operations: by slowing or pausing specific threads or asynchronous callbacks, FlakeRake induces the problematic timing conditions in a controlled manner. Given a sporadically failing test, FlakeRake uses guided search to find a delay configuration that reliably causes the test to fail on demand. I implemented FlakeRake for Java and evaluated it on 811 real flaky tests (with 1,167 historical flaky-failure instances). Compared to state-of-practice baselines (e.g., random delays or repeated reruns), FlakeRake reproduced more flaky failures and did so more reliably. By automating reproduction of elusive async-wait failures, FlakeRake eliminates tedious manual reruns and yields a stable failure instance for analysis and debugging. We released FlakeRake and its dataset as open source to facilitate replication and adoption. These results

show that making async-wait flakiness deterministic is feasible, **substantially reducing the debugging burden** that would otherwise consume developer hours.

Order-dependent (OD) test flakiness is when a test fails only because an earlier test left something behind (like a changed global state) that interferes with it. Finding which earlier test (the "polluter") caused the later test (the "victim") to fail is hard when we have hundreds or thousands of tests . Exhaustively permuting these large number of tests—or relying on trial and error—to find the polluter is prohibitively expensive. I address this issue with RankF [9], a two-pronged technique that ranks likely polluters to triage order-dependent flakiness. RankF combines RankF$_L$, a fine-tuned pre-trained BigBird transformer model to embed test code and estimate the likelihood that one test affects another, and RankF$_O$, a lightweight heuristic that scores test pairs using historical failures and signals such as shared resources or prior co-failures. Given a victim test exhibiting order sensitivity, RankF returns a prioritized list of suspect tests. In an evaluation on 155 real-world order-dependent flaky tests across 24 projects, RankF finds the true polluter as the top-ranked candidate far faster than random ordering or dependency-graph baselines. By rapidly narrowing the search space to a handful of likely culprits (e.g., polluter), RankF enables developers (or automated tools) to confirm and fix the offending interaction much more quickly—offering a scalable path to debugging flaky suites even as they grow.

Automated Repair of Broken Tests

Building on the preceding thrusts—prediction and reproducible diagnosis—I focus next on repair. Here, a broken test includes both flaky tests (nondeterministic outcomes) and tests that fail after code/API changes because their assertions, test setup code and simulated dependencies are out of date. I develop techniques that (i) stabilize nondeterminism and (ii) update tests to match current behavior, restoring trust in the test suite. Repairing tests before merge keeps the continuous integration pipeline healthy: test runs complete without spurious failures and continue to catch real bugs as intended. In this space, I contribute two complementary systems: FlakeSync [4], which repairs flakiness caused by asynchronous or concurrency issues by inserting the necessary synchronization, and UTFix [11], a change-aware, LLM-guided approach that updates tests when software evolves so suites stay aligned with the code.

FlakeSync targets asynchronous-wait flakiness, where tests fail because they do not properly coordinate with the code under test (for example, they check results before an async operation or thread has finished). The core insight behind FlakeSync is that many of these flaky tests can be fixed by introducing just the **right synchronization in the test code**, rather than relying on arbitrary sleeps or timing tweaks. FlakeSync automatically locates (i) a critical point in the code under test that must complete and (ii) a barrier point in the test that should wait for that completion. It then instruments the test to wait at the barrier until the critical action occurs (e.g., a callback fires or a shared resource is updated), converting a timing race into a deterministic sequence. In an evaluation on prior flaky tests, **FlakeSync repaired 83.8% of async-wait failures with essentially no runtime overhead in the fixed tests (median 1.0X)**. The search procedure is practical (median 59 minutes per repair) and yields patches that apply directly to project code; **I submitted 10 pull requests generated by FlakeSync to open-source projects, 3 of which have already been accepted (none rejected)**. By synchronizing the right code regions rather than inserting arbitrary sleeps, FlakeSync provides a reproducible path to stabilizing flaky tests, demonstrating how dynamic analysis and lightweight instrumentation can eliminate flakiness at its root.

