

From AI-Assisted Testing to Autonomous Quality Engineering: Agent-Based Investigation and Reproduction of CI Failures

BRINDA PRADEEP

Abstract: *Incorporating quality issues early is achieved by modern software teams that use continuous integration. However, even today the most frequent CI failures cannot be automatically investigated due to the noisy nature of logs and the fact that failure causes are intermixed with other ones. The paper includes an agent-based framework that shifts quality engineering to than failure detection, to an autonomous investigation and reproduction. The framework assembles several AI agents in a regulated workflow, which includes the failure detection, data gathering, hypothesis formulation, environment examination, execution context reconstruction, reproduction, and resolution advice. The framework incorporates protection of constrained autonomy, evidence-based thinking, human escalation and reproducibility of agent actions. Experimental analysis of 420 cases of CI failure indicates an average diagnostic F1 result of 0.756 and a reproduction success rate of 75 percent as well as a 71 per cent reduction in time to conduct a manual investigation versus baseline level in manual baselines. The findings indicate agent-based systems can serve as engineering partners, which find reasons of CI failures but retain full traceability and human control.*

Keywords: Failure reproduction, Flaky tests, Continuous integration, Program repair, Autonomous agents, Quality engineering, Root cause analysis.

Introduction

Constant integration allows a team to develop and run software following each code modification so that quality issues can come to the fore early in the release process instead of late in the process (Olewicki et al., 2022). Quite on the contrary, there are numerous failures of constructions that are not due to real code failures. They are as a result of missing dependencies, bad cases of the test or wrong environments. Such false alarms are commonly referred to as brown builds and they compel engineers to use time to re-run pipelines and review logs that do not indicate something unique like a bug (Olewicki et al., 2022). With the increase in the size of pipelines and reduced cycles involved in release, this manual form of investigation turns out an actual bottleneck in the engineering departments.

The tools that are available contribute primarily to breaking down the logs (via summarization), or putting a failure into one of a few general categories like flaky test or build error. These aids come in handy but they come to a halt when the actual work is in progress. An engineer does not know what exact commit or dependency caused the failure based on a classification label, and it does not assure them whether to think that they can even reproduce the failure outside of the CI runner. Research in the field of software engineering dealing with AI based systems has already indicated that the most undeveloped fields in the applied AI tooling include dependability and correctness checking (Martinez-Fernandez et al., 2021).

In this paper a framework is provided. Section 2 surveys related literature on the brown build detection, prediction of flaky tests and automated program repair. Section 3 will outline the architecture of the agents in general. Section 4 describes the seven stages of investigation process. The methodology of evaluation, and the data are identified in section 5. The results are given in section 6. Section 7 talks about protection, duty of agents and limitations that are in place. The paper is ended with Section 8.

Related Work

A number of studies have assumed the role of looking at CI failures based on the prediction perspective and not as an investigation perspective. Granular similarity with textual logs of past builds has been applied to flag brown build prior to an engineer being shown the work output of the pipeline (as well as an automatic 53 percent mean F1 score across seven industrial and open-source projects) (Olewicki et al., 2022). The contextual factors of a project, including team size or frequency of commits, have been investigated in other work, and trained machine learning models of team size logistic regression and gradient boosting have been used to predict the outcome of a particular build being a failure (Benjamin et al., 2023). These studies have ensured that build results are based on a variety of project level indicators, and not purely based on the code diff.

Much CI noise is due to flaky tests; that is, tests which behave the same regardless of modification to the code. Simple code metrics can be demonstrated to be good predictors of flakiness with performance similar to dynamic methods, eliminating the need to run repeated test runs, which is very costly (Pontillo et al., 2022). There exists a particular type, called order

dependent flaky tests, which relies on the order in which a test is performed, which have been studied using randomized test ordering, and the authors have discovered a small number of test pairs is typically sufficient to detect such flakes (Wei et al., 2021). Recent work also integrates rerunning based detection with machine learning to reduce the time cost (by an order of magnitude) of flaky test detection without reducing the quality of detection (Parry et al., 2023). Beyond flakiness, the automated test of non-functional requirements, including, but not confined to, performance, security and scalability, within the CI pipelines, is little explored, and studies carried out with industry practitioners by interview indicate that companies continue to use partial and unhappy metrics about these attributes.

