IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.311 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 11, November 2026

Machine Learning-Based Automated Software Failure Diagnosis and Root Cause Localization

Ananya Anand[1], S Poornima[2]

Department of Computer Science (Emerging Technologies), SRMIST, Vadapalani, Chennai, India[1]

Department of Computer Science (Emerging Technologies), SRMIST, Vadapalani, Chennai, India[2]

Abstract: Diagnosing the root cause of a software failure is a time-consuming task that requires developers to correlate failure reports, stack traces, source code and change history. This paper presents FaultTrace, a machine learning framework that ranks candidate source files by their likelihood of containing the root cause of a failure. Using real bugs from the Defects4J benchmark, we label the files modified by a bug's verified fix as weak positives and files modified by other bugs' fixes as weak negatives, extract five lightweight features (file size in lines, method count, class count, import count and a stack-trace match flag) and rank candidates with a Random Forest under leave-one-bug-out cross-validation. Only the Lang and Math projects yielded candidates that resolved in their own buggy-revision checkouts, giving 480 usable rows and 149 evaluable bugs. Source-structure features alone reach a Mean Reciprocal Rank (MRR) of 0.923 and a Recall@1 of 0.866; adding the stack-trace flag raises these to 0.950 and 0.906. Ablation and permutation importance show that file-size features dominate, while lexical-overlap features reduce accuracy. Most errors are near misses at rank 2. We discuss the limits of the weak-label design and of the small project sample, and outline extensions with harder negatives, more projects and semantic features.

Index Terms: Software fault localization, root cause analysis, machine learning, random forest, Defects4J, mean reciprocal rank.

I. Introduction

Software failures in complex applications can arise from interactions between several components, which makes identifying their underlying causes a difficult and time-consuming task. A developer who receives a failure report typically examines the failure message and stack trace, the source code and the history of changes before forming a hypothesis about the defect responsible, and fault localization is widely regarded as one of the most expensive parts of debugging [1].

Automated techniques have attacked this problem from several directions. Information-retrieval methods rank source files by their textual similarity to a bug report [2], [3], learning-based methods combine heterogeneous evidence such as spectrum, mutation and complexity information [4], transfer-learning methods reuse knowledge across projects [5], and recent work applies large language models to localization and explanation [6], [7]. These approaches are powerful, but many depend on expensive dynamic analysis, on high-quality bug-report text or on large neural models that are difficult to inspect and to reproduce. It is therefore useful to know how much of the problem can be solved with cheap, statically available and human-readable evidence.

This paper studies exactly this question. We build FaultTrace, a pipeline that ranks the candidate files of a failing program with a Random Forest [8] over five lightweight features. Ground truth comes from Defects4J [9], a benchmark of real faults, which is important because techniques evaluated on artificial faults may not transfer to real ones [10]. The contributions of the paper are:

- A weakly labelled candidate dataset in which files changed by a bug's verified fix are positives and files changed by other bugs' fixes are negatives.
- A compact, interpretable feature set and a leave-one-bug-out evaluation protocol that prevents leakage between the rows of a bug.
- An empirical study on 149 bugs showing that source-structure features reach an MRR of 0.923 and that adding a stack-trace flag raises it to 0.950.
- An ablation, permutation-importance and error analysis, together with an explicit account of the limitations of the weak-label design and of the data.

The rest of the paper is organised as follows. Section II reviews related work, Section III describes the method, Section IV the experimental setup, Section V the results and Section VI concludes.

II. Related Work

A. Fault and Bug Localization

Wong *et al.* [1] survey the field and organise techniques into spectrum-based, mutation-based, slicing-based, learning-based and retrieval-based families. In the retrieval family, BugLocator [2] ranks files by the similarity between a bug report and the code using a revised vector space model and also uses similar past reports and file size. HyLoc [3] adds a

deep neural network that learns the relation between report terms and code tokens, and Legion [11] composes many BugLocator variants with a Random Forest in an industrial setting, reporting a 36.4% improvement over BugLocator.

