BugLLM: Explainable Bug Localization through LLMs

by

Vikram N. Subramanian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Bug localization is the process of identifying the files in a codebase that contain a bug based on a bug report. This thesis presents BugLLM, a novel zero-shot bug localization method leveraging Large Language Models (LLMs) and semantic search techniques. BugLLM comprises two main phases: ingestion and inference.

In the ingestion phase, the codebase is chunked using an Abstract Syntax Tree (AST) parser, embedded using OpenAI's Ada V2 model and indexed in a Milvus vector database for efficient querying.

In the inference phase, a query is built from the bug report using an LLM to filter out non-technical details. This refined query is then used to search the vector database, retrieving semantically similar code chunks. These chunks undergo further filtering using another LLM query to establish their relevance to the bug, ensuring only the most pertinent chunks are considered.

Our method was evaluated on a dataset that includes bugs from six large Java projects. The evaluation metrics used include top-5 accuracy, where BugLLM achieved a top-5 accuracy ranging from 44.7% to 61.1%. BugLLM's performance was competitive, often surpassing traditional methods, and demonstrated efficiency with no training required.

To further aid developers, BugLLM also generates explanations for why specific files are relevant to a bug. The motivation behind this is twofold: helping developers understand why a file is important to fixing a bug and increasing transparency about how our tool works. Our methodology employs Chain-of-Thought prompting to generate detailed explanations from LLMs. These explanations are evaluated based on technical accuracy, groundedness, and informativeness. We find that the explanations generated by BugLLM are largely accurate and grounded in the actual content and context of the code, with minimal hallucination. The explanations were also found to be informative, providing valuable insights to developers. The mean scores (out of 5) for technical accuracy, groundedness, and informativeness were 3.9, 4.5, and 4.3, respectively, across different prompting techniques.

Acknowledgements

To my parents and my sister.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Developers spend most of their time fixing bugs [19]. Currently, there are over 29 million developers in the world [6]. This means that even a small reduction in time spent fixing bugs could result in massive productivity increases. The first step in fixing a bug is identifying where that bug is. This is the problem bug localization (BL) aims to solve. In this work, we propose a new zero-shot bug localization method that leverages large language models (LLMs) and semantic search techniques.

Once the files containing a bug have been identified, the next natural step for a developer trying to solve that bug is understanding what exactly is wrong. In this work, we also aim to assist with this step. We try to produce explanations about why a file is relevant for fixing a bug.

When a new bug has been identified, most modern software engineering teams file a bug report (also known as a ticket) on an issue tracking system like Jira, GitHub Issues, or Bugzilla [19]. This bug report usually has a title with a short summary and a larger body with various details. The body can include details about the bug, any error logs it produced, steps to reproduce, possible solutions, etc. See Fig. 1.1 for an example ticket from the AspectJ project[1].

Given a ticket, identifying which files in that codebase need to be modified to fix that bug is the precise definition of the bug localization problem.

1.1 Problem Statement

An effective bug localization system has the potential to make software developers more productive. It can help developers debug and fix bugs much faster. However, building an accurate and effective bug localization system that is general-purpose and easy to set up is not a solved problem. This work tries to solve that problem.

Thesis Statement: Embedding-based semantic searching and large language models as reasoning agents can be used to create a general-purpose and easy-to-set-up bug localization system that also provides explanations.

Several techniques have previously been proposed to perform bug localization [1, 27, 24, 44, 60, 67, 21, 43, 50], but all these techniques either leverage some property about the code

[1] https://eclipse.dev/aspectj/

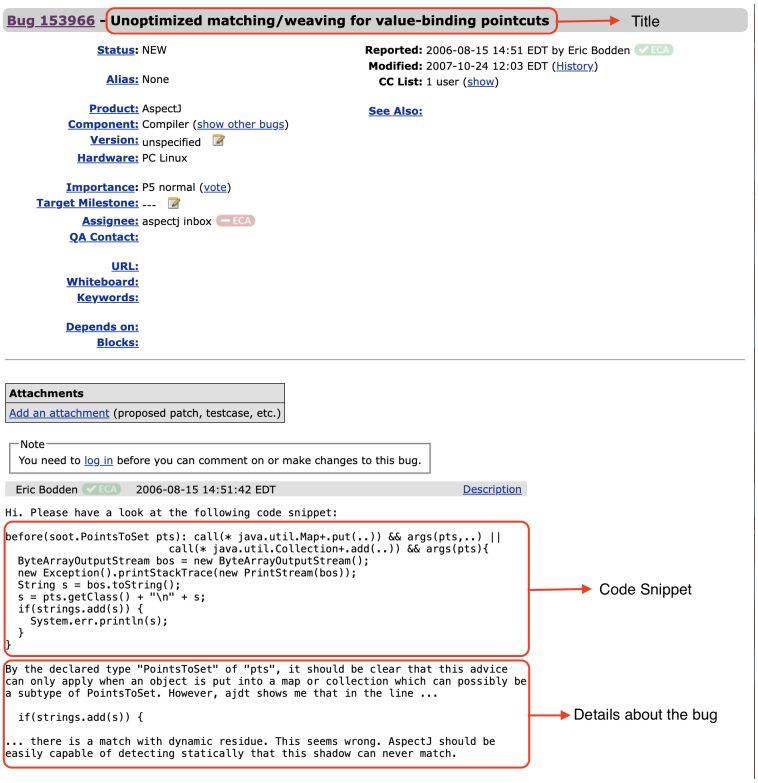

Figure 1.1: An example ticket

that severely limits their applicability to a wide set of projects, or require expensive and time-consuming setups such as training of models or simply suffer from poor performance. This makes these methods impractical to use in the real world.

Additionally, a bug localization system that can also produce explanations could further help newcomers to a project who are unfamiliar with the codebase. Also, technical debt is a major problem in many software engineering teams [20]. By explaining why a file is relevant, we could help developers better navigate this technical debt by providing insights they would otherwise have to obtain from reading the entire file. To the best of our knowledge, no prior work has successfully combined bug localization with the generation of detailed explanations for why specific files are relevant to the identified issues.

1.2 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 provides an overview of key concepts. Chapter 3 examines related research on bug localization and explainable AI applied to software engineering problems. Chapter 4 details our new bug localization method, BugLLM, while Chapter 5 discusses the evaluation of our method. Chapter 6 describes our explainable bug localization approach and evaluates its performance. Finally, Chapter 7 summarizes and concludes our work.

Chapter 2

Background

In this chapter, we will provide a brief overview of some of the important concepts and ideas leveraged in this work.

2.1 Large Language Models

Large Language Models (LLMs) are highly sophisticated AI models that are able to understand and generate human-like text. They are built using transformers, a type of deep learning architecture proposed by Vaswani et al. [49].

LLMs are able to understand concepts and generate detailed responses through their ability to learn from vast amounts of data. They can identify patterns and relationships within the data, allowing them to engage in complex reasoning. The LLM we use, GPT-4, has been trained on vast amounts of code and technical data. This allows it to "understand" and engage in complex reasoning about code. We leverage this ability extensively in this work.

