Towards Explorative IRBL: Combining Semantic Retrieval with LLM-driven Iterative Code Exploration

MOUMITA ASAD, University of California, Irvine, USA
RAFED MUHAMMAD YASIR, University of California, Irvine, USA
SAM MALEK, University of California, Irvine, USA

Information Retrieval-based Bug Localization (IRBL) aims to identify buggy source files for a given bug report. Traditional and deep learning-based IRBL techniques often suffer from vocabulary mismatch and dependence on project-specific metadata. In contrast, recent Large Language Model (LLM)-based approaches struggle to provide appropriate context to the model: they either restrict analysis to a fixed set of candidate files, overwhelm the model with repository-wide information, or rely on explicit bug report cues to guide context collection. To address these issues, we propose GenLoc, a technique that combines semantic retrieval with LLM-driven code-exploration functions to iteratively analyze the code base and identify buggy files. We evaluate GenLoc on three complementary benchmarks, including large-scale and recent Java datasets as well as the Python based SWE-bench Lite dataset. Results demonstrate that GenLoc substantially outperforms traditional IRBL, deep learning-based approaches and recent LLM-based methods, while also localizing bugs that other techniques fail to detect.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; • **Information systems** → **Information retrieval**.

Additional Key Words and Phrases: bug localization, debugging, large language model, information-retrieval

ACM Reference Format:
Moumita Asad, Rafed Muhammad Yasir, and Sam Malek. 2018. Towards Explorative IRBL: Combining Semantic Retrieval with LLM-driven Iterative Code Exploration. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Bug localization is the process of identifying the faulty or buggy locations in source code [36]. It is a crucial step in software debugging to ensure the quality of a software [85]. However, localizing bugs in practice is challenging, as large systems often incur a huge number of bugs everyday. For example, Mozilla receives around 300 new bug reports daily, which is overwhelming for the developers [8]. Moreover, given the size and complexity of large-scale software systems today, manually locating bugs is a time-consuming, tedious and an expensive task [84]. Consequently, there is a strong demand for automated bug localization techniques to ease the burden of developers and accelerate the debugging process [60].

Over the years, researchers have proposed multiple approaches for bug localization. Among these, Spectrum-based Bug Localization (SBBL) and Information Retrieval-based Bug Localization

Authors' Contact Information: Moumita Asad, University of California, Irvine, USA, moumitaa@uci.edu; Rafed Muhammad Yasir, University of California, Irvine, USA, ryasir@uci.edu; Sam Malek, University of California, Irvine, USA, malek@uci.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX, Woodstock, NY
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

(IRBL) are the most studied and widely recognized techniques [46, 60]. SBBL leverages runtime behavior (e.g., test coverage information) to analyze which parts of a program are exercised by passing and failing test cases, and ranks program elements based on their likelihood of being faulty [106]. In contrast, IRBL takes a bug report (a document that describes a software bug) as input and does not require bug-revealing test cases or execution traces [30]. Due to its simplicity, IR-based approaches have received growing attention in recent years.

IRBL approaches consider the bug report as a query and the software repository as a collection of documents (i.e., source files). Given a bug report, these approaches output a ranked list of source files based on their probability of being faulty. The goal is to rank the actual buggy files as high as possible. To calculate the similarity between bug reports and the source files, earlier techniques use Vector Space Model (VSM) [75, 104], Topic Modeling [48, 56], while later methods [10, 34, 35, 43, 44, 60, 90, 95, 96, 100] rely on machine learning and deep learning models. Many of these approaches also incorporate additional information such as historical bug-fix data. However, such information is not always available, particularly in new projects, limiting the generalizability of these methods. Furthermore, the effectiveness of these approaches is often limited by the vocabulary gap between bug reports and source code, as they tend to use different terminologies [46].

To address these limitations, recent studies [42, 70] have incorporated Large Language Models (LLMs) into IRBL. However, these approaches struggle with providing appropriate context to the model. These techniques (BRaIn [70], LLM-BL [42]) rely on an initial retrieval step that produces a fixed shortlist of candidate files and constrain the LLM to rank files solely within this subset. Consequently, the model cannot iteratively explore the code base or gather additional evidence, limiting its effectiveness when relevant context is missing. A closely related challenge also arises in LLM-based issue localization approaches. Although these methods aim to identify fine-grained program elements (e.g., function or statement), they rely on file-level localization as an intermediate step and thus inherit similar difficulties in providing appropriate context to the LLM. Some techniques (CoSIL [22], BugCerberus [12]) expose the model to large portions of the repository, overwhelming it with irrelevant information, while others (OrcaLoca [98], LocAgent [13], RepoSearcher [51]) depend heavily on explicit lexical cues in the bug report and perform poorly when such signals are weak. These limitations highlight the need for a technique that can handle lexically weak bug reports, reason over a small and relevant portion of a large code base without overwhelming the LLM's context and adaptively expand the search space by acquiring additional evidence.

In this context, we propose GenLoc (Generative AI-based Bug Localization), a novel IRBL technique that combines semantic retrieval with LLM-driven iterative code exploration to identify buggy files. Unlike many prior approaches [60, 104], GenLoc relies solely on the bug report and does not require any additional project-specific metadata or historical bug-fix data. At first, GenLoc embeds source code and retrieves semantically relevant files for a given bug report. Next, it leverages an LLM to analyze a bug report and further inspect these retrieved files, or explore additional files as needed. To enable this exploration, GenLoc provides the LLM with a set of functions (e.g., retrieving method signatures or method bodies) that support iterative analysis about which files are likely related to the reported bug. By combining semantic retrieval with LLM-guided code exploration, GenLoc reduces reliance on lexical cues, avoids the limitation of fixed candidate sets that may exclude the faulty file and prevents overwhelming the LLM with repository-wide context.

We evaluate GenLoc on three complementary benchmarks: two Java based datasets, including a large-scale benchmark of 9,097 bugs [95] and a recent GHRB (GitHub Recent Bugs) dataset designed for LLM-based IRBL evaluation [37] as well as a widely used Python benchmark (SWE-bench Lite) adopted by modern issue localization pipelines [4]. Results show that GenLoc consistently outperforms traditional IRBL techniques, deep learning-based approaches and recent LLM-based

baselines. On the large-scale benchmark, GenLoc improves Accuracy@1 by **at least 63%** compared to traditional and deep learning-based IRBL techniques. On the GHRB dataset, it outperforms state-of-the-art LLM-based IRBL methods by **at least 25%**. On SWE-bench Lite, GenLoc achieves **at least 15%** higher Accuracy@1 than recent LLM-based issue localization approaches when comparing their file-level localization stages. Furthermore, ablation studies show that GenLoc's performance gains arise from the complementary interaction between semantic retrieval and LLM-driven iterative code exploration.

In summary, this paper makes the following contributions:

- The introduction of a novel IRBL approach named GenLoc that integrates semantic retrieval with LLM-guided iterative code analysis.
- A comprehensive empirical evaluation of GenLoc on three complementary benchmarks, spanning large-scale and recent Java datasets as well as a widely used Python benchmark, with comparisons against traditional IRBL baselines, LLM-based IRBL techniques and the file-level localization stage of the state-of-the-art issue localization methods.
- A publicly accessible replication package to facilitate future research.

2 Background

This section positions GenLoc within the existing literature by clarifying the problem settings, assumptions and objectives of closely related research areas. Table 1 provides a high-level comparison of these areas (Spectrum-Based Bug Localization, Information Retrieval-Based Bug Localization, Issue Localization and Issue Resolution), which are discussed in detail in Sections 2.1–2.4. Section 2.5 then positions GenLoc within this landscape and Section 2.6 introduces the terminology used throughout the paper.

2.1 Spectrum Based Bug Localization (SBBL)

SBBL techniques leverage program execution information collected from passing and failing test cases to localize suspicious statements or lines [55, 71]. Recent work, including AutoFL [24], LLM4FL [63], CosFL [61], SoapFL [62], FlexFL [91] and FaR-Loc [72], has explored the use of LLMs in this setting. However, SBBL approaches assume the availability of failing test cases and require the faulty code be executed by at least one failing test. These assumptions limit its applicability when such tests are absent [64].

2.2 Information Retrieval Based Bug Localization (IRBL)

IRBL techniques aim to identify buggy source files based on bug reports. Unlike SBBL, IRBL has no access to execution traces or failing test cases to constrain the search space. Thus, it needs to search across the entire code base, making the task inherently more challenging. Existing IRBL approaches can be broadly divided into three categories based on their underlying methodology. The first category [26, 47, 54, 56, 66, 69, 73, 78, 79, 81, 83, 97, 104] uses **traditional IR models** such as Latent Dirichlet Allocation (LDA) and VSM for bug localization. The second category [7, 9–11, 14–16, 20, 28, 33–35, 41, 43, 49, 59, 60, 77, 89, 90, 94, 99, 100, 103] consists of **learning-based models** (i.e., machine learning and deep learning models). The third category consists of **reasoning-enhanced approaches** [42, 70] where LLMs are used as a reasoning component to analyze and rerank retrieved candidate files. However, these methods rely on an initial retrieval step that produces a fixed shortlist of candidate files and restrict the LLM to reranking within this pre-retrieved context. Since the LLM cannot iteratively explore the code base or collect additional evidence, their effectiveness is limited.

Table 1. Comparison of SBBL, IRBL, Issue Localization and Issue Resolution.

