Improving File-Level Bug Localization Using Pre-Trained Code Models: A Comprehensive Review

Al-Anzi Tuqa Emad Hussein
Department of Software, College of Computer Science and Mathematics, University of Mosul, Iraq
tuqa.23csp17@student.uomosul.edu.iq (corresponding author)

Aldabbagh Mohammad A. Taha
Department of Software, College of Computer Science and Mathematics, University of Mosul, Iraq
m.a.taha@uomosul.edu.iq

Received: 6 March 2026 | Revised: 25 April 2026 | Accepted: 30 April 2026

Licensed under a CC-BY 4.0 license | Copyright (c) by the authors | *DOI: https://doi.org/10.48084/etasr.18559*

ABSTRACT

Bug localization, the automatic detection of source code files that contain a given defect, is a fundamental problem in software maintenance. Pre-trained models, such as CodeBERT, GraphCodeBERT, UniXcoder, and CodeT5, can effectively bridge the semantic gap between bug reports in natural language and source code. However, existing studies use inconsistent datasets and evaluation protocols, leading to non-comparable and non-reproducible results. This review focuses on file-level bug localization using pre-trained models, going beyond prior surveys by identifying cross-study inconsistencies, highlighting a structural gap in LLM-based file-level evaluation, and providing a critical analysis of existing approaches. The key contributions are: a five-category taxonomy combining IR, ML, deep learning, pre-trained language model, and LLM-based approaches (the first taxonomy targeting file-level granularity among all paradigms); a cross-study analysis suggests that identical models on identical benchmarks report Top-10 accuracy values differing by up to 20 percentage points (as a result of undisclosed experimental differences); and the mapping into a structured framework of eight open research gaps to seven evidence-supported directions, followed by the most recent advances in the field, including studies published up to 2026.

Keywords-bug localization; file-level bug localization; pre-trained code models; CodeBERT; GraphCodeBERT; UniXcoder; CodeT5; software maintenance; deep learning

I. INTRODUCTION

Bug localization is one of the basic tasks in software maintenance to automatically detect source code artifacts accountable for reported defects. As large-scale software systems grow in complexity, debugging becomes one of the most expensive and tedious activities throughout the development lifecycle [1]. Empirical research shows that developers spend an unfair proportion of their maintenance time locating faulty code, demonstrating the need for effective automated localization methods. Initially, with bug localization, the problem was primarily framed in terms of Information Retrieval (IR) [2]. Traditional IR-based methods, including Vector Space Models, TF-IDF weighting, and probabilistic ranking functions, provided a practical guideline but were inherently challenged by the lack of appropriate vocabulary alignment, noisy bug descriptions, and the lack of lexical features that would hold semantic relationships of natural language to source code [3-6]. Later, ML methods improved localization and learned from encoded textual, structural, and historical tokens [7]. Deep Learning (DL) models—CNNs, RNNs, and, in particular, hybrid architectures—have also led progress by automatically learning representations, minimizing the need for feature engineering [8, 9]. However, DL approaches usually depend on large labeled datasets and fail in generalization across projects and programming languages [10].

In recent years, pre-trained language models for code have been particularly appealing to early software engineering research. Models such as CodeBERT, GraphCodeBERT, UniXcoder, and CodeT5 are pre-trained on large-scale corpora, including source code and natural language features, to be able to learn intricate contextual representations capturing syntactic structure and semantic meaning [11-14]. These models have exhibited good performance in a wide array of downstream tasks such as code search, defect prediction, program repair, and bug localization [15-17]. By embedding bug reports and the source code in a shared semantic space, pre-trained code models help reduce the semantic gap that has existed long before in traditional bug localization approaches [18, 19]. More

recently, some Large Language Model (LLM) architectures have built on this work with interpretability and zero-shot generalization [20]. However, such advances are still fragmented, as generalization, dataset bias, reproducibility, and computational efficiency issues (and many different types of bias) between studies have often been underreported.

