Shanto Rahman — Research Statement

My research is in Artificial Intelligence for Software Engineering (AI4SE), with a focus on Software Testing and its application to improve the reliability of modern software systems. Software testing is the most practical way to assess program correctness at scale. However, software testing suffers from *broken tests*, tests that fail unpredictably or produce misleading results unrelated to the correctness of the program (code-under-test). Tests can be broken for two main reasons: (1) **nondeterminism in test execution** that causes *intermittent (flaky) failures* driven by factors uncontrolled by the tests, such as timing sensitivity or thread scheduling; and (2) **change-induced test breakage** that causes *spurious assertion failures* when code changes but test logic remains outdated. These challenges are pervasive across open-source and industrial settings, including organizations such as Apple, Amazon, Google, Meta, Microsoft, and Uber. For example, in one study, Google finds *16% of 4.2 million tests (i.e., 672,000)* are flaky [1].

The goal of my research is to develop novel software testing techniques and tools that increase the reliability of software systems. I focus on repairing broken tests so their results are accurate and stable. Such tests provide clear evidence of intended behavior and ensure overall software reliability. However, repairing tests, especially when nondeterminism exists, is notoriously hard, as intermittent failures prevent effective debugging and validation. Nondeterminism can arise from different root causes, each requiring a distinct repair strategy. Moreover, identifying root causes and reliably reproducing their failures remain challenging in practice. Hence, my research proceeds in three complementary thrusts: (1) identifying root causes directly from code by fine-tuning a large language model (LLM) [2,3]; (2) reproducing failures for each root cause by lightweight heuristics, dynamic bytecode instrumentation, and fine-tuned LLMs [4,5,6]; and (3) repairing broken tests based on the type of root cause using program analysis and generative AI (GenAI) to preserve test intent, mitigate nondeterminism, and prevent breakage from code evolution [7,8].

Impact. My research is recognized in both industry and the open-source community. Google has shown strong interest in my work on flaky tests, and I am collaborating with them to mitigate flaky tests. Amazon has evaluated and deployed my research [8] to mitigate test breakage from code changes. In open-source, my work **automatically identified 2,100+ flaky tests**, and developers accepted my repair patches.

My vision is to make modern computing systems such as software, cloud services, and AI-driven intelligent systems reliable as they grow in scale, autonomy, and complexity. In the short term, I will tackle underexplored broken tests and extend my nondeterminism-aware reliability techniques to large service-based systems, where cross-component interactions and network delays create failures that are hard to reproduce, diagnose, and repair. In the long term, as AI becomes central to the computing stack, I will develop methods for testing and repairing AI models in single- and multi-agent settings, building on my AI4SE work to advance software engineering for AI, while keeping humans in the loop to ensure alignment with design intent.

Predicting Root Causes of Broken Tests

Developers often rerun the same test many times to determine whether a test is flaky, but this practice is costly and wastes developer time. Also, a binary "flaky vs non-flaky" classification is not enough, because it only indicates that flakiness exists and does not provide any root cause or repair guidance. We therefore need methods that not only flag likely flaky tests but also explain their root causes and do so quickly. A more effective strategy is to predict broken tests (e.g., flaky tests) and their root causes by analyzing code, without rerunning tests. Hence, **I developed interpretable and efficient machine learning (ML) models to identify the root causes of flaky tests**.

Interpretable and Improved Flaky Test Classification. I introduced FlakyLens [2], a fine-tuned LLM classifier that predicts a flaky test's root cause directly from test code and provides token-level attributions that pinpoint failure-causing code regions to guide concrete repair. For realistic evaluation, I curated and released FlakeBench, a dataset designed to span multiple root causes of flaky tests along with representative non-flaky tests. FlakyLens achieved 65.8% macro-F1 on root cause classification of flaky tests, outperforming the best fine-tuned baseline by 9.2 percentage points (pp) and the strongest open-source zero-shot, prompt-based LLM by 50.9pp. I further validated the explanations with semantics-preserving perturbations (e.g., injecting root cause-specific tokens such as *sleep* into tests from a different root cause), which reduced macro-

F1 by up to 18.4pp. These reductions indicate that the classifier relies on these tokens, and the attributions help developers prioritize inspection by pointing to the code responsible for the root cause.