Dimension	**SBBL**	**IRBL**	**Issue Localization**	**Issue Resolution**
Goal	Focuses on identifying buggy statements/lines	Focuses on identifying buggy files	Identifies fine-grained change locations (functions or statements)	Fixes reported issues
Input	Test cases and source code	Bug report and source code	Issue report and source code	Issue report and source code (optionally test cases)
Issue Types	Bugs only	Bugs only	Mixed issues including bugs, feature requests, and refactoring	Mixed issues including bugs, feature requests, and refactoring
Output	Ranked list of statements/lines	Ranked list of files	Relevant code locations	Patch
Difference from IRBL	Requires execution traces and failing test cases, whereas IRBL relies solely on bug report	–	Targets heterogeneous issues and fine-grained code elements, using file localization only as an intermediate step	Primarily focus on patch generation and test-based validation, while often overlooking the bug localization [12]
Representative LLM-based Techniques	AutoFL [24], LLM4FL [63], CosFL [61], SoapFL [62], FlexFL [91], FaR-Loc [72]	LLM-BL [42], BRaIn [70]	CoSIL [22], BugCerberus [12], OrcaLoca [98], LocAgent [13], RepoSearcher [51]	MAGIS [74], SWE-Agent [92], OpenHands [80], RepoGraph [58], AutoCodeRover [101], SpecRover [68], LingmaAgent [50], Agentless [88], SWE-Debate [38]

2.3 Issue Localization

Issue localization focuses on identifying fine-grained code locations (e.g., methods or statements) relevant to a reported issue [21]. Unlike IRBL and SBBL, issue localization is not restricted to bug reports and considers heterogeneous issue types, including feature requests, performance optimizations and refactoring tasks. Although the ultimate goal of issue localization is fine-grained identification, locating relevant source files is a necessary intermediate step in all issue localization pipelines and largely determines the subsequent reasoning process. From the perspective of how relevant files are identified during localization, existing approaches in this area mainly fall into two categories. The first category consists of **iterative-reasoning methods** [13, 51, 98], which employ an LLM to iteratively navigate the code base using lexical cues from the issue report, such as file or method names. The second category includes **repository-wide exploration methods** [12, 22], which provide the LLM with broad, repository-level context (e.g., the entire repository structure).

2.4 Issue Resolution

Issue resolution focuses on fixing reported issues by generating patches. MAGIS [74], SWE-Agent [92], OpenHands [80], RepoGraph [58], AutoCodeRover [101], SpecRover [68], LingmaAgent [50], Agentless [88] and SWE-Debate [38] are some representative techniques in this domain. These approaches primarily focus on patch generation and test-based validation, while often overlooking the bug localization, which remains a crucial and challenging part of the debugging process [12].

2.5 Position of GenLoc and Motivating Example

GenLoc addresses the problem of bug localization under the Information Retrieval-based Bug Localization (IRBL) setting. It operates exclusively on textual information from bug reports and source code and does not assume the availability of execution traces or failing test cases, distinguishing it from SBBL. Moreover, GenLoc differs in both scope and objective from issue localization and

issue resolution approaches: it focuses specifically on identifying buggy source files for bug reports, rather than locating fine-grained code elements for heterogeneous issues or generating patches.

As discussed earlier, existing LLM-based IRBL approaches [42, 70] often fail to retrieve sufficient relevant context to support effective reasoning. Their reliance on a fixed initial shortlist prevents the LLM from incrementally expanding the search space or validating hypotheses through targeted code inspection, ultimately leading to incorrect localization results. For the bug shown in Fig. 1 from the Apache RocketMQ project[1], the correct buggy file is not included in the initial candidate shortlist produced by these approaches; consequently, the LLM has no opportunity to rerank it, making successful localization impossible.

Bug ID: 7027
Summary: [Bug] The proxy in the cluster mode returns null address when master is down
Description: When I connect to the clustered proxy to consume messages using the remoting protocol, the master broker happened to hang up. At this time, the address returned by the proxy was empty, which caused the request to be sent to the NameServer.
Steps to Reproduce: 1. Deploy two group of brokers, such as: 1. master: broker-a, slave: broker-a-s 2. master: broker-b, slave: broker-b-s 2. Deploy a proxy. 3. Create some topics in broker-a for testing, such as message producing and consuming. 4. Kill master broker-a.

Fig. 1. Bug Report from Apache RocketMQ Project.

Similarly, recent issue localization approaches [12, 13, 22, 51, 98] face complementary limitations when applied to IRBL. Iterative-reasoning methods struggle with semantically rich but lexically weak issue reports, where little explicit information is available to reliably steer exploration. In contrast, repository-wide exploration methods may overwhelm the LLM by providing large context. Fig. 2 shows an example from the Pylint project[2], in which the bug report does not mention specific files, classes or methods that can be used by iterative-reasoning methods to guide the search process. In this case, repository-wide exploration methods expose the LLM to the directory structure of 778 files, while only a single file is relevant to the reported bug. This severe imbalance between relevant and irrelevant context dilutes useful signals and degrades reasoning quality. As repository size increases, the amount of extraneous context grows accordingly, further exacerbating this challenge.

Bug ID: 7114
Summary: Linting fails if module contains module of the same name
Description: Current behavior: Running `pylint a` if `a/a.py` is present fails while searching for an `__init__.py` file.
Expected behavior: Running `pylint a` if `a/a.py` is present should succeed.

Fig. 2. Bug Report from Pylint Project.

These limitations highlight the need for an IRBL technique that can (1) handle semantically rich but lexically weak bug reports, (2) reason over a small, relevant portion of a large code base without flooding the LLM's context, and (3) adaptively expand the search space by acquiring new evidence. GenLoc is designed to meet these requirements by integrating semantic retrieval with LLM-driven iterative code exploration, enabling adaptive evidence acquisition and search-space refinement during file-level localization.

2.6 Terminologies

This section introduces the key technical terms used in GenLoc.

Embedding and Semantic Retrieval: Embedding refers to representing complex data, e.g., text, images or source code, as numerical vectors in a high-dimensional space. The core idea is that items with similar meaning (e.g., two similar sentences or two code snippets implementing similar logic)

[1] https://github.com/apache/rocketmq/issues/7026
[2] https://github.com/pylint-dev/pylint/issues/4444

will have embeddings that are close to each other in that space. This property enables semantic retrieval, which retrieves results based on conceptual similarity rather than relying only on shared keywords or vocabulary overlap. In GenLoc, both source files and bug reports are converted into embeddings, allowing their similarity to be measured at a semantic level.

Vector Database: A vector database is a special type of database designed to store and manage large collections of high-dimensional embeddings and perform efficient similarity searches [18]. To accelerate the retrieval process, vector databases commonly use Approximate Nearest Neighbor (ANN) algorithms [76], which find vectors that are close to a given query without exhaustively comparing all entries. In GenLoc, the embeddings of all the source files are stored in a vector database to facilitate semantic retrieval.

Function Calling: This mechanism allows LLMs to interact with external tools and systems to retrieve up-to-date information or perform specific tasks that they cannot directly do. For example, if LLM is given the prompt "*What is the current temperature in Los Angeles?*", it will fail to answer because it lacks access to real-time data. However, it can be provided with access to a function like `get_current_temperature()`, which queries an external weather API and returns the latest temperature. The model can then call this function, receive the temperature and integrate it into its response. In GenLoc, we design a set of domain-specific code exploration functions (e.g., retrieving method signatures and method bodies) that enable the LLM to iteratively navigate the code base and identify potentially buggy files.

ReAct Framework: The ReAct (Reasoning and Acting) framework enables language models to interleave natural language reasoning with tool-based actions, thereby enhancing their problem-solving capabilities [93]. Instead of generating a final answer in a single step, the model alternates between articulating intermediate reasoning steps and performing actions, such as calling a function. This synergy between reasoning and acting allows the model to iteratively refine its understanding, validate hypotheses, and make informed decisions. Since ReAct has shown promise in handling complex tasks that require multi-step thinking, access to external environments, or dynamic decision-making [39, 87, 93], this paper adopts the ReAct framework to guide the LLM in navigating the code base and progressively narrowing down the search space during bug localization.

3 Methodology

GenLoc operates in two primary steps to localize relevant files based on a given bug report. First, it retrieves a set of semantically similar files using embedding-based similarity. Next, it employs an LLM augmented with a set of custom-designed code exploration functions, which allow the model to iteratively reason over the bug report and interact with the code base. During this stage, the model may examine the embedding-based retrieved files or explore other parts of the code base to generate a ranked list of potentially buggy files. An overview of the workflow is shown in Fig. 3.

3.1 Shortlisting suspicious files

To identify source files potentially related to a given bug report, this step computes the semantic similarity between the bug report and the source code using embedding-based representations. The process begins by traversing all source files in the project. For each file, its fully qualified file path and method bodies are extracted. Each file is then represented as a concatenation of its fully qualified file path and the method bodies. The inclusion of the file path provides contextual information about the file's role or feature within the overall system architecture (e.g., `org.eclipse.jdt.ui/ui/org/eclipse/jdt/ui/JavaElementLabels.java` suggests a UI-related component), while method bodies capture the actual implementation logic, which is crucial since methods serve as the fundamental units of behavior in a program.

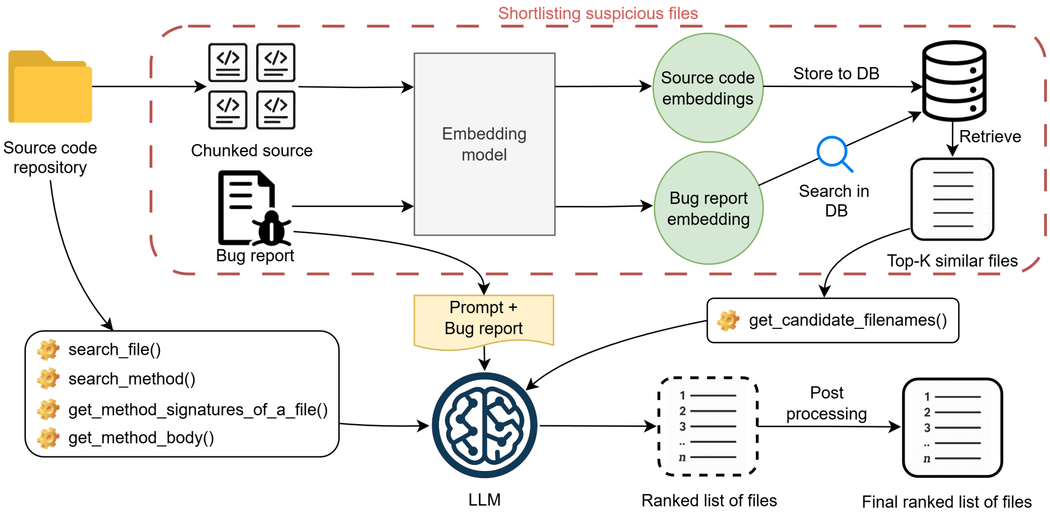

Fig. 3. Workflow of GenLoc.