This review focuses on studies published between 2012 and 2026, from IEEE Xplore, ACM Digital Library, ScienceDirect, SpringerLink, and arXiv, selected for their relevance to file-level bug localization, experimental quality, and pre-trained or LLMs. The contributions of this review can be summarized as follows:

- A five-category taxonomy of file-level bug localization methods using IR, ML, DL, pre-trained language models, and LLM-based approaches, the first taxonomy specifically aiming at file-level granularity across the five paradigms.
- A crucial cross-study analysis reveals that the same model (CodeBERT) assessed on the same benchmark (Bench4BL) could yield Top-10 accuracy values differing by up to 20 percentage points between studies due to undisclosed experimental conditions—a result missing from earlier surveys.
- Identification of a structural field gap: LLM-based bug localization studies very nearly only evaluate at the method level, leaving file-level localization systematically underserved by the most capable known models.
- The latest and most up-to-date comprehensive coverage of the field by 2026 with LLM-based frameworks (LLMLoc, BugLLM, BLAZE, Knowledge-enhanced LLMs), not examined in earlier surveys, along with an evaluation of their experimental limitations.

This review contrasts with the most analogous survey [18], which focused on DL for bug localization in general. In contrast, this work focuses specifically on pre-trained models at file-level granularity, adopts a critical analytical orientation, and covers a more recent literature window.

II. BACKGROUND ON BUG LOCALIZATION

Bug localization is the process of finding source code artifacts that lead to a detected defect. Bug reports and source entities are given in order according to the probability of containing the fault. Localization can be performed at different granularities (file, class, method, statement), the file-level being the most practical [1]. Typical evaluation measures include Top-k Accuracy, MRR, and MAP [2, 4]. Earlier solutions, such as BugLocator, BLUiR, and BRTracer, are based on textual similarity, structural information, and stack trace signals [1-3], but are limited by vocabulary mismatch and shallow lexical features [5, 6]. Semantic embeddings are used to alleviate these limitations in learning-based methods, feature fusion is employed [7-9, 21], while pre-trained code models further improve performance by embedding bug reports and code in a shared semantic space [11-17]. However, the evaluation results are still strongly correlated with the dataset's characteristics and settings [22, 23].

III. BUG LOCALIZATION TAXONOMY

Bug localization methods can be categorized into five paradigms, according to the way they represent and learn. These categories are shown in Figure 1: IR-based, ML-based, DL-based, pre-trained language model-based, and LLM-based methods.

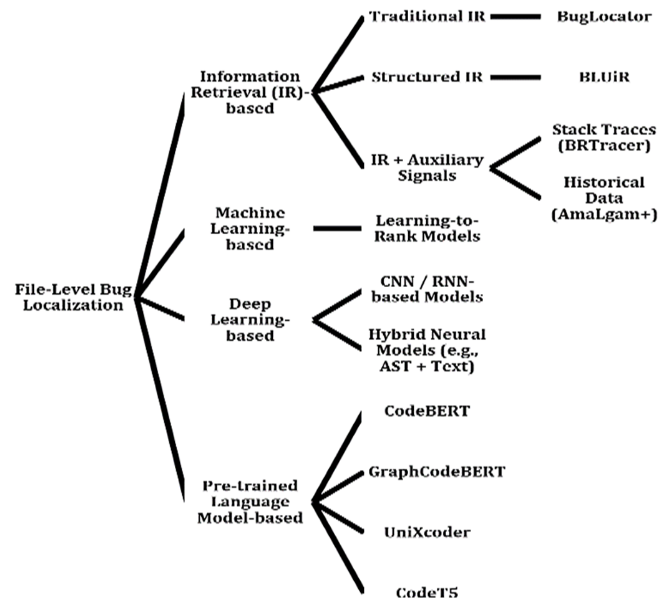

Fig. 1. File-level bug localization taxonomy.

A. *Information Retrieval (IR)-Based Methods*

IR-based approaches represent the first branch of bug localization. They frame the problem as text retrieval: a bug report is the query, and source code files are documents ranked by lexical similarity. BugLocator uses revised vector space models and file-size normalization to rank candidates [1]. BLUiR provides specific information to the class-state (by class names, method names, comments, etc. [2]). BRTracer uses stack trace information to favor files that have been named and specified during fault-inducing execution [3]. AmaLgam+ and two-level IR models combine diverse signals such as version history and coarse-to-fine search approaches [4, 24].