Efficient and Lightweight Flaky Test Classification. A complementary strategy to FlakyLens is to make the classification model faster and lightweight, since the cost of inference and high memory requirements of large models can hinder practical use, especially in continuous integration (CI) pipelines. Hence, I proposed FlakyQ [3], an approach for efficient, lightweight flaky test classification without sacrificing accuracy. FlakyQ compresses the fine-tuned CodeBERT model down to 8-bit precision by applying post-training quantization on each linear layer. It then replaces the model's final neural layer with a simple ML classifier (e.g., a random forest) trained on the LLM's extracted feature vectors. This design preserves the rich predictive signals captured by the LLM's embeddings while substantially reducing runtime and memory usage. In my experiments, FlakyQ reduced prediction time by 25.4% and memory usage by 48.0% on average, while retaining the same or even slightly improved classification accuracy compared to the full fine-tuned LLM. These results confirm that the LLM captured meaningful features from test code that simpler models could leverage, allowing FlakyQ to be both effective and highly efficient.

Together, FlakyLens and FlakyQ enable **proactive, interpretable, and low-overhead identification of flaky test root causes** at scale, so developers can identify broken tests earlier and cut CI costs.

Reproducing Broken Test Failures

When a test breaks, developers need a reliable test failure to debug. If a failure reproduces on every run (i.e., deterministic), developers can trace the root cause and repair it. However, the real trouble arises when test failures are nondeterministic and intermittent, which, in turn, blocks diagnosis and wastes many hours of debugging effort. Hence, my work made intermittent failures deterministic and reproducible, making them easier to debug and repair. I am the *first* to reproduce failures for the two most prevalent root causes of flaky tests: (1) async-wait and (2) order-dependence. For each root cause, **I designed targeted strategies that reliably trigger the failure, enabling repeatable debugging and faster repair**.

Reproducing Async-Wait Flaky Failure. An async-wait flaky test intermittently fails as it checks results before its dependent asynchronous operation completes. I developed FlakeRake that reliably reproduces these failures [4], [6]. Instead of thousands of reruns to catch a single failure, FlakeRake injects small delays via bytecode instrumentation and searches for a failure-inducing configuration (i.e., executing thread, source line to delay, and delay duration). The key idea is to slow or pause specific threads or asynchronous callbacks. FlakeRake operates in three phases: (1) profiling executions to record timing-dependent API calls (e.g., sleep, await, notify) and their invoking threads, producing <source code line, thread> pairs; (2) running a guided search (i.e., Bisection) over subsets of those pairs to find a minimal failure-inducing configuration; and (3) confirming the configuration by reproducing the same stack trace failure while automatically applying the required delays. In my evaluation, FlakeRake reproduced 136 test failures, of which 107 had a reproduction rate over 50% and 93 reached at least 99%. In contrast, the prior rerun baseline that runs each flaky test 10,000 times reproduced 115 test failures, with none achieving at least 50% reproduction rate.

Reproducing Order-Dependent Flaky Failure. Order-Dependence between tests occurs when an earlier test pollutes shared state, causing a later test to fail. Finding the earlier test (the "polluter") that makes the later test (the "victim") fail is hard when there are hundreds or thousands of tests. However, we need to identify the polluter to reliably reproduce the failure of the victim test, enabling us to repair the test. Hence, I proposed RankF [5] that identifies the polluter test for a given victim. RankF has two complementary components: (1) RankF$_L$, a fine-tuned BigBird model that embeds tests to estimate cross-test influence, and (2) RankF$_O$, a lightweight heuristic that uses historical co-failures and execution order to find the polluter test. In my evaluation of 155 real order-dependent flaky tests across 24 open-source GitHub projects, RankF found the true polluter more often and in less time than the common baselines. It ran about 2.4× faster than Bisection approach (a baseline that iteratively halves the candidate polluter tests), and about 12.6× faster than another baseline One-by-one (an approach that tries each candidate polluter test to run sequentially with the victim test). These results allowed developers to quickly confirm the polluter and understand the interfering interaction, providing a scalable path to debugging and eventual repair of flaky tests.