The concatenation of a file's path and method bodies can result in lengthy text. To better localize relevant content, this representation is split into smaller chunks if it exceeds a predefined length threshold. Each chunk is then converted into an embedding vector and stored in a vector database. Similarly, the textual content of the bug report, comprising its summary and description, is transformed into an embedding vector. The bug report embedding is then used to query the vector database, retrieving the most semantically similar file chunks. Based on these retrieval results, a predefined number of top-ranked files are shortlisted as suspicious. These files serve as candidate inputs for further analysis by the LLM in the subsequent step.

3.2 Ranking suspicious files

In this step, an LLM is employed to generate a ranked list of potentially buggy source files based on a given bug report, as illustrated in Fig. 3. To support the LLM's code exploration process, a prompt (shown in Fig. 4) is constructed by following the ReAct (Reasoning and Acting) framework [93], which enables LLMs to interleave natural language reasoning with tool-based actions. This approach has proven effective in improving performance on complex decision-making tasks by enabling iterative reasoning and environment interaction.

At first, the LLM is prompted to assume the role of an expert software engineer specializing in bug localization, as role-playing improves the reasoning abilities of LLMs [32]. Next, the prompt is designed to decompose bug localization into a series of smaller tasks since LLMs perform better when complex tasks are broken down into sub-tasks [27]. These sub-tasks include analyzing the bug report, identifying potential file or method names, analyzing method signatures and bodies and integrating evidence to prioritize suspicious files.

To support the analysis process, GenLoc provides the LLM with access to the following five external functions. The LLM may invoke one or more of these functions over a fixed number of iterations to get information about the code base. Collectively, these functions allow the LLM to discover files, understand their role and analyze the internal logic of specific code segments, thereby supporting both breadth and depth in the bug localization process.

(1) `search_file()`: This function enables the LLM to check whether a specific file exists in the code base. It can be used to (i) verify the existence of a file explicitly mentioned in the bug report (e.g., checking if `JavaElementLabels.java` exists when "JavaElementLabels" is referenced), (ii) infer

You are an expert software engineer specializing in bug localization. Your goal is to identify the most probable buggy Java files based on a given bug report. You have access to five functions that help you search for file names, locate methods and analyze source code. You must follow an iterative, reasoning-based approach, refining your strategy dynamically based on insights gathered during each step. At each iteration, you should:

- Maintain a working shortlist of files that appear potentially buggy.
- Update the shortlist based on new evidence (e.g., method matches, code analysis).
- Avoid redundant operations–do not recheck the same filenames, methods or file contents multiple times.

Continue this process until you either: (a) Produce a well-justified ranked list of the 10 most relevant files, or (b) Reach the maximum limit of 10 iterations. In the 10th iteration, you must output your final ranked list.
Workflow
1. Analyze the Bug Report:

- Extract relevant keywords, error messages and functional hints from the bug summary and description.
- Identify potentially affected components (e.g., UI, database, networking).

2. File Discovery:

- Use *search_file()* to check if filenames derived from the bug report's keywords or functionality exist in the code base.
- If the bug report references a specific method name, use *search_method()* to find all files defining it.
- If an inferred filename or method location does not exist, refine your strategy: adjust assumptions, explore variations and retry.
- If strong matches are not found, use *get_candidate_filenames()* to retrieve 50 potentially relevant filepaths.
- Add promising files to your shortlist by prioritizing those align with terminology, functionality or methods discussed in the bug report.

3. Method Analysis:

- For each file in the shortlist that hasn't been analyzed yet:
  - Use *get_method_signatures_of_a_file()* to list its methods.
  - Identify methods that directly align with the bug's context (e.g., related functionality, naming hints).
  - For any method of interest, retrieve its implementation using *get_method_body()*.
  - Analyze the logic of any identified method(s) of interest to determine whether they align with the bug's symptoms.
- If this analysis reveals new relevant class names, filenames, or methods, use the appropriate search functions.
- Continuously update the shortlist by promoting, demoting or removing files based on evolving understanding.

4. Shortlist Refinement & Ranking:

- Rank files based on:
  - Semantic alignment with the bug report's keywords and described functionality
  - Method or filename alignment with bug context
  - Code logic alignment with the bug description
- If needed, iterate with refined assumptions or explore previously overlooked filenames or methods.

5. Final Output:

- Provide a ranked list of the 10 most relevant filepaths based on their likelihood of containing the bug.
- Ensure filepaths exactly match those provided–do not modify case, structure or abbreviate them.
- Justify each file's inclusion by referencing keywords, method matches or code logic that supports its relevance to the bug report.

Fig. 4. LLM Prompt.

and validate a possible filename derived from textual clues in the bug report, or (iii) identify candidate files after inspecting a method body.

(2) `search_method()`: When a method name is mentioned in the bug report, e.g., `updateLabel()`, this function enables the LLM to find which files contain its definition and thereby narrowing down the search space. In addition, it can be employed to discover candidate files based on references observed during method body inspection.
(3) `get_candidate_filenames()`: This function retrieves a list of potentially relevant files based on their semantic similarity to the bug report from the previous step (Section 3.1). The LLM can leverage this list as an initial pool of candidates, especially when the bug report does not provide any clues.
(4) `get_method_signatures_of_a_file()`: Once a file is identified, this function enables the model to inspect the file's high-level structure by retrieving all method signatures defined within it. This helps the LLM determine whether the file's responsibilities align with the bug report.
(5) `get_method_body()`: To evaluate a method's implementation, the model can call this function to retrieve the body of a given method. This allows a deeper inspection of whether the method may be related to the described issue.

At each iteration, the model can "reason" about what information is currently available, determine what additional data is required, and then "act" by invoking one of these functions. The outcome of the function call is integrated into the LLM's internal reasoning in the subsequent iteration. Since function calls issued by the LLM may contain errors, such as supplying incorrect parameters like non-existent filenames or method signatures, GenLoc incorporates an iteration-time recovery step. When no exact match is found, GenLoc either provides tentative options (e.g., all file paths sharing the same base filename or the closest method signatures using Damerau–Levenshtein distance [102]) or returns explicit feedback indicating that the requested element does not exist in the code base. This mechanism guides the LLM in handling mistakes (e.g, typos or incomplete names) by recovering the intended file or method, or preventing it from producing non-existent ones.

Lastly, the LLM is prompted to output a ranked list of top 10 files, where each entry includes a fully qualified file path along with a brief justification for its selection. This design choice is informed by empirical findings indicating that inspecting more than ten files exceeds the acceptability threshold for nearly 98% of practitioners [31]. Since multiple files from different packages can have the same name, the fully qualified file paths are used to uniquely identify each file. The inclusion of justifications is motivated by prior work demonstrating that self-explanatory prompts significantly improve LLM comprehension [86].

Although the LLM receives feedback during intermediate iterations, it can still produce erroneous or non-existent file paths in the final ranked list. A common issue observed in our preliminary analysis is the omission of intermediate packages. For example, the model may omit the "internal" package and produce `org.eclipse.jdt.ui/ui/org/eclipse/jdt/ui/text/java/AlphabeticSorter.java` instead of `org.eclipse.jdt.ui/ui/org/eclipse/jdt/internal/ui/text/java/AlphabeticSorter.java`. To address this, GenLoc applies a post-processing recovery step to verify the correctness of each predicted file. If the predicted fully qualified file path matches an existing file, it is retained. Otherwise, the base filename (e.g., `AlphabeticSorter.java`) is extracted, and all files with the same name are retrieved. Among these candidates, the file whose fully qualified file path has the highest Jaccard similarity [57] with the predicted filename is selected. Since Jaccard similarity measures token overlap, it favors the correct file even when an intermediate package is missing. If the base filename does not exist in the code base, no suitable match can be found and the file is excluded from the final ranked list.

4 Experiement Setup

This section presents the research questions, baseline techniques, benchmark dataset, evaluation metrics and implementation details of GenLoc.

4.1 Research Questions

To evaluate GenLoc, the following research questions are investigated:

RQ1: How does GenLoc perform compared to traditional and deep learning-based IRBL techniques?
RQ2: How does GenLoc perform compared to other LLM-based IRBL techniques?
RQ3: How does GenLoc perform compared to the file-level localization stages of LLM-based issue localization techniques?
RQ4: How do different components of GenLoc contribute to the overall performance?
RQ5: How well does GenLoc perform on previously unseen bugs?
RQ6: How does the choice of retrieval model and LLM impact the performance of GenLoc?

4.2 Baseline Techniques

To comprehensively evaluate GenLoc, we compare it against publicly available and executable baselines from three categories.

Traditional and Deep Learning-Based IRBL Techniques (RQ1). We select four widely studied IRBL techniques from the traditional and deep learning categories: BugLocator [104], BLUiR [69], BRTracer [83] and DreamLoc [60]. Although BugLocator, BLUiR and BRTracer were introduced earlier, they remain actively used in recent empirical studies to characterize the strengths and limitations of IR-based bug localization techniques [30, 40]. BugLocator uses revised VSM (rVSM) by incorporating file length normalization and historical bug-fix frequency to prioritize more fault-prone files [104]. BLUiR assigns different weights to structural program elements such as class and method names, and computes their similarity to the bug report content separately [69]. BRTracer divides source files into smaller segments and integrates additional signals such as stack traces and historical fix data to rank files [83]. DreamLoc leverages a Wide and Deep architecture: the Wide component captures software-specific statistical features (e.g., fix frequency and recency), while the Deep component applies a relevance model for fine-grained similarity scoring [60].

For DreamLoc, the original implementation is used. For BugLocator, BLUiR and BRTracer, the implementations provided in [36] are adopted, since the original implementations are unavailable. While we aimed to incorporate several other IRBL techniques in the evaluation, they could not be included due to implementation-related issues. For instance, DNNLOC [35] and DeepLoc [90] do not have a publicly released implementation. Although RLocator [10] provides a replication package, it is incomplete due to missing source code. AdaptiveBL [16] and its successor FLIM [43] could not be executed due to unresolved dependency issues.