B. Learning-Based and Cross-Project Approaches

DeepFL [4] integrates spectrum, mutation, complexity and textual features in a deep model for method-level localization on Defects4J. TRANP-CNN [5] transfers semantic features across projects with a convolutional network, placing the buggy file first for 29.9% of bugs. RLocator [12] casts localization as a Markov decision process to optimise ranking metrics directly and reports an MRR of 0.62 on 8,316 bug reports, while BLAZE [13] uses dynamic chunking and hard-example fine-tuning for cross-project and cross-language localization. At the change level, DeepJIT [14] predicts defect-inducing commits with convolutional networks.

C. LLM-Based Localization and Root Cause Analysis

LLMAO [6] fine-tunes bidirectional adapters on top of code language models to find buggy lines without test coverage, and AutoFL [7] lets an LLM navigate a repository through function calls and produces an explanation with each suggestion. Beyond source files, Ahmed *et al.* [15] study LLMs for recommending root causes and mitigations for more than 40,000 cloud incidents.

D. Gap

Recent systems rely on large neural models, which limits interpretability and reproducibility, and papers seldom isolate the contribution of simple features or analyse how the choice of negative examples affects what is learned. FaultTrace addresses this with a lightweight, transparent pipeline, a systematic ablation and an honest treatment of weak labels. Its candidate pools are small, so its metrics are not comparable with those of large-pool systems.

III. METHODOLOGY

A. Problem Formulation

For each bug *b* there is a pool *C(b)* of candidate files taken from its buggy-revision checkout. The subset *C+(b)* consists of files modified by the verified fix of *b*. The task is to learn a scoring function that ranks members of *C+(b)* above the remaining candidates.

B. Candidate Construction and Weak Labels

Every (bug, file) pair is a candidate row. A row is a *verified patch candidate* (label 1) if the file was modified by the fix of that bug, and a *weak negative* (label 0) if the file was modified by the fix of a different bug and is unrelated to the current one. The labels are weak because a fixed file is only a proxy for the root cause and because a negative is merely "not this bug's fix": it may itself be defective. Features are computed from the file as it exists in the bug's own buggy-revision checkout, and a candidate that cannot be found there is discarded.

C. Features

Five features are used (Table I). All are cheap, static and human-readable. File size and method count are proxies for the amount of code in which a fault may reside and are well-known correlates of defect-proneness; BugLocator [2] also uses file size. Import count is a rough measure of coupling. The stack-trace flag is the only failure-related feature. Method and class counts are obtained with regular-expression heuristics rather than a Java parser, so they are approximate.

TABLE I
FEATURES USED BY FAULTTRACE

Feature	Description
candidate_line_count	File size in lines
candidate_method_count	Methods in file (regex heuristic)
candidate_class_count	Classes in file (regex heuristic)
candidate_import_count	Number of import statements
test_stack_match_fixed	1 if the file's class appears in the bug's stack-trace locations, else 0

Two exploratory lexical-overlap features, failure_source_token_overlap and exception_source_token_overlap, were examined only in the ablation study and are not part of the final model.

D. Random Forest Ranking

A Random Forest [8] averages many decision trees, each trained on a bootstrap sample with a random subset of features considered at each split, which reduces variance relative to a single tree. We use scikit-learn [16] with 100 trees, random_state=42 and all available cores, preceded by a StandardScaler. For a held-out bug, candidates are sorted by the predicted probability of the positive class, and the position of a true fix file is its rank. Standardisation does not alter a tree ensemble's ranking but keeps the pipeline uniform for scale-sensitive learners.

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.311 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 11, November 2026

E. Validation Protocol and Metrics

Rows of the same bug share a failure, a stack trace and fix files, so random row-level folds would leak information. We therefore use *leave-one-bug-out* cross-validation: for each bug, the model is trained on all other bugs and scores only the held-out bug. Let *rank(b)* be the position of the highest-ranked true file of bug *b* and *B* the evaluated bugs. We report

$$MRR = (1/|B|) \Sigma\ 1/rank(b) \quad (1)$$

$$Recall@k = |\{b \in B : rank(b) \leq k\}| / |B| \quad (2)$$

Recall@1 requires the first suggestion to be correct.