Complementing the above, UTFix addresses change-induced test breakage, where code or API evolution causes tests to fail not because of a product bug, but because the tests are out of date. An internal study reports that such maintenance issues can account for up to one-fifth of failures in large services, yet fixes are typically manual. UTFix is a change-aware, LLM-guided framework that keeps tests in sync by

proposing minimal edits that preserve test intent while adapting to new behavior. Given pre- and post-change code, a failing test, and its failure message, UTFix formulates repair as a contextual code-editing task: it builds a prompt from the code diff plus static/dynamic slices of affected regions, asks a fine-tuned LLM to generate a patch, and then verifies the patch by rerunning tests and ensuring that coverage of the changed code is restored; if verification fails, UTFix iteratively refines the patch using diagnostics and constraints. In a benchmark of 44 open-source Python libraries, UTFix repaired 89.2% of tests with assertion failures due to code updates and restored full coverage in 96/369 impacted tests; evaluations on historical changes in large projects show similar effectiveness. The implementation, datasets (including SynBench), and results are publicly available, and **the approach has been evaluated and adopted in industry (e.g., Amazon Web Services (AWS))**. By catching and fixing obsolete tests before merge, UTFix reduces reruns, shortens time-to-integration, and turns test maintenance from a manual, open-ended task into a measurable, automated repair loop.

Other Work

Beyond test repair, I have worked on optimizing continuous development [2], detecting thread safety violations [6], anti-pattern detection [1], and bug localization [3, 5].

Optimizing Continuous Development (OPTCD). In continuous development, every commit runs the pipeline; therefore, redundant tasks and unused outputs waste time and slow developers' workflows. Hence, I propose OptCD [2] that targets this waste by analyzing CD logs to detect unused outputs, mapping each unused directory to its producing step (via name-based IR, LLM ranking, and timestamp correlation), validating candidates by temporarily disabling them, and auto-generating patches to skip those steps. After applying our patches on 22 Maven projects, we find a mean of 7% time reduction up to 47%. **I submitted 26 pull requests to fix the developers' tests, with 12 of them already accepted.**

Thread Safety Violations (TSVD). Another line of my work targets thread-safety issues in concurrent programs. I developed a tool called TSVD4J (ICSE Demo 2023) [6] that detects hidden race conditions by inserting delays around shared-data accesses to force problematic interleavings. I implement TSVD4J as an open-source Maven plugin that can find concurrency violations than the industry tool RV-Predict. This helps developers catch subtle threading bugs early, making test execution more robust under parallel runs.

Bug Localization and Anti-Pattern Detection. In earlier work [3, 5], I designed an IR-based bug-localization method that builds a Method–Statement Dependency Graph and enriches statements with context, often ranking the true faulty statement near the top (frequently within the top-10) on large benchmarks such as Eclipse, SWT, and PasswordProtector. I also explored machine-learning methods for detecting anti-patterns (code smells) [1], finding that an SVM with SMOTE and hyperparameter tuning performed best across three open-source Java projects. These contributions improve developer productivity by reducing debugging effort and highlighting maintainability risks early.

Future Work

Looking ahead, my research vision is to create a software development ecosystem where testing is largely self-healing – predicting flaky or fragile tests before they strike, reproducing intermittent failures deterministically, and repairing test issues automatically. To realize this vision, I will pursue several interconnected directions that integrate flaky-test management into development pipelines, address new sources of flakiness arising from AI and large-scale systems, and embed human-centric design to ensure these solutions are

practical and widely adopted.