Together, FlakeRake and RankF enable **reliable and low-overhead reproduction of flaky failures**, helping developers consistently debug nondeterministic test behavior.

Repairing Broken Tests

Effective test repair depends on the type of broken tests. Hence, I focused on two major types of broken tests: (1) nondeterministic (i.e., flaky) tests and (2) change-induced (i.e., outdated) tests, which yield unreliable or spurious assertion failures and reduced code coverage. **I developed techniques that repair flaky tests, and keep tests aligned with code changes, thereby mitigating test breakage**.

Repairing Flaky Tests. Since async-wait flakiness is the most prevalent and notorious root cause, I proposed FlakeSync, the *first* repair strategy for such tests [7]. The key insight behind FlakeSync is to introduce the right synchronization between the test code and the code-under-test, rather than relying on arbitrary sleeps or timing tweaks. FlakeSync automatically locates (1) a critical point in the code-under-test that must be completed before any test action that relies on it, and (2) a barrier point in the test code where execution should pause until that critical point completes execution. FlakeSync pinpoints the critical point by instrumenting delays into overlapping method executions on different threads until a minimal-delay location reliably triggers the failure. It then identifies a barrier point, by re-executing the failing run and scanning the failure stack trace backward, stopping at the earliest location that needs to wait until the critical point has been executed before proceeding, converting the timing race into a deterministic execution. In my evaluation, FlakeSync repaired 83.8% of async-wait flaky tests with essentially no runtime overhead in the repaired tests. **I submitted 10 pull requests generated by FlakeSync to open-source projects, three of which have already been accepted (none rejected)**.

Repairing Change-Induced Test Breakage. Change-Induced deterministic test breakage occurs due to the code evolution. More specifically, the test breaks not due to product bugs but because it becomes out-of-date, e.g., assertions no longer match intended behavior or code coverage drops over changed code. Prior research reports that 22% of failures in large services come from outdated tests. Hence, keeping tests aligned with code changes is essential. I proposed UTFix [8], a change-aware, LLM-guided framework that repairs outdated tests through minimal edits that preserve tests intent while adapting to new behavior. Given pre- and post-change code, a failing test, and its failure message, UTFix first generates an automated prompt from the code diff and static or dynamic code slices of the affected regions. It then invokes an LLM to generate a patch and validates the repair by restoring the test and code coverage over the changed code. If validation fails, UTFix iteratively refines the patch using the test-repair intent. For evaluation, I curated Syn-Bench, the *first* public dataset of changed code and outdated tests from 44 open-source Python projects. UTFix repaired 89.2% of tests with assertion failures due to code changes and achieved 76.92% code coverage. **My approach has been evaluated and deployed in Amazon Web Services (AWS)**.

Collectively, FlakeSync and UTFix provide the *first* systematic techniques to automatically **repair both flaky and outdated tests**, enhancing test reliability and reducing manual maintenance effort.

Other Research Contributions

Beyond software test repair, I also worked on optimizing continuous development (CD) [9], software bug detection [10], and localization [10,11,12].

Optimizing Continuous Development (OptCD). CD is a software practice in which code is continuously built, tested, integrated, and deployed. Large companies (e.g., Amazon, Google, Meta) rely on CD pipelines to manage constant updates across thousands of services. As every commit triggers the pipeline, redundant tasks whose results no one consumes add unnecessary developer wait time. I proposed OptCD [9,13] that analyzes CD logs and artifacts to detect unused outputs, then maps each to its producing step to identify the responsible plugin (e.g., Maven). I validated candidates by disabling the corresponding plugin and generating patches to skip it. I applied those patches to 22 open-source Maven projects, which reduced CI time by up to 47%. **I submitted 26 pull requests with the generated patches, 12 of which were accepted**.