IV. Experimental Setup

A. Dataset

The data come from Defects4J [9], which covers 17 Java projects: Chart, Cli, Closure, Codec, Collections, Compress, Csv, Gson, JacksonCore, JacksonDatabind, JacksonXml, Jsoup, JxPath, Lang, Math, Mockito and Time. In our environment, 164 bugs had usable checkouts and generated 5,367 candidate rows. Only Lang (390 rows) and Math (631 rows) contained candidates that fully resolved in their own bug's buggy-revision checkout, which yielded 480 usable rows and 149 evaluable bugs (Lang 57, Math 92). This restriction stems from the candidate-resolution pipeline, since file paths change between revisions, and not from the model. Table II and Fig. 1 summarise the data.

TABLE II
DATA FUNNEL

Stage	Count
Defects4J projects considered	17
Bugs with usable checkouts	164
Candidate rows generated	5,367
Projects with resolving candidates	2 (Lang, Math)
Candidate rows: Lang / Math	390 / 631
Usable rows for feature extraction	480
Evaluable bugs: Lang / Math / total	57 / 92 / 149

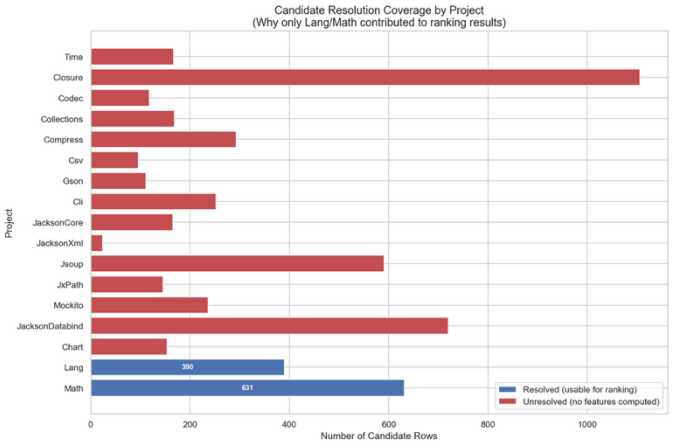


Fig. 1. Dataset coverage by project.

B. Configurations Compared

We compare (i) a baseline that relies on the stack-match signal alone, which is degenerate in this dataset and does not separate positives from negatives; (ii) the four source-structure features; and (iii) the source-structure features plus the stack-match flag, which is the final model. An ablation additionally evaluates single features, the two exploratory overlap features and the seven-feature full set, and permutation importance is computed on the full set.

C. Implementation

The pipeline is written in Python with scikit-learn [16], pandas and NumPy, using the Defects4J tooling to obtain buggy-revision checkouts. All features are deterministic and the forest is seeded, so results are reproducible.

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.311 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 11, November 2026

V. Results and Discussion

A. Overall Performance

Table III and Fig. 2 give the main results. The degenerate stack-match baseline reaches an MRR of 0.341 and a Recall@1 of 0.000. Source-structure features alone raise these to 0.923 and 0.866, and adding the stack-match flag gives 0.950 and 0.906, an additional gain of 0.027 in MRR (about 2.9% relative) and 0.040 in Recall@1 (about 4.6% relative). Thus the first suggestion is a true fix file for roughly nine of ten bugs.

TABLE III
PERFORMANCE BY FEATURE SET (149 BUGS)

Feature set	MRR	Recall@1
Baseline (degenerate stack-match)	0.341	0.000
Source structure only	0.923	0.866
Source structure + stack-match	**0.950**	**0.906**

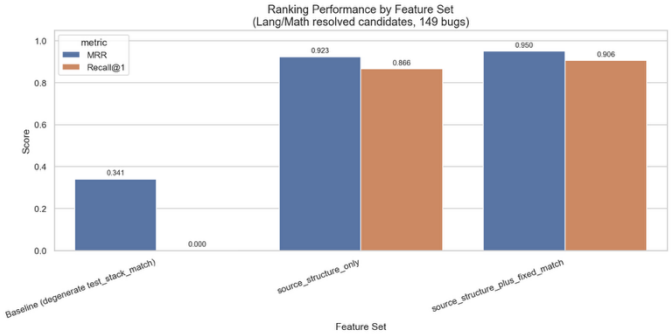


Fig. 2. MRR and Recall@1 across feature sets.