LLM-Based IRBL Techniques (RQ2). Following the taxonomy in Table 1, we compare GenLoc against two state-of-the-art LLM-based IRBL approaches: LLM-BL [42] and BRaIn [70]. LLM-BL expands the bug report using an LLM, retrieves candidate files through hybrid (lexical and semantic) search and presents the shortlisted files along with their relevant code lines to the LLM for reranking [42]. BRaIn retrieves top-K files using lexical search, identifies relevant methods from those files using an LLM and refines the results through keyword-based query expansion and reranking [70]. For both techniques, we use their original implementations. LLM-BL supports GPT-based models; therefore, we replace GPT-3.5-Turbo-0125 with GPT-4o-mini, following current OpenAI recommendations. This aligns with GenLoc's default configuration (Section 4.5.2), ensuring a fair comparison under identical model conditions. For BRaIn, we retain the default model from the original implementation, Mistral-7B-Instruct-v0.3, which is also reported as the best-performing model in the original study. We keep this configuration to ensure consistency with the original setup, as adapting BRaIn to proprietary GPT-based models would require modifications to its inference pipeline.

File-Level Localization Stages of Issue Localization Techniques (RQ3). As summarized in Table 1, issue localization techniques aim to identify fine-grained change locations but generally begin with a file-level localization phase to narrow down the search space before subsequent analysis. Accordingly, we compare GenLoc against the file-level localization stages of two state-of-the-art LLM-based issue localization approaches: CoSIL [22] and LocAgent [13]. These approaches represent repository-wide exploration and iterative-reasoning methods, respectively. Prior studies show that CoSIL outperforms other issue localization techniques such as OrcaLoca as well as RepoSearcher at the file level [22]. We exclude BugCerberus due to the unavailability of its implementation and because it adopts the AgentLess paradigm [88], which is outperformed by CoSIL [22].

CoSIL performs file-level localization by prompting the LLM to rank suspicious files using the repository's directory tree and then uses a Python import-based call graph to refine the localization results [22]. LocAgent extracts keywords from the issue description and uses them to guide dependency-graph exploration for progressively identifying relevant files [13]. We use the original implementations of both approaches and evaluate them with the same LLM as GenLoc to ensure a fair comparison under identical model conditions, while restricting the evaluation strictly to their file-level localization outputs.

4.3 Benchmark Dataset

We use three diverse benchmark datasets to evaluate GenLoc: the Ye et al. dataset [95], the GHRB (GitHub Recent Bugs) dataset [37] and SWE-bench Lite [4]. The Ye et al. dataset is the most widely adopted benchmark in prior IRBL studies [16, 60, 96]. Additionally, its large scale, over 22,000 bugs from six large, open-source Java projects (Table 2), makes it well suited for training deep learning–based approaches (e.g., DreamLoc) and comparing with GenLoc [94, 95]. The GHRB (GitHub Recent Bugs) dataset contains 131 recent bugs from 16 Java projects (e.g., Apache Dubbo, OpenAPI Generator), as shown in Table 3. This dataset is specifically designed for evaluating LLM-based IRBL approaches [37]. SWE-bench Lite is a curated subset of SWE-bench [23] that focuses mainly on bug-fixing issues [4]. It contains 300 issues from 12 popular Python projects, including Django, Matplotlib (Table 4) and has been widely adopted by recent issue localization approaches, making it suitable for evaluating file-level localization in issue localization pipelines [22, 88].

Table 2. Ye et al. Dataset Description

Project	# of Bug Reports	# of Java Files			LOC		
		Max	Median	Min	Max	Median	Min
AspectJ	593	6,879	4,439	2,076	699,250	515,153	216,387
Birt	4,178	9,697	6,841	1,700	2,322,074	1,549,525	412,795
Eclipse	6,495	6,243	3,454	382	1,156,041	637,158	47,348
JDT	6,274	10,544	8,184	2,294	922,812	609,378	133,323
SWT	4,151	2,795	2,056	1,037	934,357	639,216	301,698
Tomcat	1,056	2,042	1,552	924	487,701	413,472	270,046

Table 3. GHRB Dataset Description

# of Projects	# of Bug Reports	# of Java Files			LOC		
		Max	Median	Min	Max	Median	Min
16	131	12,025	1,977	93	1,874,139	299,697	14,759

Table 4. SWE-bench Lite Dataset Description

# of Projects	# of Bug Reports	# of Python Files			LOC		
		Max	Median	Min	Max	Median	Min
12	300	2,757	1,377	75	771,389	361,222	16,551

4.4 Evaluation Metrics

Similar to other work [65, 104], the following three metrics are used for evaluation:

1. Accuracy@k: It denotes the number of bugs for which at least one of the actual buggy files is ranked within the top-k positions (k = 1, 5, 10) of the ranked list. A higher accuracy@k value indicates a better performance of the bug localization technique. Similar to other studies [43, 60], the value of k is set to 1, 5 and 10.

2. Mean Reciprocal Rank@k (MRR@k): This metric measures the average reciprocal rank of the first correctly identified buggy file within the top-k results across all bug reports [85]. For each bug report, the reciprocal rank is defined as the inverse rank of the first correct buggy file that appears among the top-k results. MRR@k is calculated as follows:

$$MRR@k = \frac{1}{N} \sum_{i=1}^{N} \begin{cases} \frac{1}{\text{rank}_i} & \text{if rank}_i \leq k \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where, N is the total number of bug reports. rank_i denotes the rank of the first correct buggy file for the i-th bug report among the top-k results. The higher the value of MRR@k, the better the technique. Following prior studies [29, 65], the value of k is set to 10. An MRR@10 value of 0.2 indicates that, on average, the first relevant file is ranked around position 5 within the top 10 results.

3. Mean Average Precision@k (MAP@k): This metric evaluates how well a bug localization technique ranks all actual buggy files within the top-k retrieved results for a set of bug reports [19]. For each bug report, it calculates the Average Precision (AP) using the following formula, which considers both the presence and the ranking positions of all relevant buggy files:

$$AP_i = \frac{\sum_{j=1}^{M} P(j) \times \text{Rel}(j)}{\text{Number of relevant source files}} \quad (2)$$

where, M is the number of retrieved files in the top-k ranked list for the i-th bug report (i.e., M=k). $Rel(j)$ denotes whether the file at position j is relevant or not. $P(j)$ indicates the precision at rank j and computed as follows:

$$P(j) = \frac{\text{Number of relevant files in top-}j\text{ positions}}{j} \quad (3)$$

The Mean Average Precision is then computed by averaging the AP scores across all bug reports, as shown here:

$$MAP = \frac{\sum_{i=1}^{N} AP_i}{N} \quad (4)$$

where, N denotes the total number of bug reports. A higher MAP@k value indicates better ranking quality of the buggy files. Following prior studies [29, 65], the value of k is set to 10. For example, a MAP@10 value of 0.3 implies that, on average, 30% of the actual buggy files are retrieved with high precision within the top 10 results.

4.5 Implementation and Default Configurations

GenLoc currently supports Java and Python and can be easily extended to other programming languages, as it leverages the Tree-sitter library [6] for source code parsing, which supports over ten languages.

4.5.1 Configuration of Semantic Retrieval. To compute semantic similarity between bug reports and source code files, GenLoc employs OpenAI's `text-embedding-3-small` as its default model due to its cost-effectiveness ($0.02 per 1M tokens) [5]. The similarity between bug report embeddings and source code embeddings is computed using cosine similarity, as followed in prior work [21].

Following Amazon Bedrock's recommendation [1], we use a chunk size of 300 tokens to balance between granularity and context (i.e., capturing relevant code segments while preserving surrounding information for meaningful semantic comparison). To assess the robustness of this choice, we conduct a sensitivity analysis by additionally evaluating smaller (100 tokens) and larger (500 tokens) chunk sizes. Each setting is evaluated on the Ye et al. dataset using retrieval coverage, defined as the percentage of bugs for which at least one ground-truth fixed file appears among

the top-50 retrieved candidates. As shown in Table 5, the 300-token setting achieves the highest coverage, validating its selection as GenLoc's default.

Table 5. Coverage Achieved using Different Chunk Sizes.

Chunk Size	**100**	**300**	**500**
Coverage (%)	80.94	**81.28**	79.76

After embedding all chunks, GenLoc retrieves the top-ranked candidate files for each bug report. Using the Ye et al. dataset, we examined how retrieval coverage changes as the number of returned candidates increases from 1 to 100. As illustrated in Fig. 5, we observe steep gains in the early stages (e.g., top-1 to top-20), followed by diminishing returns beyond the top-50. Increasing the retrieval window from 50 to 100 candidates (i.e., doubling the number of retrieved files) yields only a 4.26% increase in retrieval coverage, while substantially increasing noise and computational overhead. Therefore, retrieving the top 50 candidates is adopted as GenLoc's default setting.

To minimize embedding cost, GenLoc stores all chunk embeddings in ChromaDB [2] when processing the first bug from each project. For subsequent bugs, embeddings are updated incrementally by re-embedding only files that are added, modified, deleted or renamed across versions

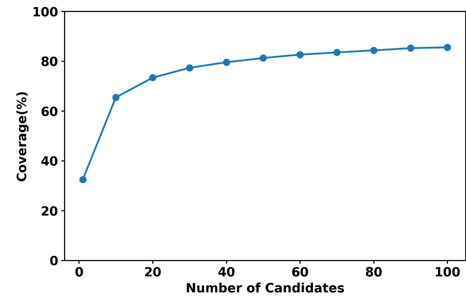

Fig. 5. Retrieval Coverage for varying Number of Candidate Files.

4.5.2 Configuration of LLM. GenLoc uses OpenAI's `GPT-4o mini` as its default model for reasoning due to its cost-effectiveness ($0.15 per 1M input tokens and $0.60 per 1M output tokens) [3]. All model parameters are kept at their default settings to avoid the computational cost of hyperparameter tuning. In particular, the temperature parameter is left at its default value (1.0) (rather than being set to 0) to enable diverse reasoning paths during inference [25, 105]. A temperature of 0 significantly reduces variability by consistently selecting the highest-probability tokens, which can lead the model to overlook less obvious but relevant files. Our preliminary analysis shows that the performance remained mostly unchanged regardless of the temperature setting. This aligns with prior findings that changes in temperature between 0.0 and 1.0 do not have a significant effect on LLM's performance for problem-solving tasks [67]. To account for LLM non-determinism, the experiments were executed three times and the average results were reported, as followed in [101].