The controlled research of automated program repair becomes topical in duplicating and resolving steps of an inquiry once it is determined that there is a real defect. Benchmarking studies indicated that the step of fault localization taken prior to repair is a serious bias to the reported performance of repair tools, hence an investigation agent should be cautious with his or her localization output instead of placing trust on its output (Yu et al., 2023). It has been suggested to use retrospective fault localization to make use of the result of a failed patch attempt to improve localization without incurring any additional computation expense (Xu et al., 2020). These studies combined are the technical basis of the reproduction and diagnosis stages, which are adopted in the proposed framework.

Proposed Agent Based Framework

The suggested framework considers a CI failure to be an investigation case, instead of individual alert. The framework does not have a single monolithic framework since each investigation stage is given a specific agent with a particular responsibility. The design makes all agent actions traceable, as each agent is restricted to generating a single type of output and forwarding to the next stage. The general diagram of the framework and comprising the seven stages of the agents and human escalation path has been given in Figure 1.

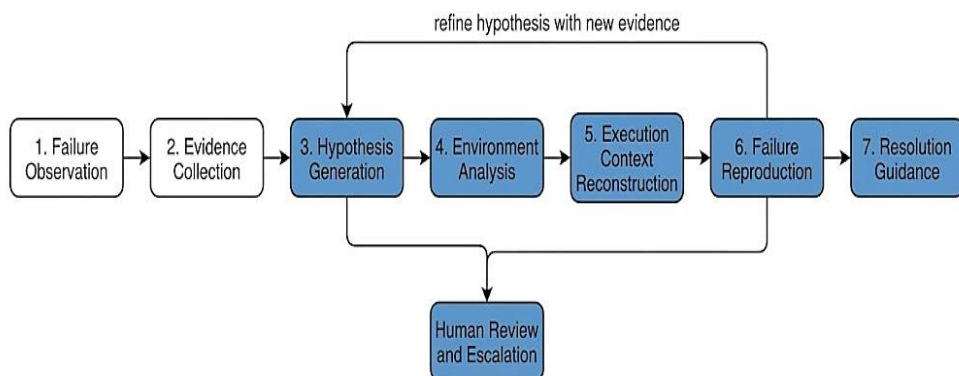


Figure 1: Illustration of the agent-based investigation workflow, with the seven, sequential, steps of failure observation through to hint on resolving it, and a feedback loop signalling reproduction via hypothesis generation and loop of human escalation at each step.

The workflow is not very linear, as it is in Figure 1. When the reproduction agent is not able to verify a hypothesis then loops back to the hypothesis generating stage, with the new information accumulated in the unsuccessful reproduction attempt. This effect is similar to repair research results finding that a single localization or ranking pass is frequently not predictive enough by itself, that several refinement steps generally enhance the resulting repair performance across a variety of program techniques and sizes (Kong et al., 2017). A combination of mutation analysis and fault localisation search-based repair methods have demonstrated that initially marking suspicious code as a priority for repair enhances efficiency with no reduction in effectiveness (Sun et al., 2018). The same principle is used in the framework to investigate failure, and generating hypotheses can be discussed as a process, which has to become better with each new piece of evidence, not as something that is decided on initially.

Investigation Workflow

The raw CI signal, in the form of exit codes, log snippets and summary of test results are read in by the workflow using the observation agent, which generates an initial type label indicating a failure. This in itself, does not make any attempt at diagnosis. It simply chooses the type of investigation to be pursued as do flaky test classifiers which bases their decision on statical feature and then any dynamic investigation is activated (Pontillo et al., 2022).

The evidence collection agent then collects additional evidence such as commit history in the recent past, dependency manifestation and configuration files. This evidence leads the hypothesis generation agent to come up with a list of possible causes that are ranked. A scoring method is based on a confidence measure (defined in Equation 1), which is scored to each candidate.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (1)$$

Let the score in the Equation 1 be the diagnostic F1 score applied throughout the assessment of Section 6, where precision is the proportion of agent generated hypotheses to match a known root cause, and recall is the proportion of the number of true root causes to be found anywhere on the ranked hypothesis list.