Unified Flaky-Test Management in CI Pipelines. Flaky tests have emerged as a critical reliability challenge in modern CI/CD pipelines. These nondeterministic failures undermine confidence in test suites and impose real costs: for example, Google has reported that roughly 1.5% of all test runs yield flaky results and nearly 16% of tests overall have exhibited some degree of flakiness, with about 84% of observed test failures in their CI system ultimately traced to flaky tests rather than actual code defects. Such false positives waste developer time, delay releases, and can even cause teams to overlook genuine regressions. Importantly, many flaky failures stem not from the test logic at all but from underlying infrastructure and orchestration issues in the pipeline. In Kubernetes-based CI environments, for instance, problems like resource contention, or scheduling delays can introduce "workflow-level" flakiness that mimics test failures despite the application code being correct. These challenges motivate a pipeline-aware solution: the future research direction is to build an integrated, self-healing system that detects, diagnoses, and automatically repairs flaky tests within the CI/CD pipeline. By enabling the pipeline to autonomously identify flaky tests (distinguishing environmental issues from real bugs) and recover from them in real time, such a system aims to greatly improve reliability and trust in continuous integration results while reducing downtime and manual intervention.

One key direction is to integrate flaky-test prediction, reproduction, and repair into a unified system within continuous integration (CI) pipelines. By co-designing testing tools with the pipeline, a flakiness management framework could automatically anticipate flaky tests, trigger deterministic replays of sporadic failures, and apply fixes on the fly. This kind of self-healing CI/CD pipeline would prevent false failure alarms, avoid wasteful re-runs, and keep the software delivery process running smoothly. Ultimately, a predictive and adaptive pipeline not only improves reliability but also shortens release cycles by catching and correcting test issues proactively.

Ensuring Reliability for AI-Generated Code and Tests. Another emerging challenge involves the reliability of AI-generated code and tests. Large language models (LLMs) are increasingly used to generate code and test suites, so it is crucial to assess and improve their quality. I will systematically study how prone LLM-generated tests are to flaky failures, comparing their flakiness patterns to those of human-written tests. Building on early evidence that auto-generated tests can be just as flaky as those written by developers, I plan to develop techniques to detect such flaky tests and then apply targeted reproduction and repair strategies – improving the robustness of AI-produced test suites. Moreover, I aim to introduce a verification-first approach for LLM-generated code: extracting implicit specifications from the user's prompt and generating targeted tests before the code is deployed. By checking model-generated code against these prompt-derived test oracles, any deviation from the intended behavior can be caught and corrected early. This line of research will leverage my expertise in test generation and repair to ensure that as AI becomes more prevalent in software development, the resulting code and tests remain dependable.

Handling Environment-Induced Flakiness at Scale. I will also address flakiness that stems from the environment and infrastructure in large-scale systems. In resource-constrained or high-traffic scenarios, tests may pass under normal conditions but intermittently fail under stress (for example, due to heavy load, memory/CPU contention, network latency, or eventually-consistent data stores). Such environment-induced flaky tests are often overlooked by existing techniques that focus only on code-level nondeterminism. To tackle this, I plan to design infrastructure-aware testing techniques that detect and deterministically reproduce failures caused by underlying system conditions. For instance, we can simulate peak load or force resource contention to reliably trigger hidden race conditions, then apply appropriate fixes (such as adjusting timeouts or adding synchronization) to harden the tests. These solutions will enable more reliable testing at scale, ensuring that test suites remain stable even under real-world stresses and extreme conditions.

Interactive and Human-Centered Debugging Tools. In parallel, I am focusing on making automated testing tools interactive, explainable, and human-centered so that developers can trust and effectively adopt them. I envision an intelligent "flaky test assistant" integrated into development environments (for example,

as an IDE plugin) that not only flags or fixes flaky tests automatically but also explains its actions through intuitive visual cues – for example, by highlighting a problematic asynchronous call or flagging an order dependency that triggers flaky behavior. The tool keeps developers in the loop by asking for guidance when appropriate – for instance, to confirm an automatically suggested patch or to select the correct root-cause category. This approach effectively combines automation with human insight.