It is important to note that the choices of embedding model and LLM described above correspond to GenLoc's default configuration. In RQ6, we further evaluate alternative embedding models and LLMs to assess the robustness of GenLoc under different model choices.

5 Result Analysis

In this section, the experimental results in accordance with the research questions are discussed.

5.1 RQ1: Comparison with Traditional and Deep Learning-based IRBL Techniques

Table 6 reports the results of evaluating GenLoc against four traditional and deep learning-based IRBL techniques (DreamLoc, BRTracer, BLUiR and BugLocator). We use the Ye et al. dataset due to its scale and popularity [10, 16, 52]. Following prior work [10, 17], 60% data is used as historical (training) data and the most recent 40%, consisting of 9,097 bugs, is used for evaluation.

Results show that GenLoc demonstrates strong overall performance across all evaluation metrics compared to prior bug localization techniques. On average, GenLoc achieves the highest Accuracy@1 (44.01%), substantially outperforming the second-best technique (BRTracer with 26.84%) by a relative margin of over 63%. This indicates that GenLoc can correctly identify the buggy file at the top position in a significant portion of bug reports. Similarly, GenLoc leads in Accuracy@5 (63.27%) and Accuracy@10 (68.28%), further affirming its ability to prioritize buggy files effectively within the top candidates. In terms of ranking quality, GenLoc achieves the highest scores in both MAP@10 (0.41) and MRR@10 (0.52), outperforming the next-best MAP@10 (DreamLoc with 0.30) and MRR@10 (BRTracer and DreamLoc with 0.37) by 36.67% and 40.54%, respectively, confirming its superiority in ranking relevant files near the top of the result list.

Table 6. Comparison of GenLoc and Non-LLM-based IRBL Techniques

Technique	**Accuracy@k (%)**			**MAP@10**	**MRR@10**
	k=1	k=5	k=10		
GenLoc	**44.01**	**63.27**	**68.28**	**0.41**	**0.52**
DreamLoc	26.33	52.32	63.07	0.30	0.37
BRTracer	26.84	51.21	60.42	0.29	0.37
BLUiR	26.49	49.04	59.82	0.28	0.36
BugLocator	24.19	46.20	56.65	0.26	0.33

It is worth mentioning that GenLoc uses only the bug report as input, without incorporating any additional information such as historical bug-fix data or file change frequency. In contrast, techniques like DreamLoc, BRTracer and BugLocator leverage such additional data to improve localization accuracy. Despite considering less information, GenLoc consistently outperforms these baselines on all metrics.

Apart from Accuracy@k, MRR@10 and MAP@10, the number of unique bugs localized by each approach is examined. In this analysis, a bug is considered successfully localized if the correct buggy file appears within the top 10 ranked results (i.e., Accuracy@10) [31]. Figure 6 presents the unique bugs localized by each technique under this criterion. To account for GenLoc's variability across trials and enable a fair comparison with existing deterministic baselines, we consider the union of all bugs localized across three trials. This union captures the set of unique bugs that GenLoc can localize in at least one run, reflecting the overall capability of the approach rather than the outcome of a single arbitrary execution.

Fig. 6. Overlap Analysis between GenLoc and Non-LLM-based IRBL techniques.

Among all approaches, GenLoc can localize the highest number of unique bugs (601 bugs). This indicates that GenLoc not only localizes more bugs but also captures a unique subset of bugs missed by all other methods, highlighting its potential to complement existing methods.

5.2 RQ2: Comparison with LLM-based IRBL Techniques

We compare GenLoc against two state-of-the-art LLM-based IRBL techniques, BRaIn and LLM-BL. We use both the GHRB dataset and a subset of the Ye et al. dataset consisting of the most recent 300 bug reports (50 per project) to enable assessment across both modern and traditional benchmarks.

Table 7. Comparison of GenLoc and LLM-based IRBL Techniques

Technique	Accuracy@k (%)			MAP@10	MRR@10
	k=1	k=5	k=10		
GenLoc	**63.36**	**76.85**	**79.65**	**0.59**	**0.69**
LLM-BL	50.38	63.11	67.18	0.47	0.56
BRaIn	22.90	51.15	61.07	0.28	0.34

Table 7 reports the comparison on the GHRB dataset. The results show that GenLoc consistently outperforms all baselines across every evaluation metric. It achieves the highest Accuracy@1 of 63.36%, substantially higher than other techniques and maintains this advantage at k=5 and k=10. For ranking-based measures, GenLoc achieves MAP@10 of 0.59 and MRR@10 of 0.69, representing over 23% improvement compared to LLM-BL and more than 100% improvement over BRaIn. In addition, GenLoc localizes the highest number of unique bugs (12), as shown in Fig. 7, indicating that it complements prior LLM-based IRBL techniques.

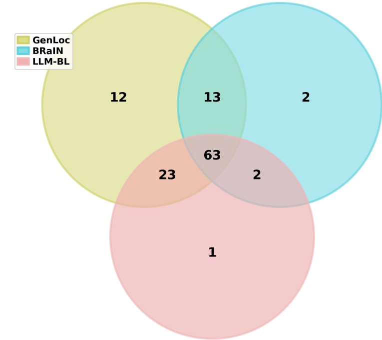

Fig. 7. Overlap Analysis between GenLoc and Other LLM-based IRBL Approaches.

Notably, GenLoc successfully localizes the bug shown in Fig. 1, whereas both BRaIn and LLM-BL fail because the buggy file is never included in their initial shortlisted candidates. In contrast, GenLoc's semantic retrieval captures the relevant file and its iterative analysis enables deeper inspection of the file's logic. We observe similar results on the Ye et al. subset, with detailed results provided in the replication package.

5.3 RQ3: Comparison with Issue Localization Techniques

We evaluate GenLoc against two state-of-the-art LLM-based issue localization techniques, CoSIL and LocAgent. We use the SWE-bench Lite dataset, which is a standard benchmark adopted by recent issue localization approaches [13, 22, 98].

Table 8 presents the results, showing that GenLoc achieves the strongest performance across all evaluation metrics. It attains the highest Accuracy@1 of 60.67%, representing a relative improvement of approximately 15% over CoSIL and more than 110% over LocAgent. In terms of ranking quality, GenLoc obtains MAP@10 and MRR@10 values of 0.70, yielding relative improvements of about 13% over CoSIL and over 110% compared to LocAgent for both metrics.

Table 8. Comparison of GenLoc and Issue Localization Techniques

Technique	Accuracy@k (%)			MAP@10	MRR@10
	k=1	k=5	k=10		
GenLoc	**60.67**	**80.22**	**84.00**	**0.70**	**0.70**
CoSIL	52.67	74.44	78.89	0.62	0.62
LocAgent	28.33	40.11	41.00	0.33	0.33

These results indicate that GenLoc is more effective at identifying the correct file at early ranks while maintaining a high-quality overall ranking. Furthermore, GenLoc localizes 9 unique bugs (Fig. 8) including the bug in Fig. 2. The bug describes a linting failure due to incorrect handling of directories that contain same-named module files. In this case, CoSIL is distracted by excessive repository-level context, while LocAgent is misled by lexical cues toward naming and import checkers. In contrast, GenLoc first applies semantic retrieval to narrow the search space, where

`expand_modules.py` aligned with the bug context. It then analyzes the method signatures of this file to confirm its relevance and inspects method bodies to uncover the failure-inducing logic.

5.4 RQ4: Ablation Study

We conduct an ablation study to isolate the impact of candidate retrieval, LLM-driven code exploration with tool support and post-processing on GenLoc's overall bug localization performance. We focus on the GHRB dataset to enable systematic evaluation of multiple GenLoc variants without prohibitive computational cost.

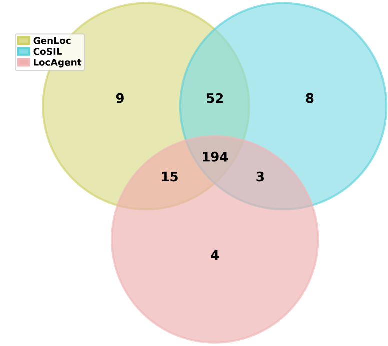

Fig. 8. Overlap Analysis between GenLoc and Other Issue Localization Approaches.

5.4.1 Comparison of Syntactic and Semantic Retrieval for Shortlisting Suspicious Files. GenLoc shortlists 50 candidate files through an initial retrieval step. To assess the effectiveness of this step independently of downstream reasoning, we compare semantic (embedding-based) retrieval with widely used BM25-based syntactic retrieval (adopted from [70]) based on their ability to include the ground-truth buggy file within the top-K candidates. Table 9 shows that semantic retrieval achieves a higher Accuracy@50 (83.21%) than syntactic retrieval (71.76%), indicating superior effectiveness in capturing bug-report intent when lexical overlap is limited. This behavior is illustrated in Fig. 1, where syntactic retrieval fails due to limited lexical overlap between the bug report and the source code, while semantic retrieval succeeded. Thus, GenLoc adopts semantic retrieval for candidate shortlisting.