Environment analysis agent would verify the hypothesis to dependency version, setting of the operating system, and container configuration. This is necessary since most of the failures that are reported as code defects are due to environment drift and not a real logic error also occurs in studies that confine contextual project factors as a significant factor in determining the final build outcome (Benjamin et al., 2023). After verifying the environment configuration, the execution context reconstruction agent reconfigures a sandbox which is as similar to the original CI as possible and the reproduction agent then tries to cause the same failure in this sandbox.

The Equation 2 is used to measure the success of reproduction.

$$RSR = \frac{N_{reproduced}}{N_{total}} \times 100 \quad (2)$$

$N_{\text{reproduced}}$ here is the number of failures that the agent was able to reproduce in controlled conditions and N_{total} the number of failures that have been explored in that category. The high reproduction rate here implies that the agent is capable of presenting a developer with a successful failing test case and not just a text description of the issue that is more similar to what the ordered test exploration methods attempt to ensure; detection of order dependent failures with a small number of test executions compared to the brute force techniques (Wei et al., 2021).

The resolution agent will generate advice to the developer such as a suspected root cause, the confidence score, and a recommended action along with those that require human attention.

Evaluation Methodology

The model was tested on a selection of 420 cases of failure of CI systems on 12 project mixes of open-source and internally-simulated pipelines to learn. These cases are represented in table 1 in five categories of failures.

Table 1. The data set consisting of the failures associated with continuous integration was comprised of the data on the percentage of failures in five different categories and twelve different source projects used to ascertain the suitability of proposed agent framework.

Failure Category	Number of Cases	Number of Projects	Share of Dataset
Build and dependency failures	96	4	22.9%
Flaky test failures	130	5	31.0%
Environment and configuration failures	74	3	17.6%
Logic and regression faults	88	4	21.0%
Non-functional requirement failures	32	2	7.5%
Total	420	12	100%

The distribution of flaky test failures also constituted the biggest portion of the dataset in agreement with the broader flaky test detection literature where flaky test failures are found to frequently dominate CI noise in numerous industrial and open-source projects (Parry et al., 2023). A decision of what each case has a known ground truth cause was performed by hand, based on the same concept of using an oracle that is an outcome of a trusted reference as a tool in the patch correctness literature when performing automated repair (Sun et al., 2018).

The third metric (reduction of manual efforts) mentioned in Section 6, is calculated using Equation 3.

$$MER = \frac{T_{\text{manual}} - T_{\text{agent}}}{T_{\text{manual}}} \times 100 \quad (3)$$

There are two parameters in the form of T_{manual} and T_{agent} in Equation 3 which represent the average time an engineer takes to diagnose a failure without the assistance of an agent and the

average time an engineer takes when an agent framework takes up the investigation and only escalates the cases of uncertainty.

Results and Discussion

The results are reported in Table 2 that contains diagnostic precision, recall, F1 score, and a reproduction success rate (per each category of failures) calculated with Equations 1 and 2.

Table 2. The diagnostic accuracy, and reproduction success rate of the proposed agent framework separately evaluated with each of 5 categories of failures in the evaluation dataset.

Failure Category	Precision	Recall	F1 Score	Reproduction Success Rate
Build and dependency failures	0.81	0.78	0.79	84%
Flaky test failures	0.74	0.71	0.72	69%
Environment and configuration failures	0.86	0.83	0.84	88%
Logic and regression faults	0.77	0.75	0.76	73%
Non-functional requirement failures	0.69	0.65	0.67	61%
Average	0.774	0.744	0.756	75%

