BLADE: Bug Localization Assisted Debugging Engine

Pooria Roy[1,3], Mariam El Mezouar[2,3]

[1]School of Computing, Queen's University, Kingston, Canada
[2]Faculty of Arts and Science, Queen's University, Kingston, Canada
[2]Department of Computer Science, Royal Military College of Canada, Kingston, Canada

1 Introduction

Recent advances in Large Language Models (LLMs) have introduced new possibilities for Automated Program Repair (APR). By understanding code context and iteratively testing their own fixes, LLMs can reduce the need for human intervention. This becomes more prominent in smaller projects, where the limited number of files and lines of code allows the entire context to fit within the model's input window.

However, this advantage diminishes as the size and complexity of a project grow. While larger codebases may technically fit within the context window, effective bug localization requires more than surface-level pattern recognition. It demands a deeper, semantic understanding of program behavior and structure something current LLMs are still not fully equipped to handle.

To address these limitations, we propose a pipeline that localizes bugs without relying on LLMs. By analyzing the codebase in tandem with the error trace, our method can more precisely identify the source of a bug. It comes within 3% of the performance of Blaze (the current state-of-the-art). In our evaluation, this approach outperforms strategies that simply feed the full codebase to an LLM, offering a more scalable solution to the problem.

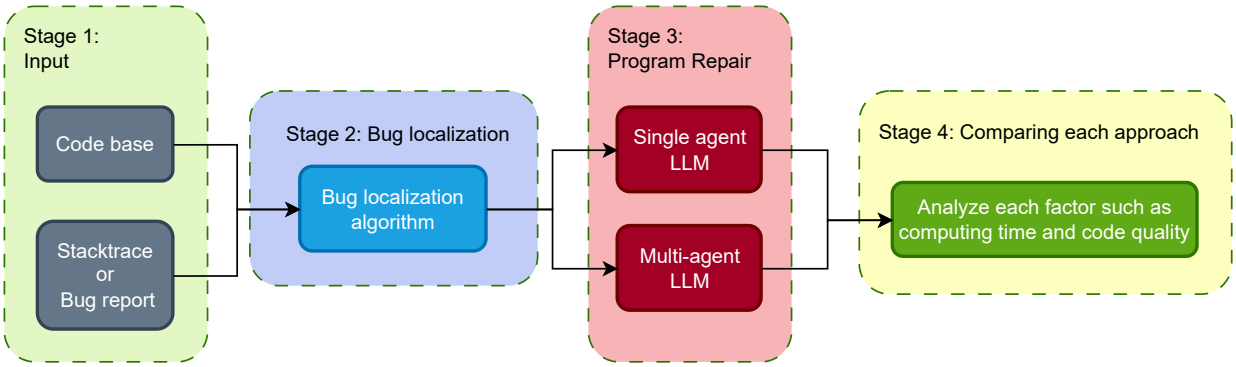

Figure 1. APR Pipeline

2 Related Work

Recent approaches to bug localization increasingly use code embeddings to represent both source files and failure contexts, such as stack traces or bug reports in a shared semantic space for similarity-based retrieval.

Early models like the DNN+IR hybrid from Lam et al. (2017) combined keyword-based search with a deep neural network, achieving around 85% Top-20 accuracy. xLoc (Yang et al., 2024) leverages a multilingual Transformer to classify buggy functions and localize errors, reaching 87.54% Top-1 accuracy.

BLAZE (Chakraborty et al., 2024) uses the CodeSage model Zhang et al. (2024) to embed entire codebases and bug reports. It applies dynamic chunking and combines semantic and lexical retrieval to support cross-project localization, achieving 60-70% Top-20 accuracy.

BugLLM (Subramanian, 2024) takes a zero-shot approach, generating natural language queries with an LLM and retrieving relevant code chunks via embeddings. It reports 44.7-61.1% Top-5 accuracy, with explanation rated highly for clarity and correctness.