Table 9. Comparison of Syntactic and Semantic Retrieval

Type	Accuracy@k (%)				MAP@10	MRR@10
	k=1	k=5	k=10	k=50		
Semantic Retrieval	**20.36**	**58.27**	**72.52**	**83.21**	**0.30**	**0.37**
Syntactic Retrieval	15.27	40.46	51.15	71.76	0.20	0.26

5.4.2 Impact of LLM-based Code Exploration. To quantify the contribution of LLM-based code exploration, we compare GenLoc against several ablated variants in which key components of the pipeline are selectively disabled:

(1) **GenLoc:** This is the complete version of the proposed approach, where the LLM is provided with five functions, including `get_candidate_filenames()` to utilize embedding-based retrieval to identify potentially buggy files.
(2) **Embedding-Only/Semantic Retrieval:** A baseline that evaluates the standalone performance of the embedding-based retrieval system, where source files are ranked solely based on their semantic similarity to the bug report, without any LLM reasoning or iterative refinement.
(3) **GenLoc-NoEmbed:** A variant where the LLM operates without access to the embedding-based recommended files, i.e., the `get_candidate_filenames()` function is removed. The model must infer filenames directly from the bug report and continue reasoning with the remaining four functions.
(4) **GenLoc-Naive:** A minimal setup, where the LLM is limited to using only two functions, `search_file()` and `search_method()`, and must rely solely on basic keyword-based exploration without access to embedding guidance or deeper code-level analysis.

Table 10 presents the average results on the GHRB dataset across all four configurations. Results show that GenLoc achieves the best performance on all metrics, demonstrating the strength of combining semantic retrieval with iterative LLM analysis. The Embedding-Only variant obtains

Table 10. Impact of Reasoning and Tooling

Technique	Accuracy@k (%)			MAP@10	MRR@10
	k=1	k=5	k=10		
GenLoc	**63.36**	**76.85**	**79.65**	**0.59**	**0.69**
Embedding-Only	20.36	58.27	72.52	0.30	0.37
GenLoc-NoEmbed	46.05	56.24	57.76	0.42	0.50
GenLoc-Naive	46.31	57.25	58.78	0.43	0.50

only 20.36% at Accuracy@1, indicating that retrieval alone is insufficient without deeper analysis. Similarly, the GenLoc-NoEmbed achieves only 46.05% Accuracy@1, showing that the absence of semantic retrieval limits the LLM's ability to identify relevant files. These results highlight that semantic retrieval and LLM-driven code analysis are complementary, and removing either degrades localization performance.

Bug ID: 20109
Summary: [BUG] [typescript-fetch] ModelNamePrefix is added twice to import statements thereby breaking the build
Description: When generating the model files the prefix is properly added once in front of the generated files. However, in the import statements the prefix is added _twice_ resulting in erroneous statements breaking the build. When e.g. 'SomePrefix' is set for 'modelNamePrefix', then the following model files are generated for 'example-for-file-naming-option.yaml': * 'SomePrefixPetCategory.ts' * 'SomePrefixPet.ts' However, _within_ 'SomePrefixPet.ts' it looks like this: "'typescript import type SomePrefixPetCategory from './SomePrefixSomePrefixPetCategory';"' I.e. 'SomePrefix' is added twice.

Fig. 9. Bug Report from OpenAPI Generator Project.

The performance improvement of GenLoc over GenLoc-NoEmbed and GenLoc-Naive indicates that GenLoc goes beyond superficial cues (e.g., filenames or keywords) and performs deeper reasoning. A representative case[3] is shown in Fig. 9, where GenLoc-NoEmbed and GenLoc-Naive failed to localize the correct file, while GenLoc succeeded. The bug report described an error in the TypeScript Fetch generator of OpenAPI, where the modelNamePrefix option was being incorrectly applied twice in generated import statements. Misled by surface-level clues, GenLoc-NoEmbed and GenLoc-Naive attempted to locate a nonexistent file named `ModelNamePrefix` and also searched for the `generate()` method, which was unrelated to the bug and provided no useful signal. In contrast, GenLoc invoked `get_candidate_filenames()` to narrow down to the TypeScript-specific implementation and by analyzing method signatures and bodies in `TypeScriptFetchClientCodegen.java` (e.g., `toModelFilename()`, `parseImports()`), it successfully localized the buggy file.

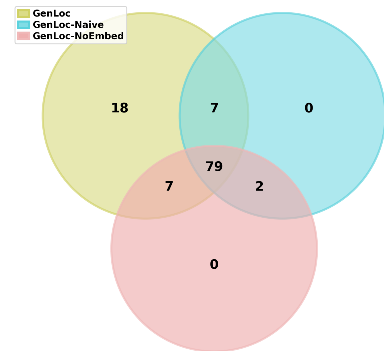

Fig. 10. Overlap Analysis between GenLoc and Its Ablated Variants.

Although GenLoc-NoEmbed and GenLoc-Naive obtain similar results, the overlap analysis (Fig. 10) shows that they capture distinct subsets of bugs. This demonstrates the importance of combining all four functions, `search_file()`, `search_method()`, `get_method_signatures_of_a_file()` and `get_method_body()`, to achieve broader coverage. Their integration enables GenLoc to leverage both keyword-level and structure-aware reasoning.

Finally, Table 11 presents the consistency of results, measured as consensus across trials for each variant (i.e., GenLoc, GenLoc-NoEmbed and GenLoc-Naive). Here, consensus indicates the proportion of bugs that are localized in at least two or all three trials, reflecting the stability of each

[3]https://github.com/OpenAPITools/openapi-generator/issues/19039

approach. GenLoc shows the highest stability, with 93.69% of bugs localized in at least two runs and 88.29% in all three runs. In contrast, GenLoc-NoEmbed and GenLoc-Naive attain lower values, reflecting greater variability across trials. It shows that the integration of all components not only improves accuracy but also enhances the consistency of GenLoc's results.

Table 11. Consistency Across Trials

Consensus	GenLoc (%)	GenLoc-NoEmbed (%)	GenLoc-Naive (%)
$\geq$ 2 trials	**93.69**	85.23	86.36
$\geq$ 3 trials	**88.29**	72.73	76.14

5.4.3 Impact of Post-Processing. We evaluate the effect of post-processing by comparing GenLoc with a variant that disables this step (GenLoc-NoPost). Table 12 shows that removing post-processing leads to a substantial performance drop across all metrics, primarily due to LLM's error in reconstructing fully qualified filenames (e.g., omitting intermediate package segments). Post-processing corrects these errors and refines the final ranking, leading to more accurate localization results.

Table 12. Impact of Post-Processing

Technique	Accuracy@k (%)			MAP@10	MRR@10
	k=1	k=5	k=10		
GenLoc	**63.36**	**76.85**	**79.65**	**0.59**	**0.69**
GenLoc-NoPost	44.27	49.87	51.66	0.47	0.52

5.5 RQ5: Performance on Unseen Bugs

Table 13 presents the performance of GenLoc on both seen and unseen bug reports from the GHRB dataset. We define unseen bugs as the subset of GHRB bug reports submitted after the knowledge cut-off date of GPT-4o-mini (October 1, 2023), resulting in 49 unseen bug reports. The results indicate that GenLoc maintains consistent performance across the two subsets. For instance, the Accuracy@1 remains stable at approximately 63% for both seen and unseen bugs. To further validate this observation, we performed a Mann–Whitney U test, a non-parametric method suitable for comparing independent groups [53]. The results reveal no statistically significant difference in Reciprocal Rank (RR) (p-value > 0.7) and Average Precision (AP) (p-value > 0.3) between the two groups. These findings suggest that GenLoc does not rely on memorization or training data leakage and can effectively localize bugs even in reports that fall outside the temporal scope of the LLM's training data.

Table 13. Performance on Seen and Unseen Bugs

Category	Accuracy@k (%)			MAP@10	MRR@10
	k=1	k=5	k=10		
Seen	63.41	78.05	80.90	0.60	0.70
Unseen	63.27	74.83	77.55	0.56	0.68

5.6 RQ6: Choice of Models

We investigate the impact of both embedding models and LLM on GenLoc's performance. For embeddings, we consider OpenAI's `text-embedding-3-small` and `CodeXEmbed` (400M). Text-embedding-small is selected as a cost-effective and widely used general-purpose embedding model, while CodeXEmbed is included because it is designed to unify code and text retrieval within a single framework and has demonstrated strong performance on code retrieval task [45]. As for LLMs, we compare `GPT-4o-mini` with `GPT-5.1`. GPT-4o-mini represents a lightweight and cost-effective model, whereas GPT-5.1 is one of the latest and more advanced LLMs. Similar to RQ4, we focus on

the GHRB dataset to enable systematic evaluation of multiple GenLoc variants without prohibitive computational cost.

Table 14 reports the corresponding results. Both embedding models yield comparable performance across all metrics, indicating that GenLoc is relatively robust to the choice of embedding model as long as the embeddings effectively capture both code and natural language semantics. In contrast, the choice of LLM reveals a clear accuracy–cost trade-off. GPT-5.1 consistently achieves higher localization accuracy than GPT-4o-mini, but incurs a per-bug cost that is approximately 6 times higher (about \$0.035 vs. \$0.006 per bug). Despite this cost difference, GPT-4o-mini delivers competitive performance, making it suitable for large-scale or budget-constrained settings, whereas GPT-5.1 is preferable when maximizing localization accuracy is the primary objective.

Table 14. Performance of Different Models

Embedding	LLM	Accuracy@k (%)			MAP@10	MRR@10
		k=1	k=5	k=10		
Text-embedding-small	GPT-4o-mini	63.36	76.85	79.65	0.59	0.69
	GPT-5.1	**70.74**	**85.50**	**88.04**	**0.66**	**0.77**
CodeXEmbed	GPT-4o-mini	60.82	77.86	80.41	0.58	0.68
	GPT-5.1	**69.21**	**86.51**	**90.08**	**0.66**	**0.77**

6 Discussion

The results show that GenLoc consistently outperforms traditional IR-based methods, deep learning-based models, recent LLM-based IRBL techniques as well as state-of-the-art issue localization approaches across all benchmarks. Unlike prior IRBL approaches that rely on fixed candidate lists, or issue localization methods that depend on strong lexical cues or expose the LLM to large repository-level context, GenLoc enables incremental hypothesis refinement through selective, on-demand code inspection guided by semantic retrieval. The ablation study confirms that neither semantic retrieval nor LLM-driven analysis alone is sufficient; removing either component substantially degrades performance, particularly for semantically rich but lexically weak bug reports.