To ensure these tools truly resonate with practitioners, I will conduct empirical user studies and real-world trials. By observing how engineers use such assistants (including my own prototypes like FlakeSync and UTFix) in their normal workflow, I will gather feedback on usability, trust, and impact on debugging productivity. Key questions include whether developers trust automated test fixes and how explanatory features influence their willingness to adopt tool recommendations. Insights from these studies will guide iterative refinements, aligning the tools with developers' mental models and workflows. Through this human-centered approach, my aim is to turn state-of-the-art flaky-test techniques into usable, explainable, and widely adopted developer aids – advancing software reliability while enhancing the developer experience.

Broadening to New Domains and Platforms. Finally, to broaden the impact of my work, I will extend these reliability techniques to new domains and computing platforms that present unique testing challenges. In the near term, I plan to adapt my test-repair and synchronization approaches for mobile applications and machine learning (ML) pipelines. These systems introduce fresh sources of flakiness – for instance, mobile app GUI tests are brittle as the UI evolves frequently, and ML model pipelines can exhibit nondeterministic behavior due to factors like concurrency, resource contention, or randomness in data handling. I will collaborate with domain experts to apply and generalize my methods (originally developed for traditional software) to handle asynchronous mobile UIs and dynamic, data-driven ML workflows, ensuring that reliable testing tools are available where they are needed most.

Beyond these, I am excited to push into underserved domains such as robotics, Internet-of-Things (IoT) systems, scientific computing, and continuously learning services – areas where inherent randomness, hardware interaction, or always-on operation pose serious reliability challenges. Testing and debugging practices in these domains are often less mature, so they stand to benefit greatly from automated flaky-test management. By tailoring or inventing new techniques for these contexts (for example, systematically triggering rare sensor conditions in robots or designing test oracles that account for acceptable randomness in probabilistic programs), I aim to raise the reliability bar across a broad spectrum of software. Ultimately, this expansion will bring cutting-edge test automation to systems that have traditionally been difficult to validate, ensuring that even hard-to-test applications can achieve high reliability.

References

[1] N. Akhter, S. Rahman, and K. A. Taher. An anti-pattern detection technique using machine learning to improve code quality. In *2021 International Conference on Information and Communication Technology for Sustainable Development (ICICT4SD)*, pages 356–360. IEEE, 2021.

[2] T. Baral, S. Rahman, B. N. Chanumolu, B. Balcı, T. Tuncer, A. Shi, and W. Lam. Optimizing continuous development by detecting and preventing unnecessary content generation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 901–913. IEEE, 2023.

[3] S. Rahman and K. Sakib. An appropriate method ranking approach for localizing bugs using minimized search space. In *ENASE*, pages 303–309, 2016.

[4] S. Rahman and A. Shi. Flakesync: Automatically repairing async flaky tests. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[5] S. Rahman, M. M. Rahman, and K. Sakib. A statement level bug localization technique using statement dependency graph. In *ENASE*, pages 171–178, 2017.

[6] S. Rahman, C. Li, and A. Shi. Tsvd4j: Thread-safety violation detection for java. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 78–82. IEEE, 2023.

[7] 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)*, pages 93–104. IEEE, 2024.

[8] S. Rahman, A. Massey, W. Lam, A. Shi, and J. Bell. Automatically reproducing timing-dependent flaky-test failures. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 269–280. IEEE, 2024.

[9] S. Rahman, B. N. Chanumolu, S. Rafi, A. Shi, and W. Lam. Ranking relevant tests for order-dependent flaky tests. In *International Conference on Software Engineering*, 2025.

[10] S. Rahman, S. Dutta, and A. Shi. Understanding and improving flaky test classification. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2), 2025.

[11] S. Rahman, S. Kuhar, B. Cirisci, P. Garg, S. Wang, X. Ma, A. Deoras, and B. Ray. Utfix: Change aware unit test repairing using llm. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1): 143–168, 2025.