IR-based techniques are quick, easy, and do not require training data, which makes them appealing baselines. Their fundamental limitation is their reliance on surface-level lexical overlap. The vocabulary mismatch between natural language bug descriptions and source code identifiers, which often have different terminology for the same concept, greatly decreases their suitability for complex or cross-project applications [5, 6]. Such limitations drive the shift to learning-based paradigms.

B. *Machine Learning (ML)-Based Methods*

ML models solve the problem of only matching lexical values with training models according to hand-constructed features built off bug reports and source code. Feature sets usually consist of textual statistics, structural properties of code files, historical change metrics, and fault-proneness indicators. In [7, 25], it was shown that multi-feature learning-to-rank

frameworks lead to performance enhancements over classical IR. In [26], a learning-to-rank framework with multi-feature fusion was proposed for hardware bug localization, reporting a notable improvement relative to spectrum-based baselines. Such techniques utilizing version control history and changelogs are inspired by the empirical observation that frequently modified files are disproportionately fault-prone [6].

ML methods are more expressive in nature than IR techniques, but introduce a new limitation, the quality of feature engineering, which is labor-intensive and so narrowly based. In general, performance does not migrate easily between projects or programming languages, and the underlying assumption that historic behavior predicts future bugs does not hold readily for emergent systems [7].

C. Deep Learning (DL)-Based Methods

DL approaches promote bug localization by analyzing feature representations and learning from raw code and bug reports, which eliminates manual feature engineering. Word embedding models, CNNs, and RNNs have been extensively used to account for the context and semantics of co-occurring textual descriptions and code artifacts [8]. Some hybrid methods, such as DRAST, jointly model structural (AST) and semantic (textual) patterns [9]. DeepFL combines coverage-based spectra, mutation features, and textual similarity together with a deep neural network, showing significantly improved robustness to diverse fault patterns [21]. Recent work explores contrastive learning to explicitly push correct and incorrect bug report-code pairs apart in the embedding space, producing more discriminative representations [27].

Deep learning significantly outperforms IR and ML methods in semantic modeling. However, consistent results across studies demonstrated that such models require large quantities of labeled data and do not generalize across projects, especially when training and testing projects diverge in size, language, or domain [22]. Their data hunger and limited transferability are the primary drivers behind the adoption of pre-trained models.

D. Pre-Trained Language Model-Based Approaches

Pre-trained language models for code represent a paradigm shift in bug localization. Models such as CodeBERT, GraphCodeBERT, UniXcoder, and CodeT5 are pre-trained on large corpora of source code and natural language, enabling them to capture rich syntactic and semantic representations applicable to diverse downstream tasks [11-14]. These models are progressively used as backbone encoders in bug localization frameworks. In [16], it was shown that BERT-based changeset localization substantially reduces the semantic gap and improves file-level accuracy. Graph-enhanced and identifier-aware models [12, 14] further strengthen semantic understanding by incorporating data-flow information and developer-defined identifiers. For instance, reports by CodeBERT on Bench4BL [15] benchmarks for Top-10 accuracy values range widely, from 58% to 79%—a substantial range for the same model using the same benchmark suite. These inconsistencies cannot be pinned on the model alone, but rather on differences between preprocessing (e.g., whether stack trace information is included), dataset splits (chronological vs. random), and evaluation assumptions (e.g., whether duplicate bug reports are excluded). No prior studies have reported this finding, which has emerged as an important reproducibility issue for the field.

Strong results have been observed in [16] in changeset-based localization, but the evaluation assumes that file-level changesets are available at localization time—an assumption that may not hold in realistic development timelines where bug reports precede code changes. This restricts the practical implications of results.