Apart from higher accuracy, GenLoc is also cost-effective. The average cost to localize a bug in Ye et al. dataset, GHRB dataset and SWE-bench Lite is \$0.010, \$0.025 and \$0.023 respectively. The lower per-bug cost on the Ye et al. projects, despite their larger size in terms of files and lines of code, arises from their multi-version structure: once the initial embedding is completed, only changed or newly added files need to be reprocessed in later versions. By contrast, GHRB and SWE-bench Lite include more projects but far fewer bug reports per project, so a larger fraction of projects must go through the full initial embedding stage without enough subsequent bugs to offset this overhead. Nevertheless, the per-bug cost remains reasonable, demonstrating that GenLoc is affordable and practical for both long-lived projects and newer projects with fewer versions. Moreover, unlike BugLocator, BRTracer and DreamLoc, GenLoc relies solely on the bug report, thereby making it broadly applicable even in projects without historical bug-fix data.

On average, GenLoc takes 47.7 seconds to produce results. This is consistent with findings from a prior practitioner study [31], which reported that 90% of developers expect a bug localization technique to return results within one minute. This suggests that GenLoc can operate within developer-expected timeframes while maintaining high accuracy and low cost, highlighting its suitability for real-world debugging scenarios.

Despite its success, GenLoc may not identify all files associated with a given bug within its top-10 ranked list, which is the current output size of the system. However, a prior work from Facebook [59] suggests that predicting even a single relevant file is useful in practice since it helps route the issue to the appropriate developer or team and can serve as a starting point for discovering

additional affected files through manual inspection or complementary techniques. Although this does not replace the need for comprehensive localization, it highlights the potential value of partial predictions in real-world debugging workflows.

7 Threats to Validity

This section presents potential ways in which the validity of the study may be compromised.

Internal Validity: One of the most significant threats to internal validity is the possibility of data leakage, such as bug-fixing commits being present in the LLM's training data. However, the result of RQ5 shows that GenLoc maintains consistent performance on both seen and unseen bug reports in the GHRB dataset, indicating that GenLoc effectively localizes bugs even for reports outside the temporal scope of the LLM's training data. Another potential threat arises from the inherent randomness of LLMs, which may yield different results across different runs. To mitigate this, all experiments were repeated three times, and a replication package is released to support reproducibility and enable further validation by the research community.

Construct Validity: Construct validity concerns whether the evaluation metrics accurately measure the effectiveness of bug localization techniques [82]. This study employs Accuracy@k, MRR@10 and MAP@10, which are widely used in prior IRBL research and considered standard for assessing ranking quality [65, 85].

External Validity: GenLoc is evaluated on three complementary benchmarks spanning two programming languages. The first is a large benchmark of over 9,000 bug reports from six large open-source Java projects that has been widely used in prior IRBL research [10]. The second is the GHRB dataset which contains 131 recent bugs from 16 Java projects and is specifically designed for evaluating LLM-based IRBL approaches [37]. The third is SWE-bench Lite, which consists of 300 issues from 12 popular Python projects [88]. While these benchmarks provide diverse evaluation settings, the findings may still not generalize to proprietary industrial systems or projects written in other programming languages. Additional studies are needed to evaluate the effectiveness of GenLoc in broader and more heterogeneous settings.

8 Conclusion and Future Work

This paper presents GenLoc, a novel IRBL technique that integrates semantic retrieval with LLM-driven iterative code analysis. We evaluate GenLoc on over 9,000 real-world bugs from six large-scale Java projects, the GHRB dataset and the SWE-Bench Lite dataset. Across all datasets, GenLoc consistently outperforms traditional IR-based methods, deep learning models, state-of-the-art LLM-based IRBL techniques and the file-level localization stages of recent issue localization approaches in terms of Accuracy@k, MRR@10 and MAP@10. Moreover Genloc can localize the largest number of unique bugs that other techniques fail to identify, demonstrating its effectiveness over existing IRBL techniques and issue localization pipelines.

In the future, GenLoc could be extended to support finer-grained bug localization at the method or statement level to provide more precise debugging assistance. Another promising direction is to integrate GenLoc with automated program repair pipelines to evaluate its effectiveness in guiding correct patch generation.

9 Data Availability

Our replication package[4] is publicly available to facilitate future research and reproducibility.

[4] https://github.com/seal-hub/Towards-Explorative-IRBL

Acknowledgments

This work has been supported, in part, by award numbers 2211790 and 2106306 from the National Science Foundation.

References

[1] [n. d.]. AWS Machine Learning Blog. https://aws.amazon.com/blogs/machine-learning/amazon-bedrock-knowledge-bases-now-supports-advanced-parsing-chunking-and-query-reformulation-giving-greater-control-of-accuracy-in-rag-based-applications/. Accessed: 2026-01-19.

[2] [n. d.]. Chroma. https://www.trychroma.com/. Accessed: 2026-01-19.

[3] [n. d.]. GPT-4o mini. https://platform.openai.com/docs/models/gpt-4o-mini. Accessed: 2026-01-19.

[4] [n. d.]. SWE-bench Lite. https://www.swebench.com/lite.html. Accessed: 2026-01-19.

[5] [n. d.]. text-embedding-3-small. https://platform.openai.com/docs/models/text-embedding-3-small. Accessed: 2026-01-19.

[6] [n. d.]. Tree-sitter. https://github.com/tree-sitter/tree-sitter. Accessed: 2026-01-19.

[7] Bui Thi Mai Anh and Nguyen Viet Luyen. 2021. An imbalanced deep learning model for bug localization. In *Proceedings of the 28th Asia-Pacific Software Engineering Conference Workshops*. IEEE, 32–40.

[8] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *Proceedings of the 28th International Conference on Software Engineering*. 361–370.

[9] Junming Cao, Shouliang Yang, Wenhui Jiang, Hushuang Zeng, Beijun Shen, and Hao Zhong. 2020. Bugpecker: Locating faulty methods with deep learning on revision graphs. In *Proceedings of the 35th International Conference on Automated Software Engineering*. 1214–1218.

[10] Partha Chakraborty, Mahmoud Alfadel, and Meiyappan Nagappan. 2024. Rlocator: Reinforcement learning for bug localization. *IEEE Transactions on Software Engineering* (2024).

[11] Partha Chakraborty, Mahmoud Alfadel, and Meiyappan Nagappan. 2025. BLAZE: Cross-language and cross-project bug localization via dynamic chunking and hard example learning. *IEEE Transactions on Software Engineering* (2025).

[12] Jianming Chang, Xin Zhou, Lulu Wang, David Lo, and Bixin Li. 2025. Bridging Bug Localization and Issue Fixing: A Hierarchical Localization Framework Leveraging Large Language Models. *arXiv preprint arXiv:2502.15292* (2025).

[13] Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 8697–8727.

[14] Agnieszka Ciborowska and Kostadin Damevski. 2022. Fast changeset-based bug localization with bert. In *Proceedings of the 44th International Conference on Software Engineering*. 946–957.

[15] Yali Du and Zhongxing Yu. 2023. Pre-training code representation with semantic flow graph for effective bug localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 579–591.

[16] Mikołaj Fejzer, Jakub Narębski, Piotr Przymus, and Krzysztof Stencel. 2021. Tracking buggy files: New efficient adaptive bug localization algorithm. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2557–2569.

[17] Jiaxuan Han, Cheng Huang, Siqi Sun, Zhonglin Liu, and Jiayong Liu. 2023. bjXnet: an improved bug localization model based on code property graph and attention mechanism. *Automated Software Engineering* 30, 1 (2023), 12.

[18] Yikun Han, Chunjiang Liu, and Pengfei Wang. 2023. A comprehensive survey on vector database: Storage and retrieval technique, challenge. *arXiv preprint arXiv:2310.11703* (2023).

[19] Thomas Hirsch and Birgit Hofer. 2023. The MAP metric in information retrieval fault localization. In *Proceedings of the 38th International Conference on Automated Software Engineering (ASE)*. IEEE, 1480–1491.

[20] Shahid Iqbal, Rashid Naseem, Salman Jan, Sami Alshmrany, Muhammad Yasar, and Arshad Ali. 2020. Determining bug prioritization using feature reduction and clustering with classification. *IEEE Access* 8 (2020), 215661–215678.

[21] Zhonghao Jiang, David Lo, and Zhongxin Liu. 2025. Agentic Software Issue Resolution with Large Language Models: A Survey. *arXiv preprint arXiv:2512.22256* (2025).

[22] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. Issue Localization via LLM-Driven Iterative Code Graph Searching. In *Proceedings of the 40th International Conference on Automated Software Engineering (to appear)*.

[23] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[24] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (2024), 1424–1446.

[25] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *Proceedings of the 45th International Conference on Software Engineering*. IEEE, 2312–2323.

[26] Saket Khatiwada, Miroslav Tushev, and Anas Mahmoud. 2018. Just enough semantics: An information theoretic approach for IR-based software bug localization. *Information and Software Technology* 93 (2018), 45–57.

[27] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406* (2022).

[28] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering* 39, 11 (2013), 1597–1610.

[29] Misoo Kim and Eunseok Lee. 2019. A novel approach to automatic query reformulation for ir-based bug localization. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 1752–1759.

[30] Misoo Kim and Eunseok Lee. 2021. Are datasets for information retrieval-based bug localization techniques trustworthy? Impact analysis of bug types on IRBL. *Empirical Software Engineering* 26 (2021), 1–66.

[31] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[32] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, Xin Zhou, Enzhi Wang, and Xiaohang Dong. 2023. Better zero-shot reasoning with role-play prompting. *arXiv preprint arXiv:2308.07702* (2023).

[33] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2019. D&c: A divide-and-conquer approach to ir-based bug localization. *arXiv preprint arXiv:1902.02703* (2019).

[34] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2015. Combining deep learning with information retrieval to localize buggy files for bug reports. In *Proceedings of the 30th International Conference on Automated Software Engineering*. IEEE, 476–481.

[35] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE, 218–229.

[36] Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. 2018. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *Proceedings of the 27th International Symposium on Software Testing and Analysis*. 61–72.

[37] Jae Yong Lee, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2024. The github recent bugs dataset for evaluating llm-based debugging applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 442–444.

[38] Han Li, Yuling Shi, Shaoxin Lin, Xiaodong Gu, Heng Lian, Xin Wang, Yantao Jia, Tao Huang, and Qianxiang Wang. 2025. Swe-debate: Competitive multi-agent debate for software issue resolution. *arXiv preprint arXiv:2507.23348* (2025).

