CodeFix-Bench: A Large-scale Benchmark for Learning to Localize Code Changes from Issue Reports

Faizan Khan *
SlashML Corp.
Montreal, Canada
faizank@slashml.com

Azeem Ghumman *
SlashML Corp.
San Francisco, US
azeem@slashml.com

***Abstract*—Software development teams increasingly rely on bots to automate routine tasks, yet current bots struggle to effectively suggest which files need modification when addressing issues or implementing new features. We present CodeFix-Bench, a comprehensive benchmark for developing and evaluating code change localization bots, built upon the SWE-benchmark dataset of real-world GitHub issues and pull requests. Our benchmark provides 2,294 high-quality instances where each input consists of an issue description and complete codebase, and the task is to identify files that require modification to resolve the issue.**

As a baseline implementation, we evaluate several traditional information retrieval approaches (BM25, TF-IDF, VSM-COS) on this task. Our experiments reveal that while BM25 achieves promising results Top-5 accuracy of 77.78%, significant challenges remain in handling large codebases (MAP drops to 0.59 from 0.63) and understanding implicit code dependencies. Analysis of failure cases highlights opportunities for more sophisticated bot implementations that could incorporate code structure, historical patterns, and developer feedback.

This paper makes three key contributions: (1) a carefully curated benchmark for evaluating change localization bots, (2) strong baseline implementations using traditional IR methods, and (3) detailed analysis of current limitations that can guide future bot development. Our benchmark and findings provide a foundation for creating more effective bots that can assist developers in navigating and modifying large codebases.

***Index Terms*—software bots, code change localization, information retrieval, benchmark dataset, software engineering**

I. Introduction

Software maintenance consumes substantial engineering resources, with developers spending significant time locating the relevant code that needs to be modified to address reported issues. Studies estimate that developers spend 50-60% of their time understanding existing code [1], [2] and navigating codebases to identify where changes need to be made.

While issue tracking systems provide valuable documentation of bugs and feature requests [3], manually translating these natural language descriptions into precise locations in source code remains a labor-intensive process. This translation requires deep understanding of both the issue description and the system architecture, making it a significant bottleneck in the software development lifecycle [4]. Furthermore, as codebases grow in size and complexity, the cognitive load on developers to maintain this comprehensive understanding increases dramatically [5].

Recent advances in machine learning, particularly in natural language processing and code understanding [6], suggest the potential for automated approaches to this problem. However, progress in developing effective automated bug localization tools has been hampered by the lack of large-scale, high-quality benchmarks that pair issue reports with their corresponding code changes [7]. Existing datasets are often limited in size, scope, or quality, making it difficult to train and evaluate modern deep learning approaches effectively.

In this paper, we present CodeFix-Bench, a comprehensive benchmark dataset designed specifically for training and evaluating models that can automatically identify relevant code locations from issue reports. Our benchmark consists of 2,294 issue reports paired with their corresponding code changes, collected from 12 popular open-source projects. Each instance in our dataset includes the complete issue report, the state of the codebase at the time of the report, and ground truth annotations indicating the defect-fixing files.

The creation of CodeFix-Bench addresses several key challenges in the field:

- **Scale:** By providing a dataset of unprecedented size, we enable the training of sophisticated deep learning models
- **Quality:** Through careful curation and validation, we ensure the reliability of the mapping between issues and code changes
- **Diversity:** Our dataset spans multiple project domains and types of issues.

We demonstrate the utility of CodeFix-Bench through extensive experiments with several baseline approaches, including traditional information retrieval techniques [8]. Our findings have several implications for tool developers and researchers:

- Traditional IR methods provide a surprisingly strong baseline, suggesting they should be considered as components in more sophisticated approaches
- The significant performance drop for large code bases indicates a need for methods that better retain context.

- The correlation between issue description length and performance suggests potential benefits from guided issue reporting practices

II. Related Work

Bug Localization Benchmarks. Several works have proposed benchmarks for evaluating bug localization techniques, often focusing on specific aspects like single-file changes [8] or particular programming languages [9]. While valuable, these benchmarks typically provide simplified scenarios that don't capture the full complexity of real-world software development. In contrast, CodeFix-Bench builds upon the comprehensive SWE-benchmark [10], transforming it into a dedicated bug localization dataset that preserves the real-world complexity of identifying change locations across large codebases with multiple files and intricate dependencies.