2.2 Prompting

Prompting refers to the method of instructing LLMs to perform specific tasks by providing them with initial text (the prompt) containing instructions and guidelines. The technique of prompting exploits the model's ability to generate relevant continuations based on a given starter text. Different prompting strategies can significantly affect the output of the model.

Prompting strategies include Zero-Shot, Few-Shot [53], and Chain-of-Thought (CoT) [55] prompting. Zero-Shot prompting asks the model to perform a task without any examples, relying on its pre-existing knowledge. Few-Shot prompting provides a few examples to guide the model's output. Chain-of-Thought prompting involves providing a series of logical steps or intermediate thoughts, which helps the model reason through complex tasks more effectively. Chain-of-Thought prompting often significantly improves the reasoning capabilities of LLMs. By breaking down the problem into smaller, manageable parts, the model can generate more accurate and logical responses. This method mimics human problem-solving processes, where each step builds upon the previous one, leading to a well-reasoned conclusion.

2.3 Embeddings and Semantic Searching

Embeddings are a type of data representation where words, phrases, code, or other types of data are mapped to a multidimensional array of real numbers called vectors. These vectors are designed such that semantically similar pieces of data have similar representations. This allows machines to understand and process human language and code by capturing semantic meaning. This is achieved using techniques like Word2Vec [34], GloVe [38], or transformer models [49]. By analyzing large corpora of text, these methods learn to place semantically related words close to each other in the embedding space. In this paper, we use CPT [36], a transformer-based embedding model.

A vector database is a specialized type of database designed to store and manage embeddings. These databases index high-dimensional vectors and provide efficient querying capabilities for operations like similarity searches. They store embeddings in a structured format, allowing for quick retrieval of the most similar vectors to a given query.

2.4 Tokens, Context Windows, and Chunking

Tokens are the basic units of text used by LLMs. They can represent whole words, subwords, or even characters, depending on the tokenization strategy employed by the model. For example, the sentence "The quick brown fox" might be tokenized into ["The", "quick", "brown", "fox"].

The context window of an LLM refers to the maximum number of tokens the model can consider at one time. This is a crucial aspect of transformer models, as it defines the limit within which the model can capture dependencies and generate relevant responses. For instance, GPT-3 has a context window of 2048 tokens, meaning it can process and generate responses based on the last 2048 tokens of input.

Due to the limitations of context windows, chunking is necessary when dealing with large texts or codebases. Chunking involves breaking down a large document into smaller, manageable pieces that fit within the model's context window.

Chapter 3

Related Work

This chapter will highlight existing work related to bug localization and explainable AI in software engineering.

3.1 Bug Localization

Several approaches have been proposed to perform bug localization

3.1.1 Program Analysis

Several bug localization methods use techniques like static and dynamic analysis. Abreu et al. [1] use statistical analysis of program spectra to correlate system failures with specific parts of the system. Liu et al. [27] use statistical methods to model predicate evaluations, identifying fault-relevant predicates through significant divergence patterns in execution. Liblit et al. [25] identify and separate the effects of multiple bugs using predictors that correlate with program failures.

3.1.2 Information Retrieval

A significant amount of work has been done to apply information retrieval and search techniques to the bug localization problem. Our work derives inspiration from the research presented here. Rao et al. [44] used simple text models like the Unigram Model to perform bug localization. Ye et al. [60] used a learning-to-rank approach leveraging domain knowledge and features like method decompositions, API descriptions, bug-fixing, and code change history. They also produced the dataset we will use to evaluate our work. Zhou et al. [67] proposed a method using a revised Vector Space Model (rVSM) to rank source files based on textual similarity to bug reports and past fixes. Liang et al. [21] proposed FLIM, a framework that extracts semantic features at the function level to bridge the lexical gap between bug reports and code. They combine function-level interactions and information retrieval features using a fine-tuned language model and a learning-to-rank model.

Rahman et al. [43] proposed the key innovation of building a prompt from a bug description by removing extraneous details. Wang et al. [50] proposed a method that uses "query filtering" to build a query from noisy search input.

3.1.3 Machine Learning

Several researchers have proposed training a custom machine learning model for specific codebases to accurately perform bug localization. Zhang et al. [63] built KGBugLocator by using knowledge graph embeddings and a bi-directional attention mechanism to extract semantic and relational information from code. Peters et al. [39] built a bi-directional language model (biLM) that captures complex word characteristics and context variation to perform bug localization. Zhu et al. [69] built BL-GAN, a semi-supervised bug localization model using a Generative Adversarial Network (GAN) and a Transformer-based architecture. They incorporate Graph Convolutional Networks to process code structure. FBL-BERT proposed by Ciborowska et al. [3] applies a BERT-based model, leveraging pre-training of BERT and fine-tuning on project-specific datasets. It uses an efficient vector similarity search for retrieval.

Xiao et al. [58] and Liang et al. [23] both propose a deep learning-based machine translation technique to bridge the lexical gap between bug reports and source code. A key innovation described in the latter two publications, which we will also use, is building an abstract syntax tree representation of the code.

3.2 Explainability

Fu et al. [7] propose GPT2SP, a Transformer-based approach for Agile story point estimation. Similar to our work, their approach focuses on providing explanations to help developers understand the reasoning behind the AI's decisions. Pornprasit et al. [40] propose PyExplainer, a technique designed to explain the predictions of Just-In-Time (JIT) defect models. PyExplainer aims to provide developers with understandable reasons behind predictions, a goal that we seek to achieve in our work as well.

Mohammadkhani et al. [35] conducted a systematic literature review on explainable AI (xAI) for Software Engineering. Their study emphasized the need for xAI to build trust and transparency in AI models used in software engineering.

He et al. [10] propose a deep learning-based approach to determine and explain valid bug reports using textual information. By applying a convolutional neural network (CNN) to capture contextual and semantic features of bug reports, their method provides explanations by backtracking the trained CNN to extract key phrases. Their approach offers insights into why certain bug reports are classified as valid, enhancing the transparency and interpretability of the model's predictions.

Jiarpakdee et al. [14] investigated the application of explainable AI techniques in defect prediction models through a qualitative survey of practitioners. Practitioners highlighted the usefulness of these techniques in understanding the important characteristics that contributed to predictions, underscoring the significance of explainable AI in enhancing the interpretability of defect prediction models in software engineering.

Wattanakriengkrai et al. [54] propose a novel framework, LINE-DP, that leverages LIME to identify defective lines in source code. By first building a file-level defect model using code token features and then using LIME to highlight risky tokens, LINE-DP provides explanations for why certain lines are predicted to be defective.

Dam et al. [5] argue for the importance of explainability and transparency in software analytics models. They also try to define what good explanations look like in the software analytics space. We use their lead in this work.

Sun et al. [47] explore the explainability needs of generative AI models in software engineering. They identify 11 categories of explainability needs for generated code, including markers indicating areas of uncertainty and a requirement for comprehensive documentation.

