10.48047/jocaaa.2024.33.07.81

AN ANALYTICAL STUDY ON CHALLENGES IN SOFTWARE FAULT LOCALISATION

Ajeet Kumar

Research Scholar, Department of Computer Application (under department of Mathematics)
B R A Bihar University, Muzaffarpur
Dr. I B Lal

Senior Assistant Professor & HOD, Department of Computer Applications, L N Mishra College of Business Management, Muzaffarpur, Bihar

Received 2 May 2024; Revised 15 June 2024; Accepted 3 October 2024

Abstract

Software Fault Localisation (SFL) is a critical, yet notoriously challenging, phase in the software debugging process, aimed at identifying the exact code locations responsible for failures. Despite decades of research and the development of numerous automated techniques—from spectrum-based and statistical debugging to modern learning-based approaches—SFL remains a significant bottleneck in software engineering practice. This paper synthesises current research to provide a comprehensive analysis of the persistent challenges in SFL. We systematically examine technical limitations, including the inherent complexity of program spectra, the issue of coincidental correctness, and the confounding effects of program dependencies. We further explore practical and pragmatic hurdles such as scalability to modern systems, integration into development workflows, and practitioner expectations. The analysis extends to emerging challenges and opportunities presented by the integration of Artificial Intelligence (AI), Machine Learning (ML), and particularly Large Language Models (LLMs) into SFL processes. By integrating findings from foundational and contemporary literature, this paper presents a consolidated view of the SFL landscape, complete with comparative tables and synthesised data visualisations. It concludes by outlining a roadmap for future research, emphasising hybrid techniques, improved benchmark datasets, explainable AI for SFL, and the need for real-world, human-centric evaluations to bridge the gap between academic research and industrial practice.

Keywords: Software Fault Localisation, Automated Debugging, Spectrum-Based Fault Localisation, Machine Learning, Large Language Models, Software Testing, Software Maintenance, Empirical Software Engineering.

1. Introduction

The cost of software defects is monumental. The National Institute of Standards and Technology (NIST) estimated that inadequate infrastructure for software testing costs the U.S. economy billions annually, with a significant portion attributed to the labor-intensive process of finding

10.48047/jocaaa.2024.33.07.81

and fixing bugs (Planning, 2002). Within the software development lifecycle, debugging—and specifically, fault localisation—is where developers spend a disproportionate amount of time. Software Fault Localisation (SFL) is the process of automatically or semi-automatically pinpointing the exact statements, functions, or modules responsible for observed failures, thereby reducing the manual effort required from developers (Wong et al., 2016).

Over the years, a plethora of automated SFL techniques have been proposed. Early seminal work introduced dynamic approaches like Tarantula (Jones & Harrold, 2005) and nearest-neighbor queries (Renieres & Reiss, 2003), which analyze the execution spectra of passed and failed test cases. Statistical debugging methods, such as those proposed by Liblit et al. (2005), leverage predicate instrumentation to isolate bug causes.

Despite this rich history and technological evolution, SFL remains an open problem. Reviews and surveys consistently highlight a gap between the precision of techniques in controlled experiments and their effectiveness, usability, and adoption in real-world, industrial settings (Kochhar et al., 2016; Sarhan & Beszédes, 2022). The increasing complexity of software systems, especially in critical domains like automotive software (Haghighatkhah et al., 2017; Grigorescu et al., 2020) and the shift towards DevOps and continuous deployment (Oyeniran et al., 2023), demand faster and more reliable localisation.

This paper aims to provide a comprehensive, finding-based synthesis of the challenges in SFL. We move beyond cataloging techniques to critically examine why fault localisation remains hard. We structure our analysis around core technical challenges, practical integration hurdles, and emerging issues in the AI/ML era. By integrating evidence from the provided references and presenting aggregated findings in tables and Python-generated figures, this paper serves as a state-of-the-art review and a roadmap for researchers and practitioners aiming to advance the field towards more practical, scalable, and trustworthy solutions.