[39] Jiawei Li, David Faragó, Christian Petrov, and Iftekhar Ahmed. 2024. Only diff is not enough: Generating commit messages leveraging reasoning and action of large language model. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 745–766.

[40] Wei Li, Qingan Li, Yunlong Ming, Weijiao Dai, Shi Ying, and Mengting Yuan. 2022. An empirical study of the effectiveness of IR-based bug localization for large-scale industrial projects. *Empirical Software Engineering* 27, 2 (2022), 47.

[41] Yue Li, Bohan Liu, Ting Zhang, Zhiqi Wang, David Lo, Lanxin Yang, Jun Lyu, and He Zhang. 2025. A Knowledge Enhanced Large Language Model for Bug Localization. *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (2025), 1914–1936.

[42] Zhengliang Li, Zhiwei Jiang, Qiguo Huang, and Qing Gu. 2025. LLM-BL: Large Language Models are Zero-Shot Rankers for Bug Localization. In *Proceedings of the 33rd International Conference on Program Comprehension*. IEEE Computer Society, 548–559.

[43] Hongliang Liang, Dengji Hang, and Xiangyu Li. 2022. Modeling function-level interactions for file-level bug localization. *Empirical Software Engineering* 27, 7 (2022), 186.

[44] Guangliang Liu, Yang Lu, Ke Shi, Jingfei Chang, and Xing Wei. 2019. Mapping bug reports to relevant source code files based on the vector space model and word embedding. *IEEE Access* 7 (2019), 78870–78881.

[45] Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. 2024. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644* (2024).

[46] Zheng Liu, Yujia Zhou, Yutao Zhu, Jianxun Lian, Chaozhuo Li, Zhicheng Dou, Defu Lian, and Jian-Yun Nie. 2024. Information retrieval meets large language models. In *Companion Proceedings of the ACM Web Conference 2024*. 1586–1589.

[47] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. 2008. Source code retrieval for bug localization using latent dirichlet allocation. In *Proceedings of the 15th Working Conference on Reverse Engineering*. IEEE, 155–164.

[48] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. 2010. Bug localization using latent dirichlet allocation. *Information and Software Technology* 52, 9 (2010), 972–990.

[49] Zhengmao Luo, Wenyao Wang, and Caichun Cen. 2022. Improving bug localization with effective contrastive learning representation. *IEEE Access* 11 (2022), 32523–32533.

[50] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2025. Alibaba lingmaagent: Improving automated issue resolution via comprehensive repository exploration. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 238–249.

[51] Zexiong Ma, Chao Peng, Qunhong Zeng, Pengfei Gao, Yanzhen Zou, and Bing Xie. 2025. Tool-integrated reinforcement learning for repo deep search. *arXiv preprint arXiv:2508.03012* (2025).

[52] Jesse Maarleveld, Jiapan Guo, and Daniel Feitosa. 2025. Gotta catch'em all! Towards File Localisation from Issues at Large. *arXiv preprint arXiv:2507.18319* (2025).

[53] Patrick E McKnight and Julius Najab. 2010. Mann-whitney U test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[54] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the use of stack traces to improve text retrieval-based bug localization. In *Proceedings of the International conference on software maintenance and evolution*. IEEE, 151–160.

[55] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. 2021. Industry-scale ir-based bug localization: A perspective from facebook. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 188–197.

[56] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 26th International Conference on Automated Software Engineering*. IEEE, 263–272.

[57] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard coefficient for keywords similarity. In *Proceedings of the International Multiconference of Engineers and Computer Scientists*, Vol. 1. 380–384.

[58] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2024. Repograph: Enhancing ai software engineering with repository-level code graph. *arXiv preprint arXiv:2410.14684* (2024).

[59] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffle: Bug localization on millions of files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 225–236.

[60] Binhang Qi, Hailong Sun, Wei Yuan, Hongyu Zhang, and Xiangxin Meng. 2021. Dreamloc: A deep relevance matching-based framework for bug localization. *IEEE Transactions on Reliability* 71, 1 (2021), 235–249.

[61] Yihao Qin, Shangwen Wang, Yan Lei, Zhuo Zhang, Bo Lin, Xin Peng, Jun Ma, Liqian Chen, and Xiaoguang Mao. 2025. Fault Localization from the Semantic Code Search Perspective. (2025). doi:10.1145/3757915

[62] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2025. Soap FL: A Standard Operating Procedure for LLM-based Method-Level Fault Localization. *IEEE Transactions on Software Engineering* (2025).

[63] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. Enhancing Fault Localization Through Ordered Code Analysis with LLM Agents and Self-Reflection. *arXiv preprint arXiv:2409.13642* (2024).

[64] Md Nakhla Rafi, Lorena Barreto Simedo Pacheco, An Ran Chen, Jinqiu Yang, and Tse-Hsun Peter Chen. 2026. SBEST: Spectrum-based fault localization without fault-triggering tests. *Empirical Software Engineering* 31, 1 (2026), 16.

[65] Mohammad Masudur Rahman and Chanchal K Roy. 2018. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 26th ACM joint meeting on European software Engineering Conference and Symposium on the Foundations of Software Engineering*. 621–632.

[66] Michael Rath, David Lo, and Patrick Mäder. 2018. Analyzing requirements and traceability information to improve bug localization. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 442–453.

[67] Matthew Renze. 2024. The effect of sampling temperature on problem solving in large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 7346–7356.

[68] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *Proceedings of the 47th International Conference on Software Engineering*. IEEE, 963–974. https://doi.org/10.1109/ICSE55347.2025.00080

[69] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Proceedings of the 28th International Conference on Automated Software Engineering*. IEEE, 345–355.

[70] Asif Mohammed Samir and Mohammad Masudur Rahman. 2025. Improved IR-Based Bug Localization with Intelligent Relevance Feedback. In *Proceedings of the 33rd International Conference on Program Comprehension*. IEEE, 560–571.

[71] Qusay Idrees Sarhan and Árpád Beszédes. 2022. A survey of challenges in spectrum-based software fault localization. *ieee access* 10 (2022), 10618–10639.

[72] Xinyu Shi, Zhenhao Li, and An Ran Chen. 2025. Enhancing LLM-based Fault Localization with a Functionality-Aware Retrieval-Augmented Generation Framework. *arXiv preprint arXiv:2509.20552* (2025).

[73] Bunyamin Sisman and Avinash C Kak. 2012. Incorporating version histories in information retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE, 50–59.

[74] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems* 37 (2024), 51963–51993.

[75] Stephen W Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E Hassan. 2013. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1427–1443.

[76] Yao Tian, Ziyang Yue, Ruiyuan Zhang, Xi Zhao, Bolong Zheng, and Xiaofang Zhou. 2023. Approximate Nearest Neighbor Search in High Dimensional Vector Databases: Current Research and Future Directions. *IEEE Data Eng. Bull.* 46, 3 (2023), 39–54.

[77] Bei Wang, Ling Xu, Meng Yan, Chao Liu, and Ling Liu. 2020. Multi-dimension convolutional neural network for bug localization. *IEEE Transactions on Services Computing* 15, 3 (2020), 1649–1663.

[78] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*. 53–63.

[79] Shaowei Wang and David Lo. 2016. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process* 28, 10 (2016), 921–942.

[80] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

[81] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Proceedings of the 31st International Conference on Automated Software Engineering*. 262–273.

[82] Ratnadira Widyasari, Stefanus Agus Haryono, Ferdian Thung, Jieke Shi, Constance Tan, Fiona Wee, Jack Phan, and David Lo. 2022. On the influence of biases in bug localization: Evaluation and benchmark. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 128–139.

[83] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 181–190.

[84] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[85] W Eric Wong and TH Tse. 2023. *Handbook of software fault localization: foundations and advances*. John Wiley & Sons.

[86] Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. 2023. Large language models in fault localisation. *arXiv preprint arXiv:2308.15276* (2023).

[87] Zihao Wu. 2025. Autono: A ReAct-Based Highly Robust Autonomous Agent Framework. *arXiv preprint arXiv:2504.04650* (2025).

[88] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (2025), 801–824.

[89] Xi Xiao, Renjie Xiao, Qing Li, Jianhui Lv, Shunyan Cui, and Qixu Liu. 2023. BugRadar: Bug localization by knowledge graph link prediction. *Information and Software Technology* 162 (2023), 107274.

[90] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology* 105 (2019), 17–29.

[91] Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. 2025. FlexFL: Flexible and Effective Fault Localization with Open-Source Large Language Models. *IEEE Transactions on Software Engineering* (2025).

[92] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[93] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.

[94] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 689–699.

[95] Xin Ye, Razvan Bunescu, and Chang Liu. 2015. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. *IEEE Transactions on Software Engineering* 42, 4 (2015), 379–402.

[96] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*. 404–415.

[97] Klaus Changsun Youm, June Ahn, and Eunseok Lee. 2017. Improved bug localization based on code change histories and bug reports. *Information and Software Technology* 82 (2017), 177–192.

[98] Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. 2025. Orcaloca: An llm agent framework for software issue localization. In *Proceedings of the 42nd International Conference on Machine Learning*. https://openreview.net/forum?id=LyUfPOvM6I

[99] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. 2020. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology* 124 (2020), 106312.

[100] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. 2020. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*. 219–229.

[101] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.

[102] Chunchun Zhao and Sartaj Sahni. 2019. String correction using the Damerau-Levenshtein distance. *BMC bioinformatics* 20 (2019), 1–28.

[103] Chunying Zhou, Xiaoyuan Xie, Gong Chen, Peng He, and Bing Li. 2026. Multi-view adaptive contrastive learning for information retrieval based fault localization. *Automated Software Engineering* 33, 1 (2026), 1–34.

[104] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 14–24.

[105] Yuqi Zhu, Jia Li, Ge Li, YunFei Zhao, Zhi Jin, and Hong Mei. 2024. Hot or cold? adaptive temperature sampling for code generation with large language models. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*, Vol. 38. 437–445.

[106] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019), 332–347.