Information Retrieval in Software Engineering. Traditional IR techniques have been widely applied in software engineering tasks [11], [12]. These approaches range from basic text similarity measures to more sophisticated probabilistic models [13]. While recent work has explored neural approaches [14], [15], the scalability and interpretability of traditional IR methods make them particularly valuable for code analysis. Our work provides a comprehensive evaluation of these classical approaches on a modern, real-world benchmark, establishing strong baselines for future research.

Change Location Prediction. Prior research has explored various approaches to predict which files need modification based on issue reports or change requests. These include:

- Developer expertise models [5]
- Dependency-based approaches [3]
- Natural language processing techniques [7]

However, most existing work has been limited by the availability of high-quality, large-scale datasets. CodeFix-Bench addresses this gap by providing a benchmark derived from real-world software projects, complete with verified ground truth from successful pull requests.

ML for Software Engineering. Recent advances in machine learning have led to increased interest in automating various software engineering tasks [16]. While much attention has focused on code generation [17] and program repair [18], the crucial task of change localization has received less attention. Our work bridges this gap by providing a benchmark designed for evaluating change localization approaches, whether based on traditional IR methods or modern ML techniques.

Repository-Scale Analysis. Most existing benchmarks focus on function-level or file-level tasks [19], [20], limiting their applicability to real-world scenarios where changes often span multiple files and require understanding of broader project context. By building on SWE-benchmark, CodeFix-Bench preserves the repository-scale complexity of software changes while focusing specifically on the localization aspect. This makes it particularly valuable for developing and evaluating tools that can handle the scale and complexity of real-world software projects.

Reproducibility and Evaluation. The software engineering community has increasingly emphasized the importance of reproducible research and standardized evaluation metrics [21]. CodeFix-Bench contributes to this effort by providing:

- Clear evaluation metrics adapted for the localization task
- Baseline implementations of traditional IR approaches
- Detailed analysis of different types of changes and their impact on localization difficulty

Our benchmark thus provides a foundation for systematic evaluation and comparison of different approaches to code change localization, while maintaining the practical relevance of the task through its connection to real-world software development practices.

III. CodeFix-Bench

CodeFix-Bench builds upon the SWE-benchmark introduced by Jimenez et al. [10], which provides a comprehensive collection of GitHub issues paired with their resolving pull requests. We chose SWE-benchmark as our foundation for several key reasons:

- **Quality-Assured Data**: Their rigorous three-stage filtering pipeline ensures high-quality issue-PR pairs, with verified test coverage and execution validation
- **Scale**: The benchmark includes 2,294 carefully curated tasks across 12 popular Python repositories, providing a robust dataset.
- **Diversity**: The dataset not only varies in size, but also in their domain of application, ranging from web-dev (django) to data-science (scikit-learn).
- **Rich Context**: Each instance contains complete codebase snapshots and detailed issue descriptions, essential for our localization task

While SWE-benchmark was designed to evaluate end-to-end fix generation, we repurpose it to create a new benchmark specifically for the code change localization task. For each issue-PR pair in SWE-benchmark, we extract the following:

- The original issue report (title and description)
- All of the comments on the issue report
- The complete codebase at the time of issue report creation
- The set of files modified in the resolving pull request as ground truth

A. Task Formulation

Model Input. Models receive:

- An issue report (title plus description) in natural language
- A complete codebase at the time of issue report creation

Expected Output. Unlike SWE-benchmark, which expects complete code changes, our task requires models to produce:

- A set of files that should be modified to resolve the issue
- Confidence scores for each predicted file

Evaluation Metrics. We evaluate models using:

- **Top-k Accuracy**: Percentage of instances where the correct files appear in the top k predictions
- **Mean Reciprocal Rank (MRR)**: Average position of the first correct file in the ranked list