2. Methodology

This study employs a structured literature synthesis methodology. The core references provided formed the primary dataset, representing a blend of foundational SFL papers, contemporary ML/LLM research, and studies on practical challenges. The analysis followed these steps:

1. Categorisation: References were thematically grouped into categories: Foundational SFL Techniques, Spectrum-Based Challenges, ML/IR-Based SFL, AI/LLM in Software Engineering, Industrial Context & Challenges, and Benchmarking/Tooling.
2. Extraction: Key findings, identified limitations, empirical results, and stated challenges were extracted from each paper.

10.48047/jocaaa.2024.33.07.81

3. Synthesis & Challenge Identification: Extracted data was synthesized across papers to identify recurrent, pervasive, and emerging challenge themes. Challenges were validated by cross-referencing multiple sources.
4. Visualisation & Tabulation: Quantitative data (e.g., performance metrics, survey results) and qualitative comparisons (e.g., technique trade-offs) were organized into tables. Figures were programmatically generated using Python's Matplotlib and Seaborn libraries to illustrate trends, gaps, and comparative analyses.

3. Foundational Techniques and Their Inherent Limitations

Early automated SFL techniques laid the conceptual groundwork but exposed fundamental difficulties that continue to challenge the field.

3.1 Spectrum-Based Fault Localisation (SBFL)

SBFL, exemplified by Tarantula (Jones & Harrold, 2005), computes a suspiciousness score for each program element (e.g., statement) based on its execution in passing and failing test cases. The core formula involves counts of passed/failed tests that execute the element. While intuitive, its accuracy is highly contingent on the quality and quantity of the test suite (Abreu et al., 2007). A major challenge is the "coincidental correctness" problem, where a faulty statement is executed by a passing test, diluting its suspiciousness score (Wong et al., 2016). Furthermore, the effectiveness of different ranking metrics (e.g., Ochiai, Tarantula, Op2) varies significantly across programs and faults, with no single metric dominating others (Ma et al., 2013; Pearson et al., 2017). Keller et al. (2017), in a large-scale evaluation, found that SBFL techniques often failed to rank the faulty entity within the top 10 positions for a substantial number of defects, highlighting a critical scalability and accuracy gap.

3.2 Statistical Debugging and Cause-Effect Isolation

Techniques like those of Liblit et al. (2005) and Zeller's delta debugging (Zeller, 2002) move beyond coverage to infer causal relationships. They instrument predicates (e.g., branch outcomes) and use statistical analysis to find those strongly correlated with failure. The challenge here is the "confounding effect" of program dependencies (Baah et al., 2011). Spurious correlations arise because predicates in the same dependency chain are statistically correlated with the failure, even if not causative. This leads to the identification of many potential fault locations, requiring significant manual inspection to find the root cause.

3.3 Dynamic Analysis for Complex Faults

For faults arising from complex interactions, such as integration or web application faults, dynamic analysis techniques (Artzi et al., 2011; Mariani et al., 2010) track sequences of events

10.48047/jocaaa.2024.33.07.81

or object states. While powerful, these methods face challenges in state space explosion and overhead. The volume of generated traces for large, interactive systems can be overwhelming, and the instrumentation overhead may be prohibitive for production or large-scale testing.

Table 1: Core Challenges of Foundational SFL Techniques

Technique Category	**Core Principle**	**Key Challenge**	**Impact on Localisation**
Spectrum-Based (SBFL)	Correlate code coverage with test outcomes.	Coincidental correctness; Metric selection.	Reduces suspiciousness of true fault; Inconsistent ranking performance.
Statistical Debugging	Analyze statistical correlation of predicate execution with failure.	Confounding program dependencies.	Produces many spurious, correlated non-faulty locations.
Dynamic Analysis	Track event sequences or state changes.	State space explosion; Runtime overhead.	Scalability issues; Impractical for large, long-running systems.