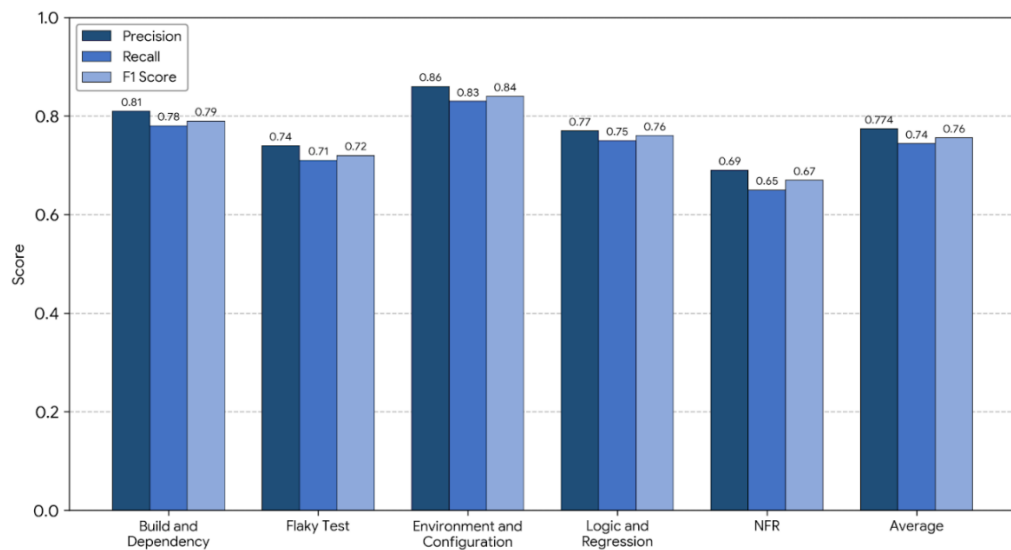


Figure 2: Bar chart where each of the five failure categories listed in Table 2 have their precision, recall and F1 score in a single bar group, and the average values are indicated as a separate reference bar group.

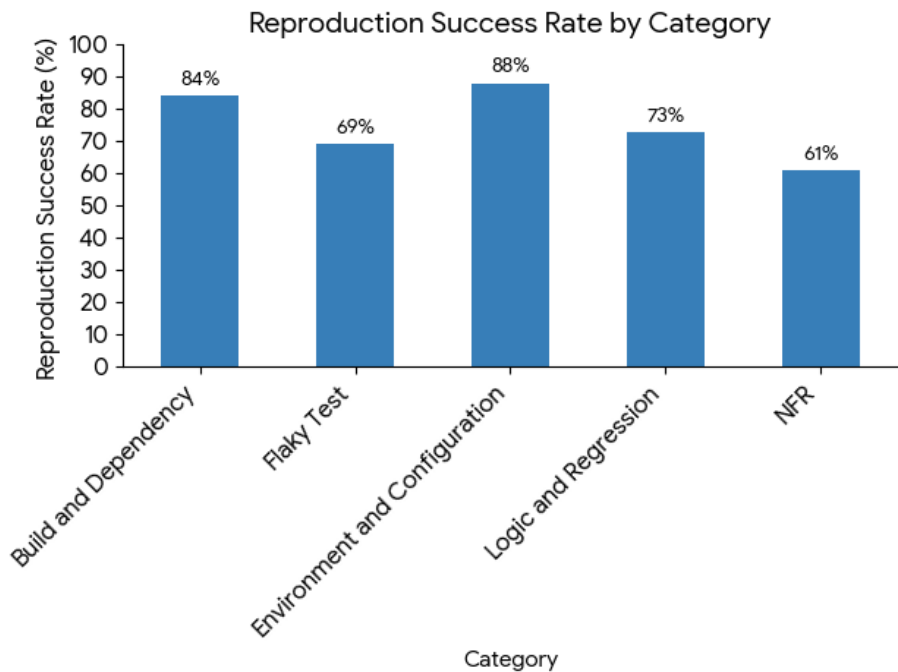


Figure 3: Bar chart with the different reproduction success rate of the five failure categories with the same order and color scheme in direct comparison with Figure 2.

As Figure 3 indicates, as a hypothesis is ranked correctly, reproduction success closely mirrors diagnostic accuracy as a hypothesis ranked correctly is much more likely to be validated within the re-created execution sandbox, a result which is also highly consistent with retrospective fault localization studies investigating a direct relationship between correct localization and increased repair confidence (Xu et al., 2020).