- **Mean Average Precision (MAP)**: Overall precision of the ranked file list

B. Dataset Characteristics

By leveraging SWE-benchmark's data, our dataset inherits several valuable properties:

- **Diverse Change Patterns**: The ground truth includes both single-file changes and multi-file changes (average of 1.7 files per fix [10])
- **Real-world Scale**: Codebases contain thousands of files, making the localization task realistically challenging
- **Rich Context**: Issue descriptions average 195 words, providing detailed context for localization
- **Verified Solutions**: All ground truth files are verified through the original benchmark's test-based validation

IV. Experimental Setup

In this section, we describe our experimental methodology for evaluating code change localization using traditional information retrieval approaches. We justify our choice of methods and detail our evaluation setup.

A. Information Retrieval Methods

For our benchmark, we employ several well-established information retrieval techniques that have proven effective in software engineering tasks:

BM25 (Okapi). As our primary retrieval method, we use BM25 [13], which has consistently demonstrated robust performance in software-related retrieval tasks [8]. BM25 extends the TF-IDF framework with better term frequency saturation and document length normalization, making it particularly suitable for comparing issue descriptions (queries) against source code files (documents).

TF-IDF. We implement the classic TF-IDF (Term Frequency-Inverse Document Frequency) approach [22] as a baseline. Despite its simplicity, TF-IDF remains competitive in software engineering applications [11], especially when dealing with code-specific terminology.

VSM-COS. We also evaluate the Vector Space Model with cosine similarity [23], which has been successfully applied in bug localization [12]. This method effectively captures the semantic similarity between issue descriptions and code files.

Our choice of these traditional retrieval methods is motivated by several factors:

- **Scalability**: Unlike dense retrieval methods that struggle with long documents, these approaches efficiently handle large codebases [24]
- **Interpretability**: The results from these methods are easily interpretable, allowing developers to understand why certain files were ranked higher [25]
- **Language Agnosticism**: These methods work effectively across different programming languages without requiring language-specific training [26]
- **Low Resource Requirements**: They can be deployed with minimal computational resources compared to neural approaches [7]

B. Implementation Details

For each issue in CodeFix-Bench, we :

1) **Preprocessing**: Filter out hidden, empty, test-directory, and non-code files from the GitHub repository
2) **Index Construction**: Generate document-term matrices and similarity scores as artifacts to support retrieval with TF-IDF, VSM-COS, and BM25. Each file is indexed as a document, creating an inverted index of the codebase
3) **Query Processing**: Tokenize issue descriptions and remove common and code-specific stop words
4) **Ranking**: Rank relevant files using TF-IDF, VSM-COS, BM25, and historical changes, with each method producing ranked lists

C. Evaluation Metrics

We evaluate the performance of our approaches using three complementary metrics that capture different aspects of localization effectiveness:

- **Mean Average Precision (MAP)** [23]: Measures the quality of file ranking in the context of multi-file changes. As issues often require modifications to several files (average 1.7 files per change in our dataset), MAP evaluates both the completeness of finding all necessary files and their ranking positions. For example, if an issue requires changes to three files, a system ranking them at positions [1,2,3] will score higher than one ranking them at [1,10,20], even though both found all relevant files.
- **Mean Reciprocal Rank (MRR)** [27]: Evaluates the system's ability to identify a good starting point for code investigation. When developers begin addressing an issue, finding at least one relevant file quickly is crucial for understanding the change context. MRR focuses on the position of the first relevant file in the ranked list, calculated as the reciprocal of its position (1/rank).
- **Top-k Accuracy** [8]: Measures the practical usability of the system by calculating the percentage of issues where at least one relevant file appears in the top k suggestions. We evaluate with k = $\{1, 3, 5\}$, reflecting realistic developer behavior where examining more than 5 files per issue becomes cognitively expensive and time-consuming. Top-k is particularly relevant because developers typically work with limited attention spans and resources; they are unlikely to scroll through lengthy result lists. For top-k we only care about one thing i.e. Is there at least one file from the gold standard in the top-k predictions

D. Baseline Comparisons

To provide context for our results, we implement two additional baselines:

Random Baseline. A random file selector that helps establish a lower bound for performance.