TABLE I. COMPARISON OF PRE-TRAINED CODE MODELS FOR BUG LOCALIZATION

Model	Key strength	Typical advantage	Main limitation
CodeBERT [11]	Joint NL-Code pre-training on large corpora	Strong semantic baseline; widely adopted	Limited structural awareness; no data-flow
GraphCodeBERT [12]	Data-flow graphs incorporated into pre-training	Better understanding of program logic	Higher computational cost and complexity
UniXcoder [13]	Unified cross-modal encoder-decoder pre-training	Strong cross-project and cross-language transfer	Large model size; higher inference cost
CodeT5 [14]	Identifier-aware encoder-decoder architecture	Flexible for multiple software engineering tasks	Less specialized for pure ranking tasks

E. Large Language Model (LLM)-Based and Advanced Hybrid Approaches

In recent works, LLMs and hybrid approaches are used for bug localization. This approach combines LLM embeddings in IR pipelines, knowledge augmentation, dynamic chunking, and reasoning approaches to enhance localization sensitivity and interpretability [28,29]. Cross-project and cross-language frameworks, including UniXcoder-based localization [17] and BLAZE [30], show that much is possible to generalize this phenomenon to both software systems and programming languages. Zero-shot and structure-aware systems such as LLMLoc [31], and explainable frameworks such as BugLLM [32], are more recent strategies for developing adaptability and developer trust.

The systematic screening of LLM-based literature shows that most of these methods are assessed at the method level, class level, or without explicit granularity specification—not at the file level. This reflects an open structural hole: the latest technologies are not being systematically tested on the problem for which file-level localization is most practically relevant. This observation was not reported in previous reviews.

Table II presents a taxonomy summary, comparing different bug localization approaches.

TABLE II. COMPARISON OF BUG LOCALIZATION APPROACHES

Approach	Strengths	Limitations	Key studies
Traditional IR	Simple; fast; no training required	Lexical matching only; severe semantic gap	[1-4]
Machine Learning (ML)	Feature fusion improves ranking	Feature engineering bottleneck; limited generalization	[7, 26]
Deep Learning (DL)	Learns semantic representations automatically	Data-hungry; poor cross-project transfer	[8, 9, 21, 27]
Pre-trained code models	Rich semantics; best generalization	Reproducibility issues; high computational cost	[11-17, 23]
LLM-based/ Hybrid	Interpretability; zero-shot capability	File-level evaluation nearly absent; inference cost	[25, 26, 29-31]

IV. DISCUSSION AND RESEARCH GAPS

Despite extensive advances in file-level bug localization, there are still many long-standing problems and systemic shortcomings that need to be worked on. Although pre-trained methods substantially reduce the semantic gap compared to IR methods, performance lags are still observed if the bug reports are brief, vague, or devoid of structural cues like stack traces. This performance degradation is reported inconsistently across studies—some do not include stack-trace-less reports in the evaluation, while others do. This methodological inconsistency renders cross-study comparisons unreliable and increases the efficacy of approaches evaluated on cleaner datasets [5, 8, 28].

As mentioned above, an identical model (CodeBERT) concerning a common set of benchmark projects (Bench4BL) is found to produce a Top-10 accuracy level that differs by ~20 percentage points. Analysis of the experimental parts of these studies shows that the main discrepancies are: (i) whether the evaluation is chronological (observing time-ordering of bug reports and code states) or random-split, (ii) whether stack trace information is used as an additional feature, and (iii) how duplicate or close-to-duplicate bug reports are handled. None of these influences is well recognized in the referenced studies. Reproducibility-first presentation, with published preprocessing scripts and data splits, is essential [15, 22].

One of the major findings of this review is that LLM-based bug localization systems tend to assess the method-level and coarser granularity. None of the LLM-based trials described in the literature [28-32] have reported primary studies at the file level, specifically of a certain kind, using a controlled experimental design. This leaves a valuable and relevant gap unaddressed. In addition, the computational cost is rarely mentioned. Since LLM-based methods require substantially longer inference time compared to IR or classical ML methods, cost must be reported, and consideration of retrieval-first or changeset-based approaches to minimize costs is important to measure practical implementation effectiveness [16]. Finally, most existing systems generate ranked file lists without any understanding of the logic. Although LLM-based approaches such as BugLLM [32] seek to obtain an interpretable output, there is still no systematic assessment of the quality of the explanations from the developer's perspective.