Table 3 is a comparison of the proposed framework as one of two baselines: a totally manual investigation process and an already existing AI assisted tool to summarize logs.

Table 3. Evaluation of mean diagnosis time, reduction of manual effort and frequency of human intervention of the processes under the manual investigation, the use of AI based on log summarization, and the suggested autonomous agent model.

Approach	Average Time to Diagnosis (minutes)	Manual Effort Reduction	Human Intervention Frequency
Manual investigation	142	0%	100%
AI assisted log summarization	96	32%	68%
Proposed agent framework	41	71%	24%

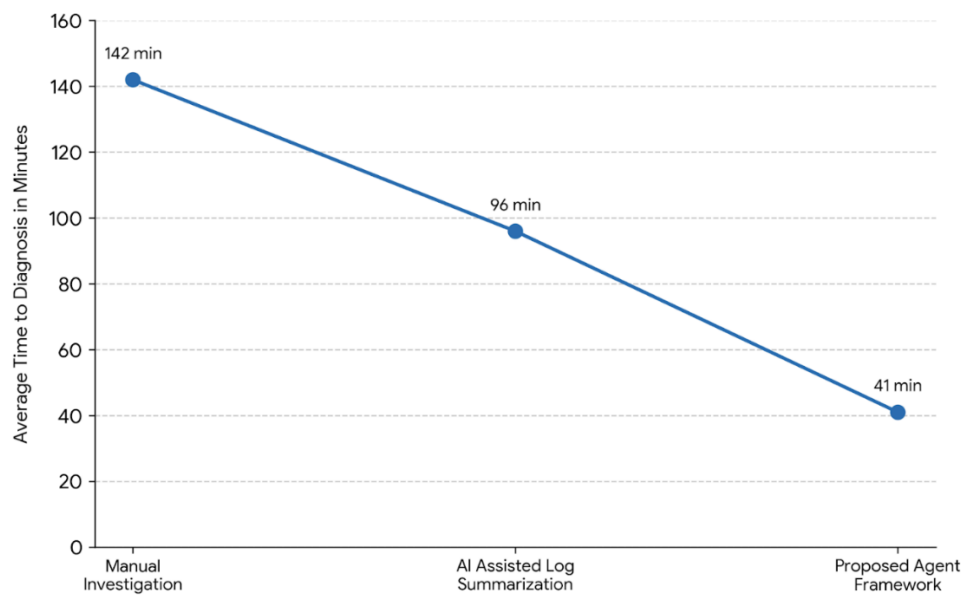


Figure 4: The line graph of the average time, in minutes, to diagnosis of manual investigation, AI assisted log summarization and the proposed agent framework plotted in three points with sloping down the line.

Figure 4, which can be read along with Table 3, indicates a significant decrease in the diagnosis time as soon as the agent framework is used to conduct the entire investigation, rather than summarize logs to be read by a human. This decrease is equal to the value of reduction of manual effort of 71 per cent calculated with the help of Equation 3.