Absolute values must be read against the pool size. Pools are small (about three candidates per bug for correctly ranked bugs and about five for missed ones). For a single-positive pool of n candidates, a random ordering has an expected MRR equal to the mean of 1, 1/2, ..., 1/n and a Recall@1 of 1/n, which gives about 0.61 and 0.33 for n = 3 and 0.46 and 0.20 for n = 5 (idealised illustrations, not measurements). The degenerate baseline lies below these levels and should be seen as a degenerate floor rather than chance. Fig. 3 shows the rank distribution of the true file and the Recall@k curve; most true files are at rank 1 and nearly all remaining ones at rank 2.

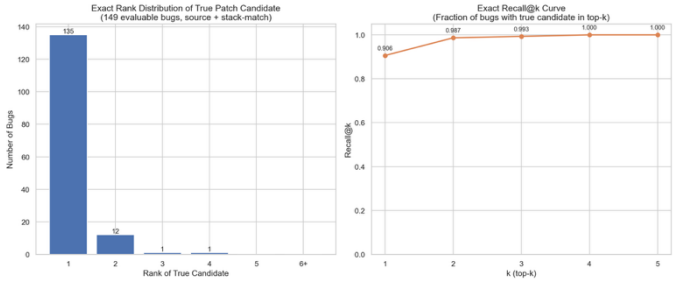


Fig. 3. Rank distribution and Recall@k curve.

B. Per-Project Breakdown

Table IV shows that performance is high in both projects, with Math slightly above Lang (by 0.043 in MRR and 0.034 in Recall@1), as also shown in Fig. 4. With two projects and few bugs the difference may reflect chance, training-set size or code-base structure, and the data cannot tell these apart.

TABLE IV
PER-PROJECT RESULTS (SOURCE STRUCTURE ONLY)

Project	Bugs	MRR	Recall@1
Lang	57	0.883	0.825
Math	92	0.926	0.859

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.311 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 11, November 2026

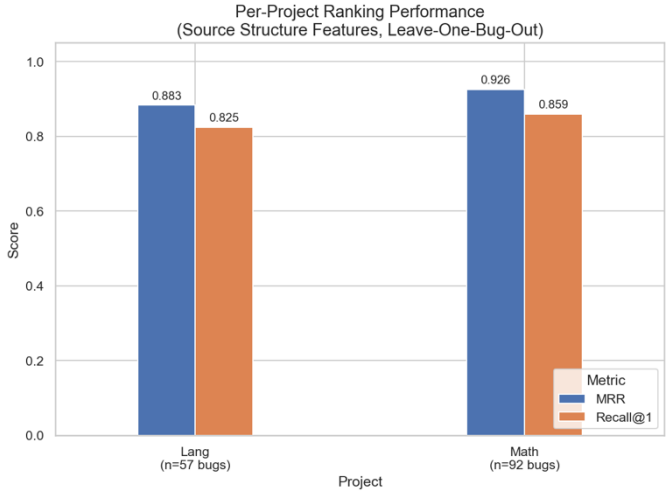


Fig. 4. Per-project breakdown for Lang and Math.

C. Feature Ablation

Table V and Fig. 5 report the ablation. File size dominates: the line count alone reaches an MRR of 0.871 and a Recall@1 of 0.792, and the method count alone 0.842 and 0.745, close to the four-feature structure set (0.923 / 0.866). Class count (0.570 / 0.289) and import count (0.699 / 0.483) are weak alone. The overlap features alone are poor (0.587 / 0.302), and adding them to the structure features *lowers* performance to 0.899 / 0.805. The full set reaches 0.900 / 0.805, which is 0.050 in MRR and 0.101 in Recall@1 below structure plus stack-match. With few training bugs, noisy features can mislead the forest.