TABLE III. SUMMARY OF REPRESENTATIVE STUDIES ON FILE-LEVEL BUG LOCALIZATION USING PRE-TRAINED MODELS

Ref. Year	Model	Dataset	Metrics	Key result	Critical observation
[15] 2018	IR + BERT baseline	Bench4BL (AspectJ, JDT, SWT, PDE, ZXing)	MAP, MRR, Top-1,5,10	Strong reproducibility analysis; IR baselines re-established	Exposes wide variance in prior IR results; demonstrates evaluation protocol is a major confound
[9] 2020	DRAST (DL + AST)	Bench4BL subsets	Top-1, Top-5, Top-10, MAP	AST integration improves structural understanding	Precedes pre-trained models; useful baseline; labeled data requirements limit applicability
[23] 2021	CodeBERT	Multiple SE task benchmarks	Task-specific	Generalizability varies significantly across tasks	Demonstrates that PLM transfer is task-sensitive; performance depends on similarity between pre-training and downstream tasks
[16] 2022	BERT (changeset-based)	Dataset of bug reports and bug-inducing changesets across multiple projects (AspectJ, JDT, PDE, SWT, Tomcat, ZXing)	MRR, MAP, Top-1, Top-5, Top-10	High Top-10 accuracy; fast changeset retrieval	Evaluation assumes changeset availability at query time — does not reflect realistic bug-report-first development workflow
[27] 2023	Contrastive learning + embeddings	Public bug localization dataset (Xiao et al.) with large-scale pre-training corpus (Fang et al.)	MRR,MAP ,Top-1, Top-5, Top-10	Improved discriminative representations	While datasets are publicly available, limited implementation and preprocessing details may affect reproducibility
[17] 2024	UniXcoder	Cross-language projects (Java, C/C++, Golang)	MAP, MRR, Accuracy@K (Acc@1, Acc@3, Acc@5, Acc@10)	Strong cross-language transfer demonstrated	File-level and method-level results not disaggregated
[28] 2025	LLM + IR pipeline	AspectJ, Eclipse JDT, SWT, Tomcat, ZXing, Birt + 16 GitHub projects (GHRB)	MAP, MRR, Accuracy@k	LLM improves IR retrieval quality	Sensitive to low-quality bug reports; performance degrades with reproduction-heavy or semantically weak descriptions
[29] 2025	LLM + knowledge graph	OSS projects (7 datasets, ~6000 bug reports)	MAP, MRR, Top-k	State-of-the-art on tested projects	Evaluated at commit, file, and hunk levels — demonstrates effectiveness of knowledge-enhanced LLMs for file-level bug localization
[30] 2025	BLAZE	Cross-project, multi-language benchmarks	MAP, Top-1, Top-5, Top-10, MRR	Robust cross-project and cross-language results	Requires complex multi-stage processing (chunking, retrieval, fine-tuning), which increases computational cost
[31] 2026	LLMLoc (structure-aware LLM)	Defects4J	MRR, MAP, Top-k (zero-shot)	Promising zero-shot performance	Early-stage work; limited benchmark coverage; predominantly function-level evaluation

V. OPEN RESEARCH DIRECTIONS

From the gaps highlighted above, seven research directions emerge as promising.

1. Strong semantic alignment for noisy/short bug reports: To overcome these challenges, better representation learning is needed in bug reports without stack traces or textual detail. Various existing studies find performance discrepancies between stack-trace-available and stack-trace-absent scenarios, although without systematic analysis or controlled ablation [5, 8, 28].
2. Standardized cross-project and cross-language benchmarks: Consistent datasets with chronological splits and unified preprocessing are essential. The variance in performance reported indicates that standardization is essential [17, 30].
3. LLM-based file-level bug localization: A key gap is the absence of systematic LLM evaluation at file-level granularity. This limitation can be directly addressed by applying existing frameworks in controlled file-level settings [29, 31].
4. Reproducibility-first experimentation: Transparent reporting of preprocessing, dataset construction, splits, and hyperparameters is required. Currently, such discrepancies are attributed to the lack of these practices [15, 22].
5. Hybrid systems with explicit ablation: Multi-component systems should incorporate ablation studies, measuring the performance impact of each component. Otherwise, performance benefits cannot be credibly attributed where they should [4, 29].
6. Efficiency-aware localization pipelines: Computational cost must be reported alongside accuracy. However, to our knowledge, there is no systematic comparison of retrieval-based, changeset-based, and LLM-based methods with respect to performance-efficiency trade-offs [16].
7. Explainable bug localization with developer-centered evaluation: Evaluation frameworks are required to assess which explanations enhance debugging efficiency, trust, and usability. Current work, including BugLLM [32], lacks such validation.