Chapter 4

Our Method: BugLLM

In this section, we describe our novel method for bug localization. Our method is a tiered multi-step process with a distinct preprocessing ingestion phase, and an inference phase similar to several popular information retrieval systems as described by Manning et al. [32]. Figure 4.1 provides an overview of BugLLM. The teal section is our ingestion phase as described in Section 4.1, and the blue section is the inference phase as described in Section 4.2. Unlike other BL methods mentioned in Section 3.1, there is no training involved. Our system is zero-shot.

In this chapter, we assume we are given a version of a codebase that contains a bug and a description of the bug derived from the codebase's issue-tracking system (such as Jira or Bugzilla).

4.1 Preprocessing: Ingesting and Indexing

In this preprocessing step, we will iterate through each file in the codebase, chunk them, produce an embedding using our embedding model and index these embeddings in a vector database (DB) for querying. See Section 2 for an overview of chunking, indexing and vector databases.

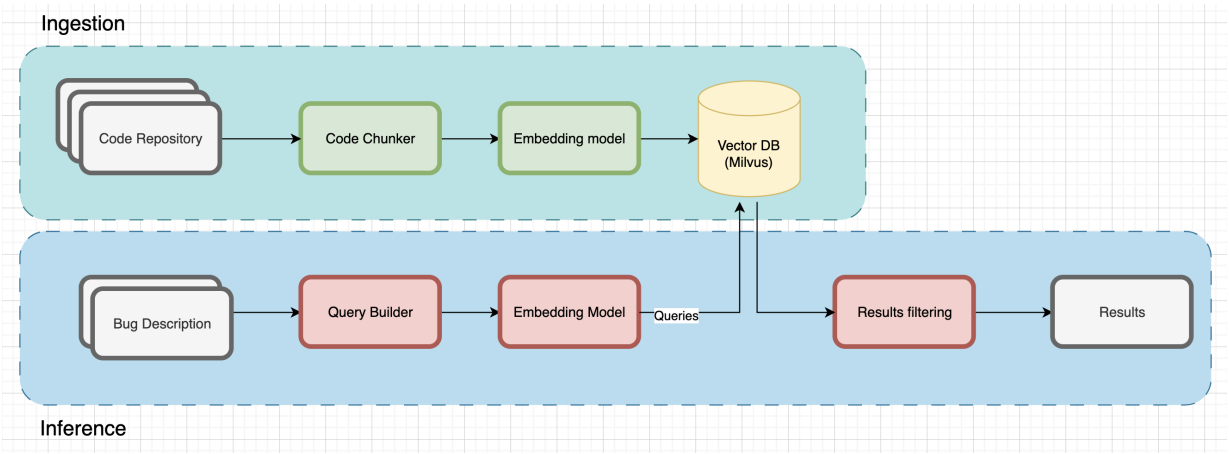

Figure 4.1: An Overview of BugLLM

Our selected embedding model, Ada v2 from OpenAI[1], has a token limit of 1536 tokens. This approximately corresponds to 90 lines of code on average based on analysis of our evaluation dataset (see Section 5.1). More than 72% of the files in our evaluation dataset were longer than 90 lines and did not fit within the token limit. Additionally, 13% of files were over 200 lines. Therefore, we use a chunking strategy to break up large files into smaller chunks.

We will use Tree-Sitter's Concrete Syntax Tree (CST) parser[2] to build an abstract syntax tree of the entire code file. We will then follow the greedy algorithm presented below to convert a code file into a list of chunks that have been split at the semantically least disruptive places. We got the inspiration for this technique from Llama Index [30].

Algorithm 1 Code Chunking Algorithm Using CST Parsers

Require: Root node of the syntax tree *node*, Source code as a string *text*, Tokenization check function *canTokenize(input_str)*

Initialize *new_chunks* as an empty list
Initialize *current_chunk* as an empty string
for each *child* in *node.children* **do**
  if *canTokenize(text[child.start_byte:child.end_byte])* returns **False then**
    Append *current_chunk* to *new_chunks*
    Reset *current_chunk* to empty
    Recursively chunk the *child* and append the result to *new_chunks*
  else if *canTokenize(current_chunk + text[child.start_byte:child.end_byte])* returns **False then**
    Append *current_chunk* to *new_chunks*
    Set *current_chunk* to the text of the *child*
  else
    Concatenate the text of the *child* to *current_chunk*
  end if
end for
return *new_chunks*

4.1.1 Why Chunk Using an AST?

Several previous BL methods [51, 52] use naive chunking strategies such as string slicing. This breaks semantic continuity. That is, if you break a function into two parts, the embedding model will fail to capture the semantic meaning of either part or the function as a whole. By being able to chunk functions or contiguous code blocks as one piece, we capture more semantic meaning. In the algorithm presented, we greedily try to chunk the largest semantic block possible without exceeding the token length.

Tree-Sitter's CST parser supports 113 programming languages. In the rare case that we are not able to support a file in our chunking process, we use a simple string-slicing chunking method.

[1] https://openai.com/index/new-and-improved-embedding-model/
[2] https://tree-sitter.github.io/tree-sitter/

4.1.2 Why Pick Ada V2?

Ada V2 is the production implementation of CPT [36]. CPT is the current leader in the CodeSearchNet Challenge [13] semantic code search leaderboard.

4.1.3 Choice of Vector Database

There are several vector databases available [59]. Our index size, even for the largest projects in our evaluation dataset (see 5.1), does not exceed 50,000 entries—a very small amount. Additionally, we are querying the DB at a very low rate. This means most DBs are functionally equivalent [59]. We chose a popular open-source DB: Milvus [3]

4.2 Inference

4.2.1 Inference I: Query Building

In querying the vector database we built in the ingestion phase, we are performing a semantic search against code blocks. The embeddings we generated try to capture the semantic meaning of the code blocks. We have stored these embeddings in a multi-dimensional space in our vector database. Any query also gets embedded into that same multi-dimensional space and the vector database returns the closest nodes to the query's position.

This means a ticket describing a bug is not the ideal search query while querying a vector database. It contains various extraneous details that do not describe the bug and/or the codebase in a technical manner. Bug 51929 (shown below) is an example of a ticket in the Ye et al. [60] dataset. It contains details that are clearly required to describe the bug, but will not help us search through code.