Bug Detection and Localization. I developed TSVD4J [10], a concurrency bug detection technique that detects hidden data races in Java programs by exposing unsafe interleavings around shared data. I implemented TSVD4J as an open-source Maven plugin and evaluated it on 12 Java applications. TSVD4J found 55 data races, compared to 17 with the baseline (RV-Predict), helping developers catch subtle threading bugs early. I also designed information retrieval based bug localization techniques that rank statements based on the likelihood of being buggy [11,12,14], leveraging method-level dependency graphs and enclosing-method context. I achieved 66.1% Top-10 accuracy on large benchmarks (e.g., Eclipse).

Future Work

As long as people write software, bugs will persist and testing will remain the most practical way to detect them. In the short term, I will continue tackling underexplored broken tests to make testing reliable. In the long term, I will address nondeterminism and noise-induced drift in emerging domains including cloud services, ML models, and ML-enabled autonomous systems, where the toughest reliability problems remain. For example, a recent AWS outage was caused by nondeterminism in a cloud service [15]. To this end, I will develop nondeterminism-aware testing for cloud services (e.g., races, flaky RPC) and extend testing principles to ML models, a transition of my AI4SE research to Software Engineering for AI (SE4AI). I will detect and repair sources of nondeterminism, adversarial threats, and subgroup bias, validating and refining these methods through human feedback to make models robust, secure, and fair. Building on my expertise in software testing and ML, I am uniquely positioned to pursue these research directions.

Tackling Underexplored Flaky Tests. My work to date on debugging and repairing flaky tests has focused on two leading root causes. Although these root causes are prevalent in practice, prior empirical study reports eight additional sources of flakiness that remain underexplored [16]. In my next research, I will target these underexplored sources of test flakiness, such as infrastructure flakiness. Following my research FlakyLens [2], I will first leverage root cause prediction and token-level attribution to identify signals that correlate with infrastructure failures. Next, I will systematically perturb the environment to expose failure reproducibility specific to the root cause. Finally, I will design automated dynamic repair strategies that adapt to the infrastructure, and validate the patched code under realistic perturbations.

Nondeterminism-Aware Reliability for Cloud Services. Cloud services execute across many nodes, threads, and components whose actions interleave in nondeterministic ways. In practice, major system failures due to concurrency are hard to detect in advance during development time, e.g., Time-of-Check to Time-of-Use (TOCTOU)-like races in the recent AWS outage were not caught during development. Building on my experience with concurrency-bug detection and localization in standalone systems (e.g., TSVD4J), I plan to extend these ideas to cloud or distributed system settings and pair them with failure reproduction via an enhanced FlakeRake [4]-style approach. Using these reproducible cases, I will develop concurrency-aware bug repair strategies for cloud services to make the services reliable.

Nondeterminism and Trustworthy ML Systems. ML pipelines introduce broader sources of nondeterminism than traditional software (e.g., seed or batch sensitivity, GPU variability). Building on my AI4SE work, I will extend Software Engineering principles to ML to improve the reliability of ML pipelines. My focus is to expose, reproduce, and repair these ML-specific sources of nondeterminism by treating pipelines as testable executions (e.g., search over seeds, batch schedules) to obtain failure-inducing configurations, and then develop repair strategies for these nondeterministic ML systems. Finally, I will elicit **developer preferences** and confirm the chosen repair before applying it as a patch. Beyond stability, ML systems must remain trustworthy under distribution shift, adversarial attacks, and subgroup variation. In practice, existing techniques rarely provide intention-preserving repairs or human-guided trade-off control, leaving a major gap. My research FlakyLens [2] shows small, structured perturbations substantially degrade model performance, evidencing their sensitivity to superficial cues. To generalize these findings, I will incorporate domain experts' input to specify fairness and calibration targets, and acceptable risk tolerances. Finally, I aim to design targeted, intention-preserving repairs that will make ML models robust, secure, and fair.