VI. CONCLUSION

This study presents a systematic exploration of file-level bug localization with a five-category taxonomy that serves as a unified framework. IR methods are useful baselines, but lexical matching limits them. Although learning-based and deep models enhance semantic representation, they require large datasets and show limited generalization. While pre-trained models are the current performance frontier, they are sensitive to evaluation protocols and dataset characteristics. LLM-based approaches show promise, but they have not yet been explored at file-level granularity.

Three primary findings emerge: (i) high cross-study variation in reported performance due to differences in experimental settings; (ii) the structural lack of LLM-based evaluation at file-level granularity; and (iii) the limitations in evaluation design in some landmark studies, which may affect the interpretation of results. Challenges, including standardization, reproducibility, and file-level evaluation of LLMs, must be addressed to provide practical and reliable bug localization solutions. The purpose of this review is to help future work produce precise, reproducible, generalizable, efficient, and interpretable techniques.

DECLARATION ON COMPETING INTERESTS

The authors declare no conflict of interest that could have influenced this study.

DATA AVAILABILITY

Not applicable in this study.

ACKNOWLEDGMENT

Not applicable in this study.

REFERENCES

[1] 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 *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 14–24, https://doi.org/10.1109/ICSE.2012.6227210.

[2] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2013, pp. 345–355, https://doi.org/10.1109/ASE.2013.6693093.

[3] C. P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept. 2014, pp. 181–190, https://doi.org/10.1109/ICSME.2014.40.

[4] S. Wang and D. Lo, "AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization," *Journal of Software: Evolution and Process*, vol. 28, no. 10, pp. 921–942, Oct. 2016, https://doi.org/10.1002/smr.1801.

[5] W. Zou, E. Li, and C. Fang, "BLESER: Bug Localization Based on Enhanced Semantic Retrieval." arXiv, Sept. 08, 2021, https://doi.org/10.48550/arXiv.2109.03555.

[6] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Information and Software Technology*, vol. 82, pp. 177–192, Feb. 2017, https://doi.org/10.1016/j.infsof.2016.11.002.

[7] Z. Shi, J. Keung, K. E. Bennin, and X. Zhang, "Comparing learning to rank techniques in hybrid bug localization," *Applied Soft Computing*, vol. 62, pp. 636–648, Jan. 2018, https://doi.org/10.1016/j.asoc.2017.10.048.

[8] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Information and Software Technology*, vol. 105, pp. 17–29, Jan. 2019, https://doi.org/10.1016/j.infsof.2018.08.002.

[9] S. Sangle, S. Muvva, S. Chimalakonda, K. Ponnalagu, and V. G. Venkoparao, "DRAST -- A Deep Learning and AST Based Approach for Bug Localization." arXiv, Nov. 06, 2020, https://doi.org/10.48550/arXiv.2011.03449.

[10] S. B. Hossain *et al.*, "A Deep Dive into Large Language Models for Automated Bug Localization and Repair," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1471–1493, July 2024, https://doi.org/10.1145/3660773.

[11] Z. Feng *et al.*, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." arXiv, Sept. 18, 2020, https://doi.org/10.48550/arXiv.2002.08155.

[12] D. Guo *et al.*, "GraphCodeBERT: Pre-training Code Representations with Data Flow." arXiv, Sept. 13, 2021, https://doi.org/10.48550/arXiv.2009.08366.

[13] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified Cross-Modal Pre-training for Code Representation." arXiv, Mar. 08, 2022, https://doi.org/10.48550/arXiv.2203.03850.

[14] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation." arXiv, Sept. 02, 2021, https://doi.org/10.48550/arXiv.2109.00859.

[15] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4BL: reproducibility study on the performance of IR-based bug localization," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2018, pp. 61–72, https://doi.org/10.1145/3213846.3213856.

[16] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with BERT," in *Proceedings of the 44th International Conference on Software Engineering*, May 2022, pp. 946–957, https://doi.org/10.1145/3510003.3510042.