To filter out only the technical details to produce our query string, we use an LLM and the few-shot prompting strategy. Here is the prompt we used:

```
You are an experienced software developer tasked with analyzing
issue tickets from software project's issue trackers.
Your goal is to extract all meaningful and technical details
relevant for debugging. Follow these guidelines:

Identify Error Messages:
- Extract key parts of error messages verbatim.
- Example: If the ticket mentions "NullReferenceException:
            Object reference not set to an instance of an object,"
            extract exactly that.
Function and Variable Names:
 - List all function names, variable names, and other code elements
   mentioned in the ticket.
 - Example: If the ticket mentions "the error occurs in the
```

[3] https://milvus.io/

```
            calculateTotal() function when accessing the totalAmount
            variable," extract calculateTotal() and totalAmount.
Technical Descriptions:
 - Retain all technical information from the issue ticket.
 - Example: If the ticket provides details such as "the issue
            appears when the input list is empty," include this
            information.

Present the extracted information in a concise format, prioritizing clarity.
Be concise and to the point.
Do not print any extra information. Just the details from the bug.
```

When this prompt was run on Bug 51927 (shown below), the LLM produced this:

```
Error Messages:

 - "Errors occurred while applying CVS decorations to resources."
 - "Resource /org.eclipse.pde.ui does not exist."

Function and Variable Names:

build.properties
org.eclipse.core.internal.resources.ResourceException
org.eclipse.core.internal.resources.Resource.checkExists
org.eclipse.core.internal.resources.Resource.checkAccessible
org.eclipse.core.internal.resources.Project.checkAccessible
org.eclipse.core.internal.resources.Resource.getSessionProperty
org.eclipse.team.internal.ccvs.core.CVSTeamProvider.isWatchEditEnabled
org.eclipse.team.internal.ccvs.ui.CVSLightweightDecorator.getOverlay
org.eclipse.team.internal.ccvs.ui.CVSLightweightDecorator.decorate
...(Shortened for brevity by author)

Technical Descriptions:

 - Checked out the PDE UI project.
 - Opened and read the build.properties file during checkout.
 - Deleted the project potentially before the UI finished
   updating CVS decorators.
 - Received error dialogues for traceless exceptions.
 - Error logs indicate multiple instances of
   "Resource /org.eclipse.pde.ui does not exist."
```

This query only contains information that is relevant to the code—information that could point to a specific location in the codebase. Non-technical information or noise, such as timestamps in error messages or greetings that might not help fix a bug, is removed.

Out of 50 random tickets evaluated from six different projects in the Ye et al. [60] dataset, we found that over 70% of tickets contained at least some extra detail similar to

Bug 51927 (see below). Historically, most bug localization techniques have used the entire bug description as a search query [25, 21, 39, 3, 23, 58]. Using an LLM to build a query is a key innovation of BugLLM.

4.2.2 Inference II: Retrieval: Vector Searching

Using the search query we generated in the previous step, we query the vector database. We use cosine similarity as our metric to query the vector database. The cosine similarity between two embeddings is calculated as the dot product of the embeddings divided by the product of their lengths. It is a widely used measure of semantic similarity [4]

We get the 20 code chunks semantically most similar to the query embedding. These 20 chunks represent the possible snippets that could contain the bug. Note that chunks could be from the same file as we break up a file into one or more chunks (see 5.4.5). We pass these 20 chunks to the next step of our process.

4.2.3 Inference III: Filtering

In the previous step, we retrieved 20 possibly related code chunks. In this step, we try to concretely identify that relationship. If we are unable to do so, we remove that chunk from our results. We will use an LLM as a reasoning agent [66] that is able to understand code and reason about relevance to filter possibly relevant code snippets. We will ask the LLM to identify a connection between the code snippet and the ticket to establish a relationship.

We can control what a "connection" between a code block and bug description means by adjusting the prompt. We are asking the LLM to perform a certain level of speculation to identify connections. If tangible and precise connections between code blocks and bug descriptions existed, we could simply build a system that checks for those directly.

Here is the prompt we used:

```
You are an expert programmer and code reviewer.
Your task is to reason which SNIPPETS should actually
be reviewed by a developer to solve the ISSUE described.

You are given the following:
- Description of an ISSUE
- Multiple code or documentation SNIPPETS possibly related to the issue

For each snippet, you are given the details in the following format:

'''
Snippet ID: <id>
Filename: <filename>
Code: <code>
'''

Each snippet is separated by a line of dashes (---).
```

```
For each snippet, consider if the fix for the issue involves
changing this snippet. If it does,
it is a relevant snippet, and that snippet ID should be returned.

Be concise and to the point.
Return only the Snippet ID of the relevant snippets.
```

We arrived at this prompt after experimenting with several different variants and different prompting strategies. The output of the LLM is a series of snippet IDs, which we generated to help the LLM identify different snippets. We map these IDs to filenames and that is the final result of our bug localization method.

Bug 51929 from the AspectJ project

Bug Title: Exceptions when deleting project shortly after checkout
Bug Description:

```
I checked out the PDE UI project and while it was
still checking out I opened the build.properties file
(all I was really interested in), read it, and decided
I was done. I then deleted the project. I'm not sure if
the project was still checking out when I deleted it
(I don't think it was), but it looks like I deleted it
before the UI was done updating the CVS decorators.
I got an error dialog for each of the traceless exceptions
below and found the following in the log:

!MESSAGE Errors occurred while applying CVS decorations to resources.
!STACK 1
...(shortened for brevity by author)
!ENTRY org.eclipse.core.resources 4 368 Feb 12, 2004 17:09:55.62
!MESSAGE Resource /org.eclipse.pde.ui does not exist.
!ENTRY org.eclipse.core.resources 4 368 Feb 12, 2004 17:09:55.954
!MESSAGE Resource /org.eclipse.pde.ui does not exist.
!ENTRY org.eclipse.core.resources 4 368 Feb 12, 2004 17:09:55.958
!MESSAGE Resource /org.eclipse.pde.ui does not exist.
!ENTRY org.eclipse.core.resources 4 368 Feb 12, 2004 17:09:55.963
!MESSAGE Resource /org.eclipse.pde.ui does not exist.
!ENTRY org.eclipse.core.resources 4 368 Feb 12, 2004 17:09:55.987
...(shortened for brevity by author)
!ENTRY org.eclipse.core.resources 4 368 Feb 12, 2004 17:09:56.58
!MESSAGE Resource /org.eclipse.pde.ui does not exist.
!ENTRY org.eclipse.core.resources 4 368 Feb 12, 2004 17:09:56.62
!MESSAGE Resource /org.eclipse.pde.ui does not exist.
!ENTRY org.eclipse.core.resources 4 368 Feb 12, 2004 17:09:56.67
!MESSAGE Resource /org.eclipse.pde.ui does not exist.
!ENTRY org.eclipse.core.resources 4 368 Feb 12, 2004 17:09:56.71
!MESSAGE Resource /org.eclipse.pde.ui does not exist.
```

Chapter 5

Evaluation and Results

5.1 Dataset

We will evaluate our method on the benchmark dataset used by other bug localization studies, Ye et al.'s [60] dataset, which contains bugs from these six large Java projects:

1. **AspectJ:** An extension to Java that allows developers to modularize cross-cutting concerns, such as logging and security, into separate units.
2. **Eclipse:** An integrated development environment for Java.
3. **JDT:** A set of plugins for the Eclipse IDE that includes tools like a Java compiler, debugger, and code assist features.
4. **Tomcat:** A widely used Java HTTP web server.
5. **BIRT:** Tools to build data visualizations, reports, and business intelligence dashboards in Java.
6. **SWT:** A widget toolkit for Java designed to provide access to the user-interface facilities of the operating system.