TABLE V
FEATURE ABLATION (MRR / RECALL@1)

Configuration	MRR	Recall@1
Line count only	0.871	0.792
Method count only	0.842	0.745
Class count only	0.570	0.289
Import count only	0.699	0.483
All source structure	**0.923**	**0.866**
Overlap only	0.587	0.302
Structure + overlap	0.899	0.805
Full set	0.900	0.805

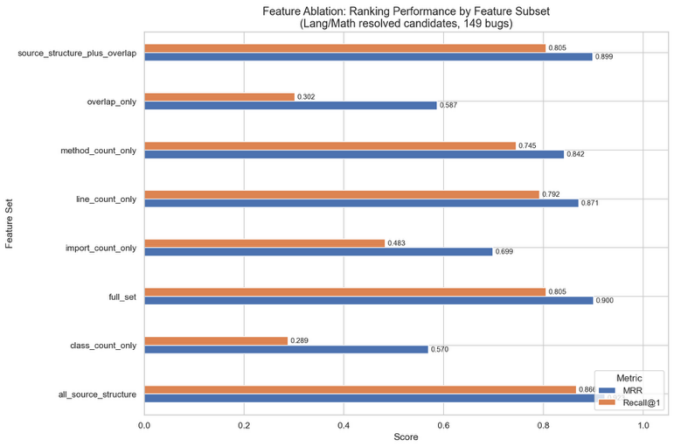


Fig. 5. Feature ablation results.

D. Permutation Importance

Table VI lists permutation importance on the full set. Method count (0.108) and line count (0.104) are most important, followed by import count (0.072); the overlap features, the stack-match flag and the class count are near zero, and two

are slightly negative, meaning that shuffling them marginally improved performance, i.e., they behave as noise. Two caveats apply. The stack-match flag has zero importance in the full set although it improves MRR from 0.923 to 0.950 when added to the structure features, so its benefit may be masked by the noisy overlap features. Moreover, line and method counts are correlated size measures, so permuting one is partly compensated by the other and the importance of each is likely understated.

TABLE VI
PERMUTATION IMPORTANCE (FULL SET)

Feature	Importance
candidate_method_count	0.108
candidate_line_count	0.104
candidate_import_count	0.072
exception_source_token_overlap	0.003
test_stack_match	0.000
candidate_class_count	−0.009
failure_source_token_overlap	−0.024

E. Error Analysis

About 14 unique bugs had a true candidate ranked below first, accounting for 20 candidate rows because some bugs, such as Math::6, have several true candidates. Table VII and Fig. 6 compare these bugs with the rest. Their true files are smaller (742 versus 1,066 lines on average) and their candidate pools larger (5.3 versus 3.1 candidates). Both observations are consistent with a model that has learned that larger files are more likely fix files: a small fix file is easily outranked by a larger distractor, and a larger pool offers more distractors. Most misses are at rank 2, so the errors are near-misses and the model degrades gracefully rather than failing randomly; a developer inspecting the top two or three suggestions would lose little.

TABLE VII
MISSED BUGS VERSUS OTHER BUGS

Characteristic	Missed	Other
Mean true-file size (lines)	742	1,066
Mean candidates per bug	5.3	3.1

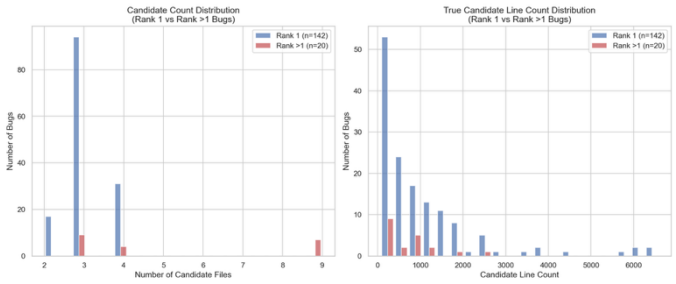


Fig. 6. Error analysis histograms.