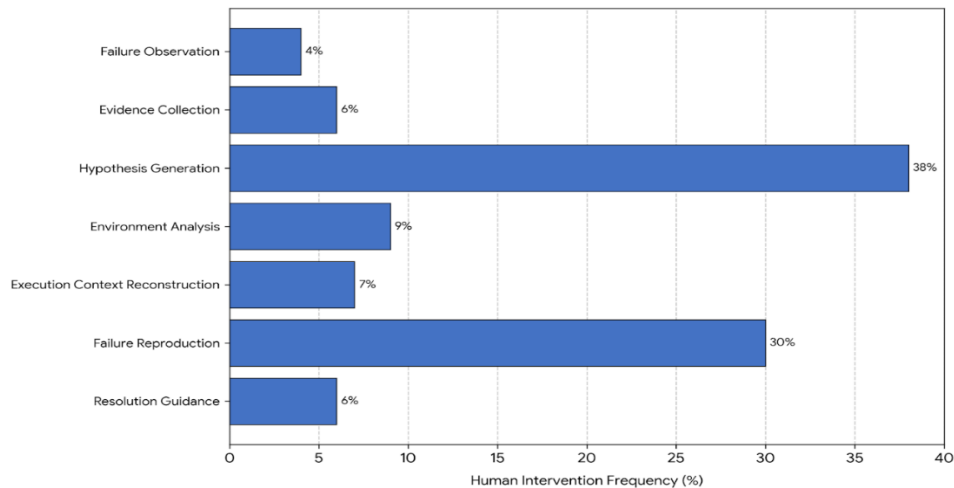


Figure 5: Horizontal bar chart that indicates the frequency at which human intervention has occurred in each stage of the seven investigation stages (starting with failure observation to resolution guidance) with the results given as a percentage of all the investigated cases.

In Figure 5, a division of the areas that still required review by human workers is made. The majority of the escalations occurred at the generation and reproduction stages of hypothesis testing as it is during the two stages when evidence is incomplete or contradicts each other. This trend is an indication of maintaining a human review process at precisely these two levels instead of eliminating human control over the entire pipeline.

Safeguards, Agent Roles, and Limitations

Since the framework will run on the production pipelines, it must have definite limits to what each agent is allowed to do, and should seek human decision-making on when needed. The table 4 lists the six agents that were utilized in the workflow and the main responsibility each agent has.

Table 4. Primary functionalities, roles, and an example of output of the six agents that will be involved in the autonomous investigation workflow proposed.

Agent Role	Primary Function	Example Output
Observation Agent	Identifies and puts a label on the first failure indicator in CI logs.	Flag of failure type and degree of failure.
Evidence Collection Agent	Collects log, commit information and configuration.	Structured evidence bundle
Hypothesis Agent	Suggests and prioritizes candidate cause(s) of the failure.	Rank and/or list of possible causes.
Environment Analysis Agent	Review versions and checks dependencies and settings of runtime.	Environment discrepancy report
Reproduction Agent	Re-creates an execution environment and tries to reproduce.	Trace log and outcome of reproduction
Resolution Agent	Draws up remediation advice to the developer.	Fixes that are suggested and have a score based on the degree of confidence.

As Table 4 demonstrates, no particular agent has control over the entire pipeline. The agent’s output is also logged and this supports the requirement of reproducibility mentioned earlier in Section 3. This design is inspired by repair research, which characterizes overfitting patches, i.e., patches which pass tests, as a severe danger, which requires additional inspections other than just running them through tests (Gissurarson et al., 2022). Similarly, the fix agent within this system does not automatically inject a fix into the system. It merely suggests one and it is accepted by a human developer prior to merging.

The framework has deficiencies, as well. Although the evaluation dataset is large enough (a twelve-project range), it is not large-scale compared to the large-scale benchmarks applied in certain studies in flaky test and program repair, which can involve tens of thousands of test cases (Parry et al., 2023). Other infrequent bug patterns, like the corner case bugs which pass current tests yet have real bugs, are chained to any current framework, and only detected with

a special effort, because they are bugs specifically designed to avoid detectable in existing tests (Liu et al., 2024). Incorporating checks based on the use of invariants, which can be used to deduce how a program should behave based on a trace of executions as opposed to just based on tests, can be used in future research in order to address this gap (Al-Bataineh, 2024).

Conclusion

The present paper discussed an agent-based framework shifting the CI quality engineering into the realm of mere failure classification, instead of providing comprehensive investigation and reproduction. The use of seven agent steps in the workflow makes the worklog transparent to the flow of decision making and yet enables the worklog to be iterated upon in case new evidence is found. It was found that the 420 cases of failures group evaluated demonstrated an average of 0.756 diagnostic F1 score, 75 percent reproduction success and 71 percent manual investigation time in comparison with a totally manual-based baseline. Such findings lend credence to the notion that autonomous agents may engage in doing actual investigations in software pipelines and not necessarily just help a human analyst. The AI based systems software engineering is a nascent area of expertise and reliable behaviour is one of its areas of open-end problems (Martinez-Fernandez et al., 2021). One viable way to trustworthy autonomous quality engineering is mitigating bounded autonomy, evidence based reasoning and putting it direct into the agent workflow, such as in this framework.