3 Inputs

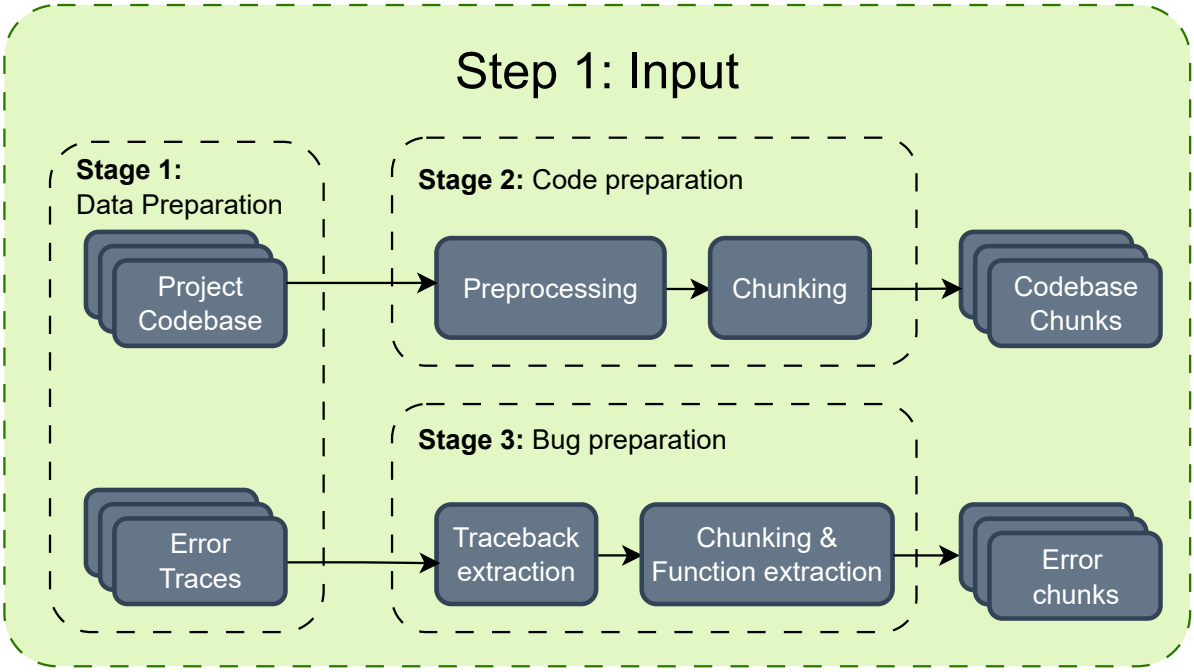

Figure 2. Input processing

3.1 Data Preparation

Selecting an appropriate dataset is key to evaluating the pipeline. We use an enhanced version of the BugsInPy dataset (Aguilar et al., 2023), which improves reproducibility by providing validated Docker environments for each bug. Projects were selected based on criteria that ensures diversity and sufficient number of bug cases. An overview of the selected projects is provided in Table 1

Projects	Bugs	Avg. Trace LoC	Min. Trace LoC	Max. Trace LoC	File count	Avg. LoC	Min. LoC	Max. LoC
matplotlib	23	91.2	49	232	762	244.3	1	8054
pandas	124	261.2	30	4375	297	602.8	1	11407
youtube-dl	31	26.3	2	56	809	159.7	4	3888
luigi	30	84.2	27	319	119	261.3	17	1647
black	19	138.5	18	591	23	4134.5	2	41441
scrapy	37	33.7	18	133	177	100.5	1	531
thefuck	30	84.8	27	363	198	33.6	1	338
keras	34	375.9	79	3476	135	362.8	1	4494
ansible	135	74.8	12	246	582	236.9	1	3446

Table 1
Projects chosen from BugsInPy dataset

3.2 Code preparation

Before embedding, we preprocess each project by cloning its repository and filtering out non-source files (e.g., tests, configuration scripts, and data). To handle long source files and improve semantic focus, we apply an AST-based chunking strategy. Using Python's built-in Abstract Syntax Tree (AST) module, we split each file into logical units corresponding to top-level classes and functions. This reduces noise from unrelated code and ensures that the resulting chunks are semantically meaningful.

3.3 Bug preparation

As part of bug preprocessing, we extract the relevant segments of each tracebacks by removing non-informative frames, such as logging calls and third-party utility layers. This refinement improves the overall success rate by 3%. To ensure clear evaluation targets, we include

only bugs where exactly one file is modified, based on the available ground truth.