All these projects are very large with hundreds of files and hundreds of thousands of lines of code. Hundreds of open-source developers have helped in building each project. All these projects are very mature with very high complexity.

5.2 Evaluation Metrics

To the best of our knowledge, our tool is the only one proposing a zero-shot, no-training approach using LLMs for bug localization. We will compare our results to other state-of-the-art bug localization methods by running our tool on the benchmark dataset from Ye et al. [60].

We will use the top-5 metric for evaluation. On average, our method produces around 3-5 results. The top-5 metric measures if at least one of our top five guesses is correct.

We will also measure the percentage of correct and incorrect guesses. If we produce 100 guesses across all our bugs and 20 are correct, we have a correct percentage of 20.0% and an incorrect percentage of 80.0%.

Most other bug localization solutions use Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) in addition to the top-5 metric to measure their performance. However, we cannot use MAP and MRR as we cannot produce an ordered set of results. MRR requires a rank order to calculate the reciprocal rank, and MAP requires a rank order to compute precision at each relevant item position.

We cannot produce an ordered set because our final filtration step returns a subset of chunks from a larger list (that is ordered by cosine similarity). This subset was made by an LLM trying to identify relationships the cosine similarity metric could not capture in the initial ordering. There is no way for us to quantify the accuracy of this process.

Project Name	**Top-5**	**Percentage Correct**	**Percentage Incorrect**
AspectJ	56.5%	43.4%	56.6%
Birt	44.7%	31.8%	68.2%
Eclipse Platform UI	54.3%	42.4%	57.6%
JDT	52.6%	39.7%	60.3%
SWT	61.1%	49.5%	50.5%
Tomcat	60.3%	48.1%	51.9%

Table 5.1: BugLLM performance

5.3 Evaluation Setup

In the previous section, while describing our method, we assumed we are given a version of a codebase that contains a bug and a description of the bug. Now we will describe how we achieved that for each bug in our dataset while running our evaluation scripts.

Ye et al.'s [60] dataset provides a list of bug descriptions with the associated commit hash that fixed that bug. This means the repository at the commit right before that hash contains the bug. We sort the dataset by the date at which the fixing commit was made. We then checkout[1] to the commit right before the earliest fixing commit. This version of the repository is ready to run BugLLM for the first bug in our dataset.

We ingest the entire repository from scratch as described in the previous section and run our bug localization method and save the result. To process the next bug on the dataset, we checkout the repository to the commit right before the fixing commit of that

[1]The git checkout command lets us go to a version of the codebase as referenced by the commit hash

bug. We perform a git diff to identify which files have been changed between the two commits. Any file that has been deleted will also be deleted from our index. Any file that has been modified will be replaced by running our ingestion process again and any file that has been added will also be ingested. We are now ready to run BugLLM on the second bug in our dataset.

We can similarly iterate through each bug and only modify the index to update changes instead of rebuilding our index for each commit. This saves considerable time and energy.

5.4 Results

The performance of our method is summarized in Table 5.1. Across the six projects, our top-5 accuracy varied, with Tomcat achieving the highest accuracy (60.3%) and BIRT the lowest (44.7%). Our variability in observed performance is consistent across other BL works, indicating an inherent difference in the code and/or bug reports of these various projects that is making it harder/easier for bug localization techniques to identify correct files.

5.4.1 Comparison With Other Methods

Project Name	**BugLLM**	**BugLocator [51]**	**DNNLoc [17]**
AspectJ	56.5%	51.05%	71.2%
Birt	44.7%	N/A	42.2%
Eclipse Platform UI	54.3%	53.76%	70.5%
JDT	52.6%	41.3%	65.0%
SWT	61.1%	67.35%	69.0%
Tomcat	60.3%	66.5%	72.9%

Table 5.2: Comparison of top-5 accuracy with other bug localization methods.

Our method requires no training, and no energy or computation is spent creating a specific model for each project. We can ingest an entire project within minutes and produce results. Additionally, our application is compatible with most major languages. In contrast to BugLLM, some other methods [39, 63, 69, 3, 58, 23] involve significant training and setup requirements. For instance, DNNLoc [17] employs a custom deep neural network model trained on that project's data to predict the location of bugs. This is reflected in DNNLoc's performance, which is noticeably better.

BugLocator [51] uses an information retrieval-based method, specifically utilizing a revised Vector Space Model (rVSM), to rank source code files based on their relevance to the bug report. This approach is similar to that of BugLLM. While the performance of both methods is comparable, BugLLM achieves better results on three of the five projects.

5.4.2 How Much Did Each Step Contribute

Query Builder

To understand how effective our query-building step was, we selected a hundred random bugs from the six projects in our evaluation dataset and ran our pipeline without the filtration step. It produced a top-5 score of 48.8%—an almost 8% reduction in performance from our full setup that was run on the same 100 bugs.

Filtration step

Our filtration step significantly improves the overall accuracy of our bug localization method, as shown in Table 5.3. This table presents the percentage of snippets that were removed, and the percentage of files removed that were false negatives. In this context, a false negative refers to a scenario where a correct answer is filtered out. The minimal number of false negatives indicates that our filtration step causes little harm, even in the worst-case scenario.

By querying for a large number of snippets and then applying filtration, we are able to perform a more comprehensive search. By casting a wider net, we can bridge the semantic gap inherent in querying code snippets using bug descriptions. Since our queries are not very precise, it is counterproductive to restrict our search narrowly. Therefore, by querying a wide range of snippets, we achieve a more thorough search that ultimately improves our bug localization results.

Project Name	**Pertange Filtered**	**Percentage False Negative**
AspectJ	83.0%	1.2%
Birt	79.2%	0.95%
Eclipse Platform UI	85.1%	0.81%
JDT	82.5%	1.7%
SWT	85.8%	2.1%
Tomcat	87.8%	1.5%

Table 5.3: Effectiveness of our filtration step

5.4.3 Timing Efficiency

The initial ingestion of the entire repository took between 15 and 25 minutes for all the projects we studied while actual inference for a specific bug took less than two minutes. Querying the vector database is very efficient, taking less than two seconds, while prompting our LLM takes between 30 seconds and a minute.

5.4.4 Improvements Over Time

Our method uses LLMs as reasoning agents and uses embedding models to capture semantic meaning. Our system is not tied to any one LLM or embedding model. This means as models improve in their reasoning ability, our approach also improves.

5.4.5 Chunks vs. Files

Our method recommends relevant chunks. In our method, a file is a series of one or more chunks. In the projects we studied, 72% of files have more than one chunk. Providing a more specific location within such large files, rather than just the file itself, could be much more useful for developers. However, the bug localization problem has been defined as predicting files by past work and therefore, we simply use the file the chunk is from as our final answer for our analysis and comparison with other methods.