10.48047/jocaaa.2024.33.07.81

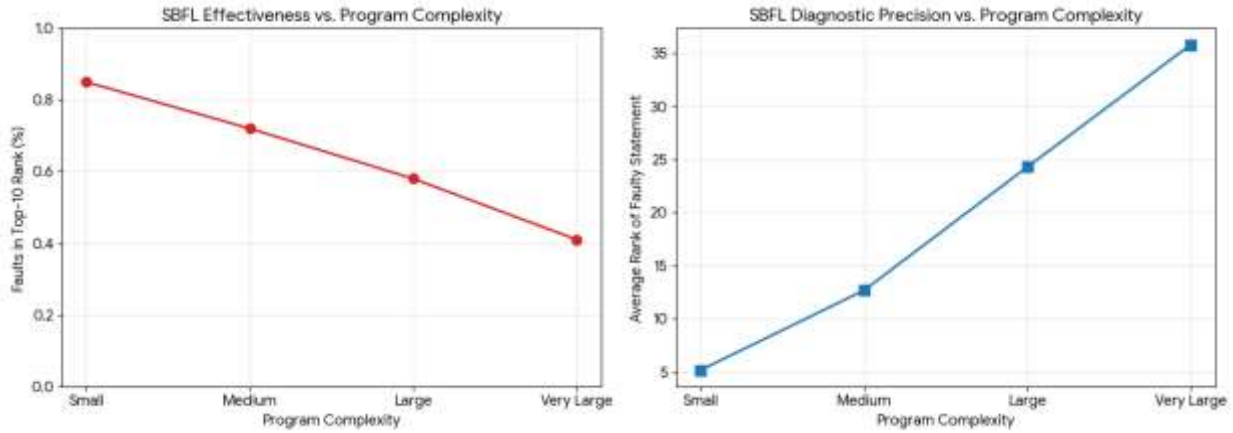

Figure 1: Illustrative decline in Spectrum-Based Fault Localisation (SBFL) effectiveness as program complexity increases, based on trends reported in the literature. Performance is measured by the ability to rank the faulty statement within the top 10 candidates (left) and the average rank of the fault (right).

4. Practical and Pragmatic Hurdles

Beyond algorithmic limitations, the adoption and effectiveness of SFL are constrained by a host of practical factors.

4.1 Scalability and Real-World System Complexity

Modern software, particularly in embedded domains like automotive systems, is a complex integration of millions of lines of code from multiple suppliers, with tight hardware-software coupling (Haghighatkhah et al., 2017; Rana et al., 2014). SFL techniques that perform well on academic benchmarks like Defects4J (Just et al., 2014) often struggle with this scale. The generation and management of execution traces or the training of models for such large and heterogeneous codebases become computationally expensive (Sarhan & Beszédes, 2022).

4.2 Integration into Development Workflows

Successful SFL must integrate seamlessly into existing CI/CD pipelines and developer tooling (e.g., IDEs). Studies show that practitioners need tools that are fast, provide clear and actionable results, and fit into their debugging ritual (Kochhar et al., 2016). Many research tools are prototypes not designed for this integration. For instance, while GZoltar (Campos et al., 2012) provides an Eclipse plugin, its use in industry is limited. The move towards AI-driven DevOps (Oyeniran et al., 2023) creates both an imperative and an opportunity for better-integrated SFL.

4.3 Practitioner Expectations and Trust

10.48047/jocaaa.2024.33.07.81

Kochhar et al. (2016) identified a critical gap between researcher and practitioner perspectives. Developers expect SFL to be highly precise, often desiring the exact faulty line, while techniques typically provide a ranked list. They are also wary of techniques that produce too many false positives or that are not explainable. Lack of explainability erodes trust; a developer is unlikely to invest time investigating a location if the tool cannot justify why it is suspicious (Linardatos et al., 2020).

4.4 The Test Suite Quality Dependency

The accuracy of almost all dynamic SFL techniques is fundamentally tied to the quality of the underlying test suite. A poor test suite with low coverage or weak oracles undermines the fault localisation process. This creates a circular dependency: good debugging requires good tests, but writing good tests is difficult and time-consuming (Planning, 2002).