Reliability and Safety in ML-Enabled Intelligent Systems. As intelligent systems such as robots, autonomous vehicles, and ML-enabled medical devices integrate ML into physical devices and networked services, ensuring reliability, robustness, and safety is paramount. These systems face unique challenges because they combine data from multiple modalities (e.g., sensors, logs, and ML predictions), each with its own noise, drift, and failure modes. Even moderate shifts in environment, sensor behavior, or workload can silently push these intelligent systems toward failure. Hence, I will detect early warning signs in intelligent systems by mining testing logs and simulation runs to learn patterns that typically occur before failures so the system can flag and mitigate problems early. I will also develop learning-based mechanisms that adapt safely when operating conditions such as environment, sensor behavior, or workload change, so the system remains stable and reliable as conditions evolve. Despite these safeguards during operation, any changes in these systems can still introduce unexpected behavior at deployment time. Therefore, I will enforce runtime checks such as specifications, invariants, and sanity tests with staged deployments that require human approval.

References

[1] John Micco. The State of Continuous Integration Testing@Google. In *International Conference on Software Testing, Verification, and Validation (ICST)*, 2017.

[2] **Shanto Rahman**, Saikat Dutta, and August Shi. Understanding and Improving Flaky Test Classification. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2025.

[3] **Shanto Rahman**, Abdelrahman Baz, Sasa Misailovic, and August Shi. Quantizing Large-Language Models for Predicting Flaky Tests. In *International Conference on Software Testing, Verification, and Validation (ICST)*, 2024.

[4] **Shanto Rahman**, Aaron Massey, Wing Lam, August Shi, and Jonathan Bell. Automatically Reproducing Timing-Dependent Flaky-Test Failures. In *International Conference on Software Testing, Verification, and Validation (ICST)*, 2024.

[5] **Shanto Rahman**, Bala Chanumolu, Suzzana Rafi, August Shi, and Wing Lam. Ranking Relevant Tests for Order-Dependent Flaky Tests. In *International Conference on Software Engineering (ICSE)*, 2025.

[6] **Shanto Rahman**, Talank Baral, August Shi, and Wing Lam. Reproducing Timing-Dependent Flaky Test Failures via a Template-Guided LLM Pipeline. In *Under Submission*, 2026.

[7] **Shanto Rahman** and August Shi. FlakeSync: Automatically Repairing Async Flaky Tests. In *International Conference on Software Engineering (ICSE)*, 2024.

[8] **Shanto Rahman**, Sachit Kuhar, Berk Cirisci, Pranav Garg, Shiqi Wang, Xiaofei Ma, Anoop Deoras, and Baishakhi Ray. UTFix: Change Aware Unit Test Repairing Using LLM. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2025.

[9] Talank Baral, **Shanto Rahman**, Bala Naren Chanumolu, Başak Balcı, Tuna Tuncer, August Shi, and Wing Lam. Optimizing Continuous Development by Detecting and Preventing Unnecessary Content Generation. In *International Conference on Automated Software Engineering (ASE)*, 2023.

[10] **Shanto Rahman**, Chengpeng Li, and August Shi. TSVD4J: Thread-Safety Violation Detection for Java. In *International Conference on Software Engineering (ICSE'Demo)*, 2023.

[11] **Shanto Rahman**, Md Mostafijur Rahman, and Kazi Sakib. A Statement Level Bug Localization Technique using Statement Dependency Graph. In *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2017.

[12] **Shanto Rahman** and Kazi Sakib. An Appropriate Method Ranking Approach for Localizing Bugs using Minimized Search Space. In *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2016.

[13] Talank Baral, Emirhan Oğul, **Shanto Rahman**, August Shi, and Wing Lam. OptCD: Optimizing Continuous Development. In *International Conference on Software Engineering (ICSE'Demo)*, 2025.

[14] **Shanto Rahman**, Kishan Kumar Ganguly, and Kazi Sakib. An Improved Bug Localization using Structured Information Retrieval and Version History. In *International Conference on Computer and Information Technology (ICCIT)*, 2015.

[15] AWS Outage. `https://aws.amazon.com/message/101925`, 2025.

[16] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering (FSE)*, 2014.