4 Bug Localization

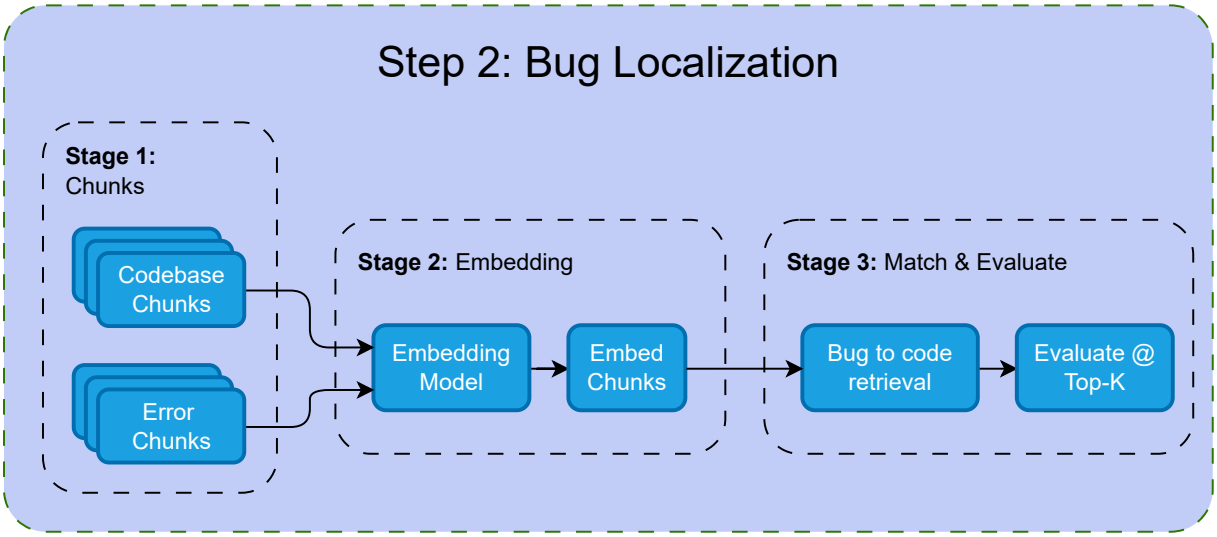

Figure 3. Bug localization pipeline

4.1 Model Selection and Performance

Empirically, we find that smaller models trained specifically on code embeddings outperform both general-purpose language models and larger code models in our localization task. Table 3 summarizes the top-k accuracy across several embedding models. We adopt the `blaze` model due to its strong performance and efficient resource usage; It supports batch sizes up to 128 on an 8GB GPU, compared to over 15GB required by larger models such as `jinaai/jina-embedding-v2-base-code` at batch size 2. Earlier experiments with `MiniLM-L12-v2` served as a baseline, yielding 60% Top-5 accuracy.

4.2 Retrieval Method

For retrieval, we compute cosine similarity between the embedded bug trace and each code chunk, returning the top-k most similar candidates. As shown in Figure **??**, performance improves significantly between $k = 1$ and $k = 10$.

4.3 Dataset Characteristics

We also observe that localization accuracy declines in projects with larger file counts. Table 1 highlights that `pandas`, `matplotlib`, `ansible`, and `youtube-dl` have substantially larger codebases, and three of the four have reduced performance compared to other projects. This suggests that increasing the search space negatively impacts retrieval precision, even when using semantically rich embeddings.

4.4 Evaluation Bias

Table 4 presents top-k accuracy based on the file name, while Table5 reports performance based on the function name.