[17] M. Chandramohan, D. Q. Nguyen, P. Krishnan, and J. Jancic, "Supporting Cross-language Cross-project Bug Localization Using Pre-trained Language Models." arXiv, July 03, 2024, https://doi.org/10.48550/arXiv.2407.02732.

[18] F. Niu, C. Li, K. Liu, X. Xia, and D. Lo, "When Deep Learning Meets Information Retrieval-based Bug Localization: A Survey." arXiv, Apr. 30, 2025, https://doi.org/10.48550/arXiv.2505.00144.

[19] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proceedings of the 44th International Conference on Software Engineering*, May 2022, pp. 1169–1180, https://doi.org/10.1145/3510003.3510147.

[20] D. Chhabra and R. Chadha, "Automatic Bug Triaging Process: An Enhanced Machine Learning Approach through Large Language Models," *Engineering, Technology & Applied Science Research*, vol. 14, no. 6, pp. 18557–18562, Dec. 2024, https://doi.org/10.48084/etasr.8829.

[21] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2019, pp. 169–180, https://doi.org/10.1145/3293882.3330574.

[22] M. N. Rafi, A. R. Chen, T. H. P. Chen, and S. Wang, "Revisiting Defects4J for Fault Localization in Diverse Development Scenarios," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, Apr. 2025, pp. 63–75, https://doi.org/10.1109/MSR66628.2025.00022.

[23] X. Zhou, D. Han, and D. Lo, "Assessing Generalizability of CodeBERT," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept. 2021, pp. 425–436, https://doi.org/10.1109/ICSME52107.2021.00044.

[24] S. Alsaedi, A. A. A. Gad-Elrab, A. Noaman, and F. Eassa, "Two-Level Information-Retrieval-Based Model for Bug Localization Based on Bug Reports," *Electronics*, vol. 13, no. 2, Jan. 2024, Art. no. 321, https://doi.org/10.3390/electronics13020321.

[25] S. Rathi, N. L. B. Murthy, and L. Kumar, "An empirical evaluation of the effectiveness of various ML, DL, and CodeBERT models to enhance the quality of software with the application of AST and Embedding techniques," in *Proceedings of the 18th Innovations in Software Engineering Conference*, Feb. 2025, pp. 1–4, https://doi.org/10.1145/3717383.3717401.

[26] M. Yang and J. Hu, "Precise Learning-to-Rank Bug Localization Using Multi-Feature Fusion for Hardware Code," *ACM Transactions on Design Automation of Electronic Systems*, vol. 31, no. 2, pp. 1–25, Mar. 2026, https://doi.org/10.1145/3779450.

[27] Z. Luo, W. Wang, and C. Cen, "Improving Bug Localization With Effective Contrastive Learning Representation," *IEEE Access*, vol. 11, pp. 32523–32533, 2023, https://doi.org/10.1109/ACCESS.2022.3228802.

[28] M. Asad, R. M. Yasir, and S. Malek, "Towards Explorative IRBL: Combining Semantic Retrieval with LLM-driven Iterative Code Exploration." arXiv, Apr. 21, 2026, https://doi.org/10.48550/arXiv.2508.00253.

[29] Y. Li *et al.*, "A Knowledge Enhanced Large Language Model for Bug Localization," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 1914–1936, June 2025, https://doi.org/10.1145/3729356.

[30] P. Chakraborty, M. Alfadel, and M. Nagappan, "BLAZE: Cross-Language and Cross-Project Bug Localization via Dynamic Chunking and Hard Example Learning," *IEEE Transactions on Software Engineering*, vol. 51, no. 8, pp. 2254–2267, Aug. 2025, https://doi.org/10.1109/TSE.2025.3579574.

[31] G. Nam and G. Yang, "LLMLoc: A Structure-Aware Retrieval System for Zero-Shot Bug Localization," *Electronics*, vol. 14, no. 21, Nov. 2025, Art. no. 4343, https://doi.org/10.3390/electronics14214343.

[32] V. N. Subramanian, "BugLLM: Explainable Bug Localization through LLMs," M.S. Thesis, University of Waterloo, Canada, 2024.