History-based Baseline. A simple approach that ranks files based on their historical modification frequency [28], providing an intuitive comparison point.

Table I: Performance comparison of different retrieval methods on CodeFix-Bench

Method	MAP	MRR	Top-1	Top-3	Top-5
Random	0.04	0.05	1.78%	3.56%	6.22%
History-based	0.22	0.24	14.67%	23.11%	32.00%
TF-IDF	0.40	0.41	22.67%	52.44%	62.67%
VSM-COS	0.34	0.35	17.78%	42.67%	55.56%
BM25	**0.58**	**0.60**	**46.67%**	**68.89%**	**77.78%**

Table II: BM25 performance breakdown by issue and codebase characteristics

Category	Subcategory	MAP	Top-1	Top-3	Top-5
Issue Length	Short (<250 words)	0.57	44.23%	**71.15%**	**79.81%**
	Long (≥250 words)	**0.59**	**48.76%**	66.94%	76.03%
Codebase Size	Small (<500 files)	**0.63**	**50.64%**	**72.44%**	**81.41%**
	Large (≥500 files)	0.49	37.68%	60.87%	69.57%
Change Scope	Single-file	0.53	37.90%	62.10%	71.77%
	Multi-file	**0.65**	**57.43%**	**77.23%**	**85.15%**

V. Results and Analysis

We present our experimental results and analyze the performance of different retrieval approaches for code change localization. Our analysis reveals several key insights about the effectiveness of traditional IR methods in this domain.

A. Overall Performance

Table I presents the main results across all retrieval methods. BM25 consistently outperforms other approaches across all metrics, achieving a MAP of 0.58 and MRR of 0.6. This suggests that BM25's term saturation and length normalization properties are particularly beneficial for matching issue descriptions with relevant code files.

B. Impact of Issue and Codebase Characteristics

We analyze how different characteristics of issues and codebases affect localization performance. Table II shows BM25 performance across different categories.

Key findings include:

- **Issue Description Length**: Longer issue descriptions ($\geq$ 250 words) lead to a slightly better localization performance (MAP: 0.59 vs 0.57), likely due to richer contextual information.
- **Codebase Size**: Performance degrades with larger codebases, with MAP dropping from 0.63 for small codebases to 0.49 for large ones, highlighting the challenge of scale.
- **Change Scope**: Multi-file changes yield better localization (MAP: 0.65) compared to single-file changes (MAP: 0.53), suggesting that additional context from related files enhances the model's ability to identify relevant changes.

C. Retrieval Efficiency

Table III compares the computational requirements of different methods. All methods demonstrate practical efficiency and reasonable memory requirements, making them suitable for real-world deployment, with BM25 being the most expensive, particularly because of the need to calculate term saturation and document-specific weighting factors, which increase computational overhead compared to TF-IDF and VSM-COS

Table III: Computational efficiency comparison

Method	Index Time (s)	Query Time (ms)	Memory (MB)
TF-IDF	**11.66**	110	**663**
VSM-COS	23.67	**80**	850
BM25	75.24	720	896

VI. Discussion

Limitations and Future Directions.

- **Language Coverage**: Currently, our benchmark is derived from Python projects in SWE-benchmark. Future work should extend this to other programming languages, as localization patterns and challenges may vary across languages and their ecosystems.
- **Basic IR Approaches**: Our experiments establish baselines using traditional IR methods like BM25 and TF-IDF. While these provide strong baselines, they don't capture structural code information or semantic relationships. Future work could explore:
  - Hybrid approaches combining IR with static analysis
  - Fine-tuning a code specific langauge model such as llama-coder.
- **Evaluation Metrics**: While our current metrics (MAP, Top-k) are standard in IR, they may not fully capture all aspects of localization. Additional metrics could consider:
  - Function-level and even syntatical-block level granularity within files
  - Relationship between multiple changed files

Practical Implications. Our findings have several implications for tool developers and researchers:

- Traditional IR methods provide a surprisingly strong baseline, suggesting they should be considered as components in more sophisticated approaches
- The significant performance drop for large code bases indicates a need for methods that better retain context.
- The correlation between issue description length and performance suggests potential benefits from guided issue reporting practices

Conclusion. Code change localization remains a critical challenge, extending beyond simple code search or bug localization. By repurposing the comprehensive SWE-benchmark dataset, CodeFix-Bench provides a realistic evaluation environment that captures the complexity of real-world software development. Our evaluation of traditional IR approaches establishes strong baselines while highlighting specific challenges that more advanced techniques must address.