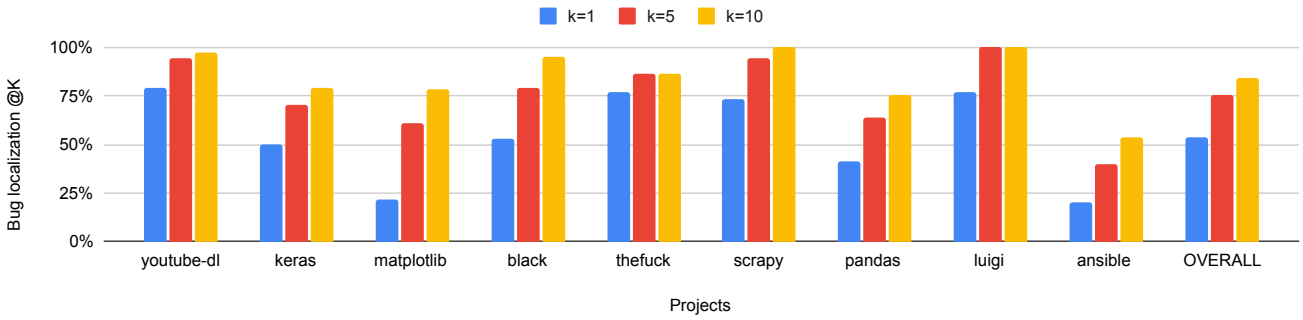

Figure 4. File level bug localization accuracy @k for "blaze" a fine-tuned version of "codesage/codesage_small_v2"

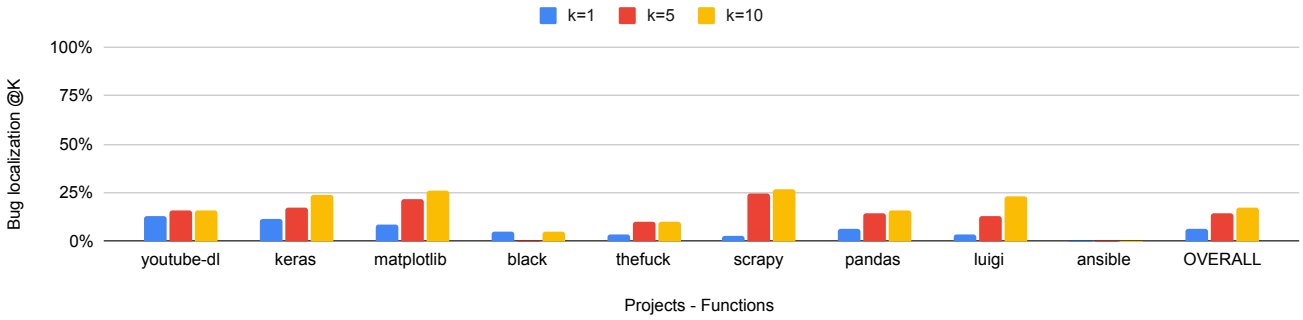

Figure 5. Function level bug localization accuracy @k for "blaze" a fine-tuned version of "codesage/codesage_small_v2"

Model	Top-1	Top-5	Top-10	MAP@60	MRR@60
BLAZE	53.76%	75.43%	84.10%	40.81%	63.91%
all-MiniLM-L12-v2	39.94%	63.85%	72.89%	32.15%	51.11%