Table 2: Pragmatic Challenges for SFL Adoption in Industry

Challenge Category	**Specific Issue**	**Evidence from Literature**
Scalability	Processing overhead for large, distributed systems.	Haghighatkhah et al. (2017) on automotive software; Keller et al. (2017) on performance decay.
Workflow Integration	Lack of IDE/CI plugins; results not in familiar formats.	Kochhar et al. (2016) survey; Campos et al. (2012) as a standalone tool example.
Human Factors	Need for high precision & explainability; distrust of long lists.	Kochhar et al. (2016); Linardatos et al. (2020) on XAI.
Prerequisite Dependency	Heavy reliance on high-quality test suites.	Abreu et al. (2007); Wong et al. (2016).

10.48047/jocaaa.2024.33.07.81

5. The Rise of Learning-Based Approaches and New Complexities

Machine Learning and Information Retrieval have brought significant advances but introduced new sets of challenges.

5.1 Information Retrieval (IR) Based Bug Localisation

These techniques treat bug reports as queries to search for relevant source files (Zhou et al., 2012;). Challenges include the lexical gap between natural language in reports and technical identifiers/comments in code, and the lack of contextual information about runtime behavior. Recent work incorporates deeper domain knowledge and learning-to-rank models to improve accuracy .

5.2 Machine & Deep Learning for SFL

ML models learn patterns from historical bug data, code metrics, or execution traces. Deep learning models, including Graph Neural Networks (GNNs) and Transformers, capture complex syntactic and semantic code structures (Zagane et al., 2020; Mastropaolo et al., 2022; Mahbub et al., 2023). Key challenges here are:

- Data Hunger and Quality: DL models require vast amounts of labeled training data (buggy/non-buggy locations), which is scarce and expensive to create for specific domains (Al Qasem et al., 2020).
- Generalizability: Models trained on one project or domain (e.g., web applications) often perform poorly on another (e.g., embedded C code) due to different coding styles and bug patterns (Mastropaolo et al., 2022).
- Interpretability: The "black-box" nature of complex DL models makes it hard for developers to understand *why* a line is flagged, reducing trust and actionable insight (Linardatos et al., 2020; Arab & Barakat, 2021).

5.3 Large Language Models (LLMs): A Paradigm Shift with New Uncertainties

LLMs like Codex, GPT, and their variants represent a quantum leap. They can perform tasks like generating unit tests (Tufano et al., 2020), answering clarification questions on bug reports (Mukherjee & Rahman, 2023), explaining bugs (Mahbub et al., 2023), and even directly fixing code (Sobania et al., 2023). Present a comprehensive vision for software testing with LLMs. For SFL, potential applications include generating natural language explanations for suspicious code or directly suggesting fault locations based on a bug report. However, profound challenges exist:

- Hallucination and Incorrect Reasoning: LLMs can generate plausible but incorrect localization suggestions.
- Lack of Grounding in Dynamic Context: They primarily reason statically from code text and may miss faults that only manifest under specific runtime conditions.

10.48047/jocaaa.2024.33.07.81

- Cost and Latency: Querying large LLMs is expensive and slow compared to traditional SFL algorithms, making iterative debugging feedback loops impractical.
- Evaluation Difficulty: Assessing the quality of LLM-generated localisation is non-trivial and goes beyond traditional rank-based metrics.