5.5 Threats to Validity

1. The Ye et al. [60] dataset is not representative of all software projects. It only contains bugs from extremely large and complex Java projects. Our method will perform worse as complexity increases. We aim to capture semantic meaning and reason with code, and our ability to do so diminishes with increasing complexity.

2. The characteristics of open-source projects can differ significantly from those of commercial or smaller-scale projects, which may affect the generalizability of our findings.

3. Our reliance on embedding models and LLMs introduces another layer of variability. These models have inherent biases based on the data they were trained on. Our code search model will perform differently in different programming languages and our LLMs are inherently biased in how well they comprehend and reason with different programming languages.

4. Our method's performance is inherently tied to the quality of bug reports. Poorly written or incomplete bug reports can adversely affect the accuracy of our bug localization process.

5. The prompts used in our method have been designed to work with an "ideal" textbook bug report—that is, one that contains a title and a body with reproducibility details, crash logs, etc. While theoretically bug reports should conform to this format, they are not required to do so. This affects the generalizability of our method.

6. The LLMs and embedding models we used are offered by companies that are constantly modifying and improving them. These entities are not static. This might potentially affect the reproducibility of our results.

Chapter 6

Explainable Bug Localization

6.1 Motivation

Identifying the location of bugs can be helpful, but explaining why a particular file was picked could be much more useful. Developers need to first understand why a particular segment of code is faulty to effectively address the issue. An explanation that contextualizes the bug within the broader codebase, highlighting the specific reasons that contribute to its relevance, could be very useful. Just as bug localization makes developers more productive, going one step further and giving developers an explanation of what is happening in a file would reduce the time spent understanding code, further enhancing their productivity.

Every bug localization technique proposed has always been a black box. There is no clarity as to how or why a model chose a specific file. This lack of transparency makes these tools hard to trust and understand, and this directly inhibits adoption by developers. This has been proven in previous research by Jiarpakdee et al. [14]. Providing transparency through explainable artificial intelligence could help in this direction.

6.2 Methodology

6.2.1 Prompting for Explanation Generation

In this section, we outline our approach for generating explanations using LLMs. We start with the correct results from the previous section—a bug description and its related files. We then use the Chain-of-Thought [56] prompting method and iterate through each related file.

We use Chain-of-Thought because Li et al. [29] identified it as a very effective way to get LLMs to perform complex reasoning while producing explanations. Prompting an LLM is more of an art than a precise science [62], and we experiment with different prompts to produce the best possible results. We attempt two different prompting approaches:

1. **One-file-at-a-time :** We iterate through each relevant file or snippet and ask the LLM what needs to be changed in that one file and how that file is relevant to the described bug. The exact prompt we used:

```
You are an expert programmer and code reviewer.
Here is a description of a bug:
    <insert bug description>.
Here is a file that is related to this bug:
    <insert file snippet>.
For the file, you are given the details in the
following format:
‘‘‘
Filename: <filename>
snippet: <snippet>
‘‘‘
How is this file relevant to the described bug?
What would need to be changed in this file, if any,
to solve this bug?
Please provide an explanation.
Be concise and to the point.
```

2. **All-the-context-at-once:** By providing more context, we give the LLM a better understanding of the whole picture (the theorized trade-off being that each snippet receives less attention compared to the previous method). We give the LLM all the related snippets and ask it to tell us what needs to be changed in each file and how each file is relevant to the described bug.

```
You are an expert programmer and code reviewer.
Here is a description of a bug:
    <insert bug description>.
Here is a series of files related to this bug:
    <insert file snippets>.
For each file, you are given the details in the
following format:
‘‘‘
Filename: <filename>
snippet: <snippet>
‘‘‘
 For each snippet consider the following:
    1. How is this file relevant to the described bug?
    2. What would need to be changed in this file, if any,
        to solve this bug?

Please provide an explanation.
Be concise and to the point.
```

6.3 Results

Below are some sample explanations we produced:

AspectJ Bug 53999: Explanations produced with the one-file-at-a-time method

Bug Title: "can't bind type" message has $ for . in type name for declare soft

Bug Description:

```
cosmetic mistake in error message...

expected: can't bind type name 'unknown.Error'

actual: can't bind type name 'unknown$Error'

(not sure why it would ever be munged with $)

public class Test {
 public static void main(String[] args) {
  test();
 }
 static void test() {
  throw new Error("hello");
 }
 static aspect A {
  declare soft : unknown.Error : call(void test()); // CE
 }
}
```

File: weaver/patterns/DeclareSoft.java

Explanation: Contains the type binding logic and error message generation. Necessary to ensure the type name is formatted correctly by replacing `$` with `.` before displaying the error message.

AspectJ Bug 82062: Explanations produced with the one-file-at-a-time method

Bug Title: WeaveMessage should provide more information
Bug Description: It would be wonderful if the WeaveMessage object would provide two additional things:
- the affected class name
- the aspect name
I could use this to determine which aspect is woven into which class during load-time weaving (need this for dependency management).
File: bridge/src/org/aspectj/bridge/WeaveMessage.java
Explanation: This snippet contains the 'WeaveMessage' class and the 'constructWeavingMessage' method, which need to be updated to include new placeholders for source file and line number information.

AspectJ Bug 48990: Explanations produced with the all-context-at-once method

Bug Title: Special case if(false) to not require a dynamic test
Bug Description: FastMatch, the efficient pattern matching for pointcuts on the constant pool, is currently only implemented for the within pointcut. I am planning to provide further implementations for other pointcuts by release 1.2.
File: ajdt/internal/compiler/ast/FalseLiteral.java
Explanation: The FalseLiteral class needs to be modified to optimize if(false) constructs by skipping unnecessary code generation.

The first two examples were produced by the one-file-at-a-time method while the third example was produced by the all-context-at-once method. While the explanations produced through the all-context method were slightly shorter in length on average, the results were virtually indistinguishable in terms of style of output. This makes sense as we essentially gave the same prompt to the same LLM. There was a difference in quality of content produced which we measure in the next section.

6.4 Evaluation

We will now evaluate the quality of our explanations by providing an LLM with the "answer"—i.e., the code fix as a code diff—and asking it to rate the quality of our explanation.

6.4.1 Evaluation Criteria

It is important that our explanations are correct on various fronts. Specifically, we will measure these criteria:

1. **Technical Accuracy**: The LLM should have understood the bug description and the file in question. It should then provide technical reasoning as to why this file is related to the bug. The explanation produced should be consistent with the project, the specific piece of code, and the specific bug. We will use a score from 1 (completely inaccurate) to 5 (very accurate) to measure this.
2. **Groundedness**: LLMs can hallucinate explanations [28]. We need to ensure that all our generated explanations are based on evidence from the bug descriptions and/or the file. We will use a rating from 1 (no groundedness) to 5 (fully grounded) to measure this.
3. **Informativeness**: The more detailed and insightful an explanation, the more helpful it is to a developer. We will try to gauge whether the LLM is able to provide complex reasoning about a file's relevance to help the developer, or if it just summarizes the code or mentions the obvious. Furthermore, the explanation should focus on pieces of information that are relevant to the final solution. We will use a rating from 1 (not informative) to 5 (very informative) to measure this.