Table 2
Overall success rate @k for different models at the file level

Model	Top-1	Top-5	Top-10	MAP@60	MRR@60
BLAZE	?%	?%	?%	?%	?%

Table 3
Overall success rate @k for different models at the function level

5 APR

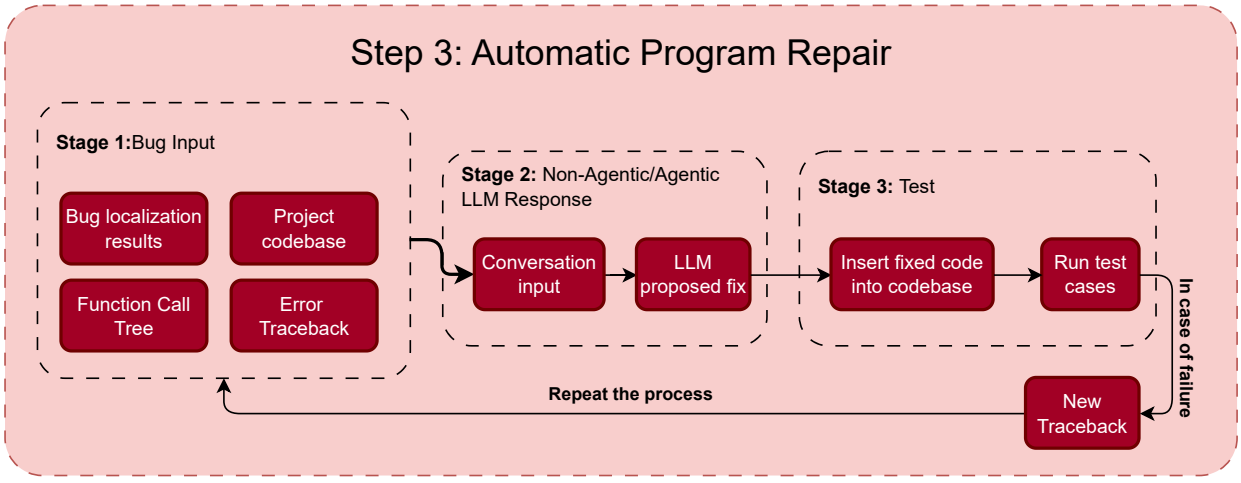

Figure 6. APR pipeline

LLMs are sensitive to the size and relevance of the input context. In practice, we found that providing entire source files—especially in legacy codebases where individual files may exceed 2,000 lines, substantially degrades the quality of repair suggestions. This is due to the dilution of relevant signal within overly broad inputs, often resulting in incoherent or incorrect fixes.

To address this limitation, we developed a targeted context-construction strategy that leverages both the error traceback and static code structure.

5.1 Function Extraction from Error Traceback (Functional Call Tree)

When a test case fails, we begin by analyzing the raw error traceback to identify all function calls that led to the failure. These functions are assumed to be highly relevant to the bug location. For each function, we retrieve the corresponding source code and extract any directly nested function calls within it (i.e., a call depth of 1). This process builds a partial call graph centered around the failing path.

5.2 Reducing Context with Bug Localization (Not implemented yet)

To further refine the set of candidate functions, we incorporate insights from the bug localization module. Functions that are deemed irrelevant based on localization confidence scores are discarded, ensuring that only high-signal code snippets are retained. The resulting context—comprising the error traceback and filtered function bodies is then provided to the LLM.

5.3 Iterative Repair and Retesting

The LLM generates a proposed fix based on the curated input. This fix is inserted directly into the codebase at the appropriate location. The modified code is then recompiled and retested. If the test fails again, the process is repeated using the new error traceback, allowing for up to three iterations. If no successful fix is found after three attempts, the process is terminated.

5.4 Results

Project	First Pass	Second Pass	Third Pass
youtube-dl	48.64%	62.16%	72.22%

Table 4
Bug repair accuracy across multiple passes for the projects

6 Reproducibility

All code, scripts, and instructions required to reproduce our experiments are available in the BLADE repository[1]. Data preparation and analysis scripts are included, along with detailed usage examples and Colab notebooks.

[1] `https://github.com/regularpooria/BLADE`

Acknowledgments

This research was conducted at the Royal Military College of Canada (RMC), Kingston, under the supervision of Professor Dr. El Mezouar. It was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Aguilar, Faustino, Samuel Grayson, and Darko Marinov (2023). "Reproducing and Improving the BugsInPy Dataset". In: *Proceedings of the 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, pp. 260–266. DOI: 10.1109/SCAM59687.2023.00036.

Chakraborty, Partha, Mahmoud Alfadel, and Meiyappan Nagappan (2024). "BLAZE: Cross-Language and Cross-Project Bug Localization with CodeSage Embeddings". In: *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*. To appear.

Lam, An Ngoc et al. (2017). "Bug Localization with Combination of Deep Learning and Information Retrieval". In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pp. 218–229. DOI: 10.1109/ICPC.2017.24.

Subramanian, Poorna (2024). "BugLLM: Explainable Zero-Shot Bug Localization using Large Language Models". Master's thesis. University of Waterloo. URL: https://dspacemainprd01.lib.uwaterloo.ca/server/api/core/bitstreams/9586bd26-7a3c-4e56-8c75-51bfd6b0d082/content.

Yang, Haoran et al. (2024). "Learning to Detect and Localize Multilingual Bugs". In: *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. To appear. ACM.

Zhang, Dejiao et al. (2024). *Code Representation Learning At Scale*. arXiv: 2402.01935 [cs.CL]. URL: https://arxiv.org/abs/2402.01935.