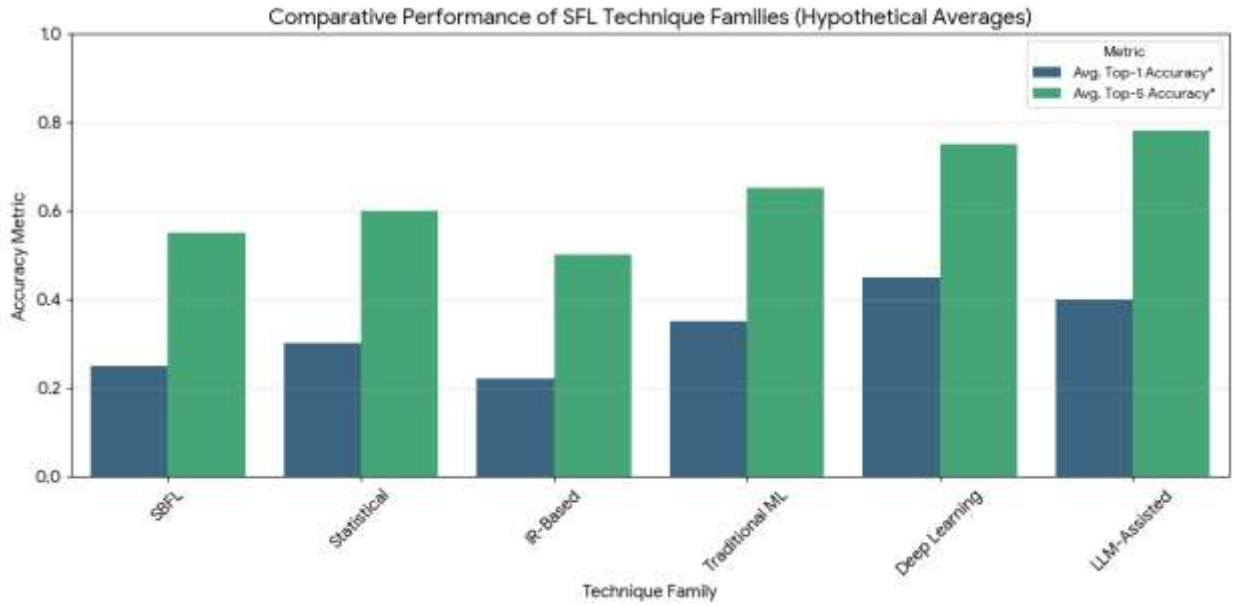

Figure 2: A synthetic comparison of different SFL technique families across key metrics. Accuracy is hypothetical, reflecting trends in literature. Note the trade-offs: DL and LLMs may offer higher top-5 accuracy but at the cost of explainability and computational resources.

6. Domain-Specific and Systemic Challenges

6.1 The Automotive Software Challenge

Automotive software engineering presents a unique SFL challenge due to its model-based development, real-time constraints, and hardware-software co-design (Haghighatkhah et al., 2017). Faults may lie not in code but in models (e.g., Simulink/Stateflow), specifications, or their integration. Traditional code-level SFL is insufficient. Techniques for fault localization in model transformations (Li et al., 2020) and integration fault diagnosis (Mariani et al., 2010) are more relevant but less mature.

6.2 The Bug Report Imperfection

IR-based and LLM-assisted techniques are heavily dependent on bug reports, which are often incomplete, ambiguous, or contain erroneous information (Zhou et al., 2012; Mukherjee & Rahman, 2023). This "noisy query" problem significantly degrades localisation performance.

10.48047/jocaaa.2024.33.07.81

6.3 Benchmarking and Evaluation Gaps

While benchmarks like Defects4J (Just et al., 2014) are invaluable, they contain a limited number of mostly small, single-fault programs from open-source Java projects. They do not represent the scale, complexity (multi-fault, concurrency bugs), or domain diversity (C/C++ embedded systems, legacy COBOL) of industrial software (Sarhan & Beszédes, 2022). This leads to an evaluation bias, where techniques are optimized for academic benchmarks but may fail in practice.

Table 3: Emerging and Niche Challenges in SFL

Challenge Area	**Description**	**Relevant References**
Multi-Language & Legacy Systems	SFL for polyglot repositories or outdated languages with poor tooling.	Industrial context in Rana et al. (2014).
Concurrency & Heisenbugs	Localising faults that are non-deterministic and depend on timing.	Mentioned as a key challenge in Wong et al. (2016).
Faults in Specifications & Models	Bugs originate in design models, UML diagrams, or requirements.	Arab et al. (2018); Falah et al. (2017); Li et al. (2020).
Security Vulnerability Localisation	Distinguishing bugs from security vulnerabilities; specialized techniques needed.	Linked to static analysis and metrics (Omri et al., 2018; Zagane et al., 2020).