6.4.2 Evaluation Method

To evaluate the quality of our generated explanations, we will use an LLM just like in G-Eval [29]. The LLM will be provided with the bug description, the relevant file, the explanation for that file, and the bug fix as a code diff for that file. The LLM will then produce ratings for the three criteria mentioned above. This process will be looped through for every file that had an explanation produced.

Below is the Chain-of-Thought prompt used for this evaluation:

```
You are an expert software developer tasked with
evaluating the quality of generated explanations
for bug fixes.

Here is a description of a bug:
<insert bug description>.

Here is the relevant file before the fix:
<insert file before fix>.

Here is the file after the fix (as a diff):
<insert file after fix>.

Here is the generated explanation for the file:
<insert generated explanation>.
```

```
Please evaluate the explanation based on the following criteria:

1. **Technical Accuracy (1-5)**: How accurately does the
    explanation describe why the file is relevant to the
    bug and what changes are needed?
        - Consider if the explanation correctly identifies the issue
          in the code and provides a precise reasoning for the necessary
          changes.

2. **Groundedness (1-5)**: How well is the explanation backed
    by evidence from the bug description and file contents?
        - Assess if the explanation uses specific details from
          the bug description and code to support its reasoning,
          avoiding fabricated or irrelevant information.

3. **Informativeness (1-5)**: How detailed and helpful is the
    explanation in understanding the relevance of the file to the bug?
       - Determine if the explanation provides insightful details,
         helping the developer understand the code and the bug better.

Provide your ratings for each criterion and
a brief justification for your ratings:

Technical Accuracy: <insert rating>
Justification: <insert justification>

Groundedness: <insert rating>
Justification: <insert justification>

Informativeness: <insert rating>
Justification: <insert justification>
```

6.4.3 Evaluation Results

Prompting Technique	Mean Technical Accuracy	Mean Groundedness	Mean Informativeness
One-file-at-a-time	3.9	4.6	4.3
All-the-context-at-once	3.9	4.4	4.2

Table 6.1: Evaluation of Explanation Quality

Table 6.1 lists the evaluation results of our explanation generation method, comparing the two prompting techniques: one-file-at-a-time and all-context-at-once.

The results we observed indicate that our method achieves a decent level of technical accuracy. The one-file-at-a-time approach and the all-context-at-once approach both scored 3.9 on average. This suggests that while the LLMs understand and correctly process most

of the technical details, sometimes they fall short. We theorize that a lack of complete context or background about the code might be the reason for this drop in performance.

One of the key observations from our results is the low level of hallucination. Historically, LLMs are known to hallucinate explanations when they lack sufficient information [28]. However, our explanations were largely backed by what is present in the code, as reflected by the groundedness scores of 4.6 and 4.4 for the one-file-at-a-time and all-context-at-once approaches, respectively. This indicates that the explanations produced are firmly based on the actual content and context of the code, reducing the risk of fabricated information.

While informativeness is somewhat subjective, our method performs well in this regard, scoring 4.3 and 4.2 for the one-file-at-a-time and all-context-at-once methods, respectively. This suggests that the explanations generated are not only clear but also valuable, helping developers understand the relevance of the code snippets to the described bugs.

Incorrect Results Misleading Developers

An explanation with a very low technical accuracy score or groundedness score could potentially mislead and confuse a developer and do more harm than good. Fortunately, only 4% of explanations produced a groundedness score less than three and only 9% of explanations produced a technical accuracy score less than three. We theorize that the potential benefits of producing an accurate explanation for over 90% of the files outweigh the cost of producing incorrect explanations in some cases. Future work could measure the precise impact on developer productivity.

6.5 Threats to Validity

We used an LLM to reason complexly and produce explanations, and then used an LLM with the correct answer as context to grade the correctness of the produced explanations. There is a potential issue here—asking an LLM to evaluate its own work can lead to major biases.

If LLMs were perfectly unbiased reasoning agents, this would not be an issue. The fact that an LLM produced the input should not affect an LLM according to the work done by Zhang et al. [65] and Shankar et al. [46]. But something to note is that Yang et al. [29] contradictorily identified that LLM-based evaluators may exhibit bias towards LLM-generated texts. They theorize that LLM-generated text may align more closely with their generation process, leading to a preference for LLM-generated texts. Future work could involve quantifying this and studying this more precisely.

Even if an LLM was not biased towards its own work there is the question of how accurate it is in evaluating produced explanations. Based on work by Lin et al. [26], we theorize that an LLM is just as good as humans in grading produced explanations.

Chapter 7

Conclusion

We presented BugLLM in this work—a new and novel approach to bug localization that leverages semantic searching through embeddings produced by very large general-purpose embedding models and uses LLMs as reasoning agents to filter results and build queries. BugLLM requires minimal setup to work with a new project and needs no training or other computationally expensive setups. We evaluated our approach on the benchmark dataset by Ye et al. [60] and found that BugLLM performs very well for a zero-shot system.

BugLLM also uses LLMs to produce explanations about why it selected specific files. To the best of our knowledge, this is the first bug localization technique to do so. This should help developers fix bugs even faster and make the tool more transparent, and therefore more trustworthy and understandable. We used LLMs to evaluate our explanations on various metrics and found them to be informative and accurate.

Future work could explore further enhancements to the explainability and efficiency of bug localization methods, leveraging advancements in LLMs and embedding models. We could also work on understanding the specific details users would find helpful in the explanations produced for buggy files.

References

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, nov 2009.

[2] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: for the student programs, from the student programs. In *ICSE: SEET*, pages 78–87, 2018.

[3] Agnieszka Ciborowska and Kostadin Damevski. Fast changeset-based bug localization with bert. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 945–956. ACM, 2022.

[4] Richard Connor. A tale of four metrics. pages 210–217, 10 2016.

[5] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Explainable software analytics. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, page 53–56, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Evans Data. Global developer population and demographic study 2023, 2023. Published by Evans Data, August 2023.

[7] Michael Fu and Chakkrit Tantithamthavorn. Gpt2sp: A transformer-based agile story point estimation approach. *IEEE Transactions on Software Engineering*, 49(2):611–625, 2023.

[8] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[9] Ke Shi Jingfei Chang Guangliang Liu, Yang Lu and Xing Wei. Convolutional neural networks-based locating relevant buggy code files for bug reports affected by data imbalance. In *IEEE Access*, pages 131304–131316, 2019.

[10] Jianjun He, Ling Xu, Yuanrui Fan, Zhou Xu, Meng Yan, and Yan Lei. Deep learning based valid bug reports determination and explanation. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 184–194, 2020.

[11] David Huber, Mauro Paltenghi, and Michael Pradel. Where to look when repairing code? comparing the attention of neural models and developers. *arXiv preprint arXiv:2305.07287*, 2023.