The complete codebase, dataset, and documentation can be found at https://github.com/AzeemGhumman/CodeFix-Bench

REFERENCES

[1] A. J. Ko, B. A. Myers, and M. J. Coblenz, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[2] A. Hora, "Googling for software development: What developers search for and what they find," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 317–328.

[3] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: Association for Computing Machinery, 2008, p. 308–318. [Online]. Available: https://doi.org/10.1145/1453101.1453146

[4] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang, "An empirical study on bug assignment automation using chinese bug data," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 451–455.

[5] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "Using psycho-physiological measures to assess task difficulty in software development," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 402–413.

[6] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2103.06333

[7] G. Giray, K. E. Bennin, Ömer Köksal, Önder Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," 2022. [Online]. Available: https://arxiv.org/abs/2210.02236

[8] 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)*. IEEE, 2012, pp. 14–24.

[9] Y. Kim, S. Mun, S. Yoo, and M. Kim, "Precise learn-to-rank fault localization using dynamic and static features of target programs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–34, 2019.

[10] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" 2024. [Online]. Available: https://arxiv.org/abs/2310.06770

[11] V. S. Jayapati and A. Venkitaraman, "A comparison of information retrieval techniques for detecting source code plagiarism," 2019. [Online]. Available: https://arxiv.org/abs/1902.02407

[12] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Inf. Softw. Technol.*, vol. 52, no. 9, p. 972–990, Sep. 2010. [Online]. Available: https://doi.org/10.1016/j.infsof.2010.04.002

[13] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends in Information Retrieval*, vol. 3, pp. 333–389, 01 2009.

[14] S. Chakraborty, Y. Li, M. Irvine, R. Saha, and B. Ray, "Entropy guided spectrum based bug localization using statistical language model," *arXiv preprint arXiv:1802.06947*, 2018.

[15] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, and N. Sundaresan, "Automating code review activities by large-scale pre-training," 2022.

[16] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," 2023.

[17] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, and J. K. et. al, "Evaluating large language models trained on code," 2021.

[18] T. Dinh, J. Zhao, S. Tan, R. Negrinho, L. Lausen, S. Zha, and G. Karypis, "Large language models of code fail at completing code with potential bugs," *arXiv preprint arXiv:2306.03438*, 2023.

[19] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, T. Xie, and Q. Wang, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," 2023.

[20] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," 2023.

[21] S. R. Bowman and G. E. Dahl, "What will it take to fix benchmarking in natural language understanding?" 2021.

[22] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing Management*, vol. 24, no. 5, pp. 513–523, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0306457388900210

[23] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.

[24] M. M. Rahman and C. K. Roy, "Improving bug localization with report quality dynamics and query reformulation," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 348–349. [Online]. Available: https://doi.org/10.1145/3183440.3195003

[25] Y. Yang, Z. Wang, Z. Chen, and B. Xu, "Cops: An improved information retrieval-based bug localization technique using context-aware program simplification," *Journal of Systems and Software*, vol. 207, p. 111868, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223002637

[26] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, "Comparing bug finding tools with reviews and tests," *ArXiv*, vol. abs/1711.05019, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:8376907

[27] E. M. Voorhees and D. M. Tice, "The TREC-8 question answering track," in *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC'00)*, M. Gavrilidou, G. Carayannis, S. Markantonatou, S. Piperidis, and G. Stainhauer, Eds. Athens, Greece: European Language Resources Association (ELRA), May 2000. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2000/pdf/26.pdf

[28] A. Hassan and R. Holt, "Predicting change propagation in software systems," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 284–293.