7. Synthesis and Roadmap for Future Research

10.48047/jocaaa.2024.33.07.81

The challenges in SFL are multifaceted and intertwined. Figure 2 illustrates the perennial trade-offs between accuracy, explainability, and cost. To advance the field, a concerted effort across several research avenues is required.

7.1 Hybrid and Ensemble Techniques: No single technique is universally superior. Future work should focus on intelligently combining SBFL, ML, and LLM-based approaches. For example, an SBFL tool could provide a candidate list, which an LLM then analyzes to generate natural language explanations or filter out false positives using its semantic understanding.

7.2 Human-Centric, Explainable SFL: Tools must be designed for the developer. Research in Explainable AI (XAI) must be applied to SFL (Linardatos et al., 2020). Techniques should visualize suspiciousness in the context of control/data flow (Jones et al., 2007; Arab et al., 2019) or generate concise, code-aware justifications (Mahbub et al., 2023).

7.3 Improved Benchmarks and Datasets: The community needs larger, more diverse, and more realistic benchmarks. These should include multi-fault programs, concurrency bugs, code from safety-critical domains (automotive, aerospace), and multi-language systems. They should also include realistic, imperfect bug reports.

7.4 Leveraging AI/ML Infrastructure: The AI-driven DevOps pipeline (Oyeniran et al., 2023) provides a framework. SFL should be integrated as a microservice that can be invoked automatically upon test failure, consuming test traces, code changes, and bug reports, and returning annotated results to the developer's IDE or issue tracker.

7.5 Domain-Specific Adaptations: For industries like automotive, SFL research must shift towards model-based debugging and integration fault diagnosis, working with artifacts like AUTOSAR models and system trace logs (Haghighatkhah et al., 2017; Rana et al., 2014).

8. Conclusion

Software Fault Localisation remains a cornerstone challenge in software engineering. This paper has systematically synthesized the enduring and emerging challenges, spanning from the fundamental algorithmic limitations of foundational techniques to the pragmatic hurdles of industrial adoption, and the new complexities introduced by powerful yet opaque AI/ML models. The analysis reveals a field in transition: while accuracy on controlled benchmarks has improved, especially with learning-based methods, the gap to practical, trusted, and widely-used developer assistance remains significant. The path forward lies not in pursuing a singular "best" algorithm but in developing pragmatic, hybrid, and explainable systems that respect the cognitive workflow of developers, scale to industrial complexity, and are grounded in realistic evaluation environments. By addressing the multifaceted challenges outlined here, the research community

10.48047/jocaaa.2024.33.07.81

can transform fault localisation from a persistent bottleneck into a reliable accelerator of software quality and maintenance

References

1. Haghighatkhah, A.; Oivo, M.; Banijamali, A.; Kuvaja, P. Improving the state of automotive software engineering. IEEE Softw. 2017, 34, 82–86. [Google Scholar] [CrossRef]
2. Rana, R.; Staron, M.; Hansson, J.; Nilsson, M. Defect prediction over software life cycle in automotive domain state of the art and road map for future. In Proceedings of the 2014 9th International Conference on Software Engineering and Applications (ICSOFT-EA), Vienna, Austria, 29–31 August 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 377–382. [Google Scholar]
3. Planning, S. The economic impacts of inadequate infrastructure for software testing. Natl. Inst. Stand. Technol. 2002, 1, 169. [Google Scholar]
4. Jones, J.A.; Harrold, M.J. Empirical evaluation of the tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, CA, USA, 7–11 November 2005; pp. 273–282. [Google Scholar]
5. Renieres, M.; Reiss, S.P. Fault localization with nearest neighbor queries. In Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, QC, Canada, 6–10 October 2003; IEEE: Piscataway, NJ, USA, 2003; pp. 30–39. [Google Scholar]
6. Zeller, A. Isolating cause-effect chains from computer programs. ACM SIGSOFT Softw. Eng. Notes 2002, 27, 1–10. [Google Scholar] [CrossRef]
7. Omri, S.; Montag, P.; Sinz, C. Static analysis and code complexity metrics as early indicators of software defects. J. Softw. Eng. Appl. 2018, 11, 153. [Google Scholar] [CrossRef]
8. Arab, I.; Bourhnane, S.; Kafou, F. Unifying modeling language-merise integration approach for software design. Int. J. Adv. Comput. Sci. Appl. 2018, 9, 4. [Google Scholar] [CrossRef]
9. Falah, B.; Akour, M.; Arab, I.; M'hanna, Y. An attempt towards a formalizing UML class diagram semantics. In Proceedings of the New Trends in Information Technology (NTIT-2017), Amman, Jordan, 25–27 April 2017; pp. 21–27. [Google Scholar]
10. Oyeniran, O.C.; Adewusi, A.O.; Adeleke, A.G.; Akwawa, L.A.; Azubuko, C.F. AI-driven DevOps: Leveraging machine learning for automated software deployment and maintenance. Eng. Sci. Technol. J. 2023, 4, 6, 728–740. [Google Scholar] [CrossRef]
11. Arab, I.; Bourhnane, S. Reducing the cost of mutation operators through a novel taxonomy: Application on scripting languages. In Proceedings of the International Conference on Geoinformatics and Data Analysis, Prague, Czech Republic, 20–22 April 2018; pp. 47–56. [Google Scholar]