F. Interpretation and Limitations

The evidence indicates that the model has learned a size-based prior, supplemented modestly by stack-trace evidence, and has not been shown to reason about why a failure occurs. Five limitations qualify the results. (1) Only two of 17 projects had usable data, owing to the candidate-resolution pipeline rather than the model. (2) Negatives are other bugs' patch files rather than random non-buggy files, which makes labels weak and permits size- and file-identity shortcuts, so accuracy may be overstated for open-ended localization. (3) With 149 bugs, a normal-approximation 95% interval for a Recall@1 of 0.906 is about ±0.05, so small differences should not be over-interpreted. (4) Method and class counts come from regular expressions rather than a Java parser. (5) Training and test bugs come from the same two projects, so project-level overfitting is only partly mitigated by the per-project breakdown.

VI. Conclusion and Future Work

We presented FaultTrace, a lightweight machine learning pipeline that ranks candidate files for software failures on real Defects4J bugs. A Random Forest over source-structure features reaches an MRR of 0.923 and a Recall@1 of 0.866 under leave-one-bug-out validation, and adding a stack-trace flag raises these to 0.950 and 0.906. File-size features carry

most of the signal, lexical overlap does not help, and errors are mostly near-misses at rank 2. The results show that cheap, interpretable evidence can help narrow down where a fault lies, but they cover only two projects and weak labels, so they do not yet demonstrate generalisation.

Future work will (1) extend candidate resolution to more projects, through fuzzy path matching or by sampling negatives from files present in each bug's checkout; (2) use harder negatives such as random non-patch source files; (3) replace lexical overlap with semantic AST-based and embedding-based features; and (4) scale to more projects for stronger generalisation claims, including leave-one-project-out validation.

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[2] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Zurich, Switzerland, 2012, pp. 14–24.

[3] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Lincoln, NE, USA, 2015, pp. 476–481.

[4] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing and Analysis (ISSTA)*, Beijing, China, 2019, pp. 284–295, doi: 10.1145/3293882.3330574.

[5] X. Huo, F. Thung, M. Li, D. Lo, and S.-T. Shi, "Deep transfer bug localization," *IEEE Trans. Softw. Eng.*, vol. 47, no. 7, pp. 1368–1380, Jul. 2021, doi: 10.1109/TSE.2019.2920771.

[6] A. Z. H. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE)*, Lisbon, Portugal, 2024, doi: 10.1145/3597503.3623342.

[7] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of LLM-based explainable fault localization," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1424–1446, 2024, doi: 10.1145/3660771.

[8] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[9] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing and Analysis (ISSTA)*, San Jose, CA, USA, 2014, pp. 437–440.

[10] S. Pearson *et al.*, "Evaluating and improving fault localization," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, Buenos Aires, Argentina, 2017, pp. 609–620.

[11] D. Jarman, J. Berry, R. Smith, F. Thung, and D. Lo, "Legion: Massively composing rankers for improved bug localization at Adobe," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 3010–3024, Aug. 2022, doi: 10.1109/TSE.2021.3075215.

[12] P. Chakraborty, M. Alfadel, and M. Nagappan, "RLocator: Reinforcement learning for bug localization," *IEEE Trans. Softw. Eng.*, vol. 50, no. 10, pp. 2695–2708, Oct. 2024, doi: 10.1109/TSE.2024.3452595.

[13] P. Chakraborty, M. Alfadel, and M. Nagappan, "BLAZE: Cross-language and cross-project bug localization via dynamic chunking and hard example learning," *IEEE Trans. Softw. Eng.*, vol. 51, no. 8, pp. 2254–2267, Aug. 2025, doi: 10.1109/TSE.2025.3579574.

[14] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, Montreal, QC, Canada, 2019, pp. 34–45.

[15] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "Recommending root-cause and mitigation steps for cloud incidents using large language models," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, Melbourne, Australia, 2023.

[16] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

 This work is licensed under a Creative Commons Attribution 4.0 International License