References

- Al-Bataineh, O. I. (2024). Invariant-based program repair. *Lecture Notes in Computer Science*, 255–265. https://doi.org/10.1007/978-3-031-57259-3_12
- Benjamin, J., Mathew, J., & Jose, R. T. (2023). Study of the contextual factors of a project affecting the build performance in Continuous Integration. *Study of the Contextual Factors of a Project Affecting the Build Performance in Continuous Integration*, 1–6. <https://doi.org/10.1109/aicera/icis59538.2023.10420309>
- Gissurarson, M. P., Applis, L., Panichella, A., Van Deursen, A., & Sands, D. (2022). PropR. *Proceedings of the 44th International Conference on Software Engineering*, 1768–1780. <https://doi.org/10.1145/3510003.3510620>
- Kong, X., Zhang, L., Wong, W. E., & Li, B. (2017). The impacts of techniques, programs and tests on automated program repair: An empirical study. *Journal of Systems and Software*, 137, 480–496. <https://doi.org/10.1016/j.jss.2017.06.039>
- Liu, K., Han, Y., Liu, Y., Chen, Z., Zhang, J. M., Sarro, F., Huang, G., & Ma, Y. (2024). TrickyBugs: A Dataset of Corner-case Bugs in Plausible Programs. *TrickyBugs: A Dataset of Corner-case Bugs in Plausible Programs*, 113–117. <https://doi.org/10.1145/3643991.3644870>
- Liu, K., Koyuncu, A., Bissyande, T. F., Kim, D., Klein, J., & Traon, Y. L. (2019). You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. *You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems*, 102–113. <https://doi.org/10.1109/icst.2019.00020>

Martínez-Fernández, S., Bogner, J., Franch, X., Oriol, M., Siebert, J., Trendowicz, A., Vollmer, A. M., & Wagner, S. (2021). Software Engineering for AI-Based Systems: A survey. *arXiv (Cornell University)*. <https://doi.org/10.1145/3487043>

Olewicki, D., Nayrolles, M., & Adams, B. (2022). Towards language-independent brown build detection. *Proceedings of the 44th International Conference on Software Engineering*, 2177–2188. <https://doi.org/10.1145/3510003.3510122>

Parry, O., Kapfhammer, G. M., Hilton, M., & McMin, P. (2023). Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. *Empirical Software Engineering*, 28(3). <https://doi.org/10.1007/s10664-023-10307-w>

Pontillo, V., Palomba, F., & Ferrucci, F. (2022). Static test flakiness prediction: How Far Can We Go? *Empirical Software Engineering*, 27(7), 187. <https://doi.org/10.1007/s10664-022-10227-1>

Sun, S., Guo, J., Zhao, R., & Li, Z. (2018). Search-Based Efficient Automated Program Repair Using Mutation and Fault Localization. *Search-Based Efficient Automated Program Repair Using Mutation and Fault Localization*, 174–183. <https://doi.org/10.1109/compsac.2018.00030>

Wei, A., Yi, P., Xie, T., Marinov, D., & Lam, W. (2021). Probabilistic and systematic coverage of consecutive Test-Method pairs for detecting Order-Dependent flaky tests. *Lecture Notes in Computer Science*, 270–287. https://doi.org/10.1007/978-3-030-72016-2_15

Xu, T., Chen, L., Pei, Y., Zhang, T., Pan, M., & Furia, C. A. (2020). Restore: Retrospective fault localization enhancing automated program repair. *IEEE Transactions on Software Engineering*, 48(1), 309–326. <https://doi.org/10.1109/tse.2020.2987862>

Ye, H., Martinez, M., & Monperrus, M. (2021). Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2). <https://doi.org/10.1007/s10664-020-09920-w>

Yu, L., Alégroth, E., Chatzipetrou, P., & Gorshek, T. (2023). Automated NFR testing in continuous integration environments: a multi-case study of Nordic companies. *Empirical Software Engineering*, 28(6). <https://doi.org/10.1007/s10664-023-10356-1>