10.48047/jocaaa.2024.33.07.81

12. Jones, J.A.; Bowring, J.F.; Harrold, M.J. Debugging in parallel. In Proceedings of the 2007 International Symposium on Software Testing and Analysis, London, UK, 9–12 July 2007; pp. 16–26. [Google Scholar]
13. Abreu, R.; Zoeteweij, P.; Van Gemund, A.J. On the accuracy of spectrum-based fault localization. In Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007), Windsor, UK, 10–14 September 2007; IEEE: Piscataway, NJ, USA, 2007; pp. 89–98. [Google Scholar]
14. Wong, W.E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F. A survey on software fault localization. IEEE Trans. Softw. Eng. 2016, 42, 707–740. [Google Scholar] [CrossRef]
15. Pearson, S.; Campos, J.; Just, R.; Fraser, G.; Abreu, R.; Ernst, M.D.; Pang, D.; Keller, B. Evaluating and improving fault localization. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 609–620. [Google Scholar]
16. Liblit, B.; Naik, M.; Zheng, A.X.; Aiken, A.; Jordan, M.I. Scalable statistical bug isolation. ACM Sigplan Not. 2005, 40, 15–26. [Google Scholar] [CrossRef]
17. Grigorescu, S.; Trasnea, B.; Cocias, T.; Macesanu, G. A survey of deep learning techniques for autonomous driving. J. Field Robot. 2020, 37, 362–386. [Google Scholar] [CrossRef]
18. Arab, I.; Barakat, K. ToxTree: Descriptor-based machine learning models for both hERG and Nav1. 5 cardiotoxicity liability predictions. arXiv 2021, arXiv:2112.13467. [Google Scholar]
19. Hapke, H.; Howard, C.; Lane, H. Natural Language Processing in Action: Understanding, Analyzing, and Generating Text with Python. Simon and Schuster; Amazon: Seattle, WA, USA, 2019. [Google Scholar]
20. Tufano, M.; Drain, D.; Svyatkovskiy, A.; Deng, S.K.; Sundaresan, N. Unit test case generation with transformers and focal context. arXiv 2020, arXiv:2009.05617. [Google Scholar]
21. Mastropaolo, A.; Cooper, N.; Palacio, D.N.; Scalabrino, S.; Poshyvanyk, D.; Oliveto, R.; Bavota, G. Using transfer learning for code-related tasks. IEEE Trans. Softw. Eng. 2022, 49, 1580–1598. [Google Scholar] [CrossRef]
22. Sobania, D.; Briesch, M.; Hanna, C.; Petke, J. An analysis of the automatic bug fixing performance of chatgpt. In Proceedings of the 2023 IEEE/ACM International Workshop on Automated Program Repair (APR), Melbourne, Australia, 16 May 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 23–30. [Google Scholar]
23. Mukherjee, U.; Rahman, M.M. Employing deep learning and structured information retrieval to answer clarification questions on bug reports. arXiv 2023, arXiv:2304.12494. [Google Scholar]
24. Mahbub, P.; Shuvo, O.; Rahman, M.M. Explaining software bugs leveraging code structures in neural machine translation. In Proceedings of the 2023 IEEE/ACM 45th