[12] Jack Humphreys and Hoa Khanh Dam. An explainable deep model for defect prediction. In *2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 49–55, 2019.

[13] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020.

[14] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John Grundy. Practitioners' perceptions of the goals and visual explanations of defect prediction models, 2021.

[15] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 273–282, New York, NY, USA, 2005. Association for Computing Machinery.

[16] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[17] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. *ICPC*, pages 218–229, 2017.

[18] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[19] Thomas D. LaToza and Brad A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 185–194, New York, NY, USA, 2010. Association for Computing Machinery.

[20] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. The technical debt dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'19, page 2–11, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Hongliang Liang, Dengji Hang, and Xiangyu Li. Modeling function-level interactions for file-level bug localization. *Empirical Software Engineering*, 27, 10 2022.

[22] Hongliang Liang, Dengji Hang, and Xiangyu Li. Modeling function-level interactions for file-level bug localization. *Empirical Softw. Engg.*, 27(7), dec 2022.

[23] Hongliang Liang, Lu Sun, Meilin Wang, and Yuxing Yang. Deep learning with customized abstract syntax tree for bug localization. *IEEE Access*, 7:116309–116320, 2019.

[24] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, 40(6):15–26, jun 2005.

[25] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 15–26, New York, NY, USA, 2005. Association for Computing Machinery.

[26] Luyang Lin, Lingzhi Wang, Jinsong Guo, and Kam-Fai Wong. Investigating bias in llm-based bias detection: Disparities between llms and human perception, 2024.

[27] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, oct 2006.

[28] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

[29] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2511–2522. Association for Computational Linguistics, 2023.

[30] Run LLAMA. Llama index, 2023. Accessed: 2024-07-18.

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

[32] Christopher D Manning. *Introduction to information retrieval*. Syngress Publishing,, 2008.

[33] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[34] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[35] Ahmad Haji Mohammadkhani, Nitin Sai Bommi, Mariem Daboussi, Onkar Sabnis, Chakkrit Tantithamthavorn, and Hadi Hemmati. A systematic literature review of explainable ai for software engineering, 2023.

[36] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, Joanne Jang, Peter Welinder, and Lilian Weng. Text and code embeddings by contrastive pre-training, 2022.

[37] Mauro Paltenghi, Rahul Pandita, Amy Z. Henley, and Andreas Ziegler. Extracting meaningful attention on source code: An empirical study of developer and neural model code exploration. *arXiv preprint arXiv:2210.05506*, 2022.

[38] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. volume 14, pages 1532–1543, 01 2014.

[39] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

[40] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. Pyexplainer: Explaining the predictions of just-in-time defect models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 407–418, 2021.

[41] Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. Scaffle: Bug localization on millions of files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 12, 2020.

[42] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p : $Aneuralprogramcorrectorformoocs. InSPLASHCompanion, pages39 - -40, 2016.$

[43] Mohammad Masudur Rahman, Foutse Khomh, Shamima Yeasmin, and Chanchal K. Roy. The forgotten role of search queries in ir-based bug localization: An empirical study, 2021.

[44] Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 43–52, New York, NY, USA, 2011. Association for Computing Machinery.

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

[46] Shreya Shankar, J. D. Zamfirescu-Pereira, Björn Hartmann, Aditya G. Parameswaran, and Ian Arawjo. Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences, 2024.

[47] Jiao Sun, Q. Vera Liao, Michael Muller, Mayank Agarwal, Stephanie Houde, Kartik Talamadupula, and Justin D. Weisz. Investigating explainability of generative ai for code through scenario-based design. In *Proceedings of the 27th International Conference on Intelligent User Interfaces*, IUI '22, page 212–228, New York, NY, USA, 2022. Association for Computing Machinery.

[48] Chakkrit Tantithamthavorn, Jürgen Cito, Hadi Hemmati, and Satish Chandra. Explainable ai for se: Challenges and future directions. *IEEE Software*, 40(3):29–33, 2023.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[50] Haoyu Wang, Ruirui Li, Haoming Jiang, Jinjin Tian, Zhengyang Wang, Chen Luo, Xianfeng Tang, Monica Cheng, Tuo Zhao, and Jing Gao. Blendfilter: Advancing retrieval-augmented large language models via query generation blending and knowledge filtering, 2024.

[51] Shaohua Wang, Foutse Khomh, and Ying Zou. Improving bug localization using correlations in crash reports. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 247–256, 2013.

[52] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 708–719, 2016.

[53] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Comput. Surv.*, 53(3), jun 2020.

[54] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering*, 48(5):1480–1496, 2022.

[55] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[56] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[57] Yan Xiao, Jacky Keung, Kwabena Bennin, and Qing Mi. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, 105, 08 2018.

[58] Yan Xiao, Jacky Keung, Kwabena E. Bennin, and Qing Mi. Machine translation-based bug localization technique for bridging lexical gap. *Information and Software Technology*, 99:58–61, 2018.

[59] Xingrui Xie, Han Liu, Wenzhe Hou, and Hongbin Huang. A brief survey of vector databases. In *2023 9th International Conference on Big Data and Information Analytics (BigDIA)*, pages 364–371, 2023.

[60] Xin Ye, Razvan Bunescu, and Chang Liu. 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*, FSE 2014, page 689–699, New York, NY, USA, 2014. Association for Computing Machinery.

[61] Gokul Yenduri, Ramalingam M, Chemmalar Selvi G, Supriya Y, Gautam Srivastava, Praveen Kumar Reddy Maddikunta, Deepti Raj G, Rutvij H Jhaveri, Prabadevi B, Weizheng Wang, Athanasios V. Vasilakos, and Thippa Reddy Gadekallu. Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions, 2023.

[62] J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. Association for Computing Machinery.

[63] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. Exploiting code knowledge graph for bug localization via bi-directional attention. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 219–229, New York, NY, USA, 2020. Association for Computing Machinery.

[64] Jinglei Zhang, Rui Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. Exploiting code knowledge graph for bug localization via bi-directional attention. In *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, pages 219–229, 2020.

[65] Xinghua Zhang, Bowen Yu, Haiyang Yu, Yangyu Lv, Tingwen Liu, Fei Huang, Hongbo Xu, and Yongbin Li. Wider and deeper llm networks are fairer llm evaluators. *arXiv preprint arXiv:2308.01862*, 2023.

[66] Yadong Zhang, Shaoguang Mao, Tao Ge, Xun Wang, Adrian de Wynter, Yan Xia, Wenshan Wu, Ting Song, Man Lan, and Furu Wei. Llm as a mastermind: A survey of strategic reasoning with large language models, 2024.

[67] Jian Zhou, Hongyu Zhang, and David Lo. 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*, ICSE '12, page 14–24. IEEE Press, 2012.

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

[69] Ziye Zhu, Hanghang Tong, Yu Wang, and Yun Li. Bl-gan: Semi-supervised bug localization via generative adversarial network. *IEEE Transactions on Knowledge and Data Engineering*, 35(11):11112–11125, 2023.