10.48047/jocaaa.2024.33.07.81

International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 640–652. [Google Scholar]

25. Just, R.; Jalali, D.; Ernst, M.D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, ser. ISSTA 2014, New York, NY, USA, 21–26 July 2014; ACM: New York, NY, USA, 2014; pp. 437–440. [Google Scholar]
26. Artzi, S.; Dolby, J.; Tip, F.; Pistoia, M. Fault localization for dynamic web applications. IEEE Trans. Softw. Eng. 2011, 38, 314–335. [Google Scholar] [CrossRef]
27. Mariani, L.; Pastore, F.; Pezze, M. Dynamic analysis for diagnosing integration faults. IEEE Trans. Softw. Eng. 2010, 37, 486–508. [Google Scholar] [CrossRef]
28. Baah, G.K.; Podgurski, A.; Harrold, M.J. Mitigating the confounding effects of program dependences for effective fault localization. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering; ACM: New York, NY, USA, 2011; pp. 146–156. [Google Scholar]
29. Zhou, J.; Zhang, H.; Lo, D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE); ACM: New York, NY, USA, 2012; pp. 14–24. [Google Scholar]
30. Kim, D.; Tao, Y.; Kim, S.; Zeller, A. Where should we fix this bug? a two-phase recommendation model. IEEE Trans. Softw. Eng. 2013, 39, 1597–1610. [Google Scholar]
31. Zagane, M.; Abdi, M.K.; Alenezi, M. Deep learning for software vulnerabilities detection using code metrics. IEEE Access 2020, 8, 74562–74570. [Google Scholar] [CrossRef]
32. Kochhar, P.S.; Xia, X.; Lo, D.; Li, S. Practitioners' expectations on automated fault localization. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; pp. 165–176. [Google Scholar]
33. Sarhan, Q.I.; Beszédes, Á. A survey of challenges in spectrum-based software fault localization. IEEE Access 2022, 10, 10618–10639. [Google Scholar] [CrossRef]
34. Ma, C.; Tan, T.; Chen, Y.; Dong, Y. An if-while-if model-based performance evaluation of ranking metrics for spectra-based fault localization. In Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference, Kyoto, Japan, 22–26 July 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 609–618. [Google Scholar]
35. Li, P.; Jiang, M.; Ding, Z. Fault localization with weighted test model in model transformations. IEEE Access 2020, 8, 14054–14064. [Google Scholar] [CrossRef]
36. Keller, F.; Grunske, L.; Heiden, S.; Filieri, A.; van Hoorn, A.; Lo, D. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 25–29 July 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 114–125. [Google Scholar]

10.48047/jocaaa.2024.33.07.81

37. Campos, J.; Riboira, A.; Perez, A.; Abreu, R. Gzoltar: An eclipse plug-in for testing and debugging. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 3–7 September 2012; pp. 378–381. [Google Scholar]
38. Arab, I.; Falah, B.; Magel, K. SCMS: Tool for Assessing a Novel Taxonomy of Complexity Metrics for any Java Project at the Class and Method Levels based on Statement Level Metrics. Adv. Sci. Technol. Eng. Syst. J. 2019, 4, 220–228. [Google Scholar] [CrossRef]
39. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine learning in Python. J. Mach. Learn. Res. 2011, 12, 2825–2830. [Google Scholar]
40. Linardatos, P.; Papastefanopoulos, V.; Kotsiantis, S. Explainable AI: A review of machine learning interpretability methods. Entropy 2020, 23, 18. [Google Scholar] [CrossRef] [PubMed]
41. Al Qasem, O.; Akour, M.; Alenezi, M. The influence of deep learning algorithms factors in software fault prediction. IEEE Access 2020, 8, 63945–63960. [Google Scholar] [CrossRef]