Fault Localization via Fine-tuning Large Language Models with Mutation Generated Stack Traces

Neetha Jambigi
University of Cologne
Cologne, Germany
njambigi@smail.uni-koeln.de

Bartosz Bogacz,
Moritz Mueller
SAP
Walldorf, Germany
{bartosz.bogacz,
moritz.mueller07}@sap.com

Thomas Bach
SAP
Walldorf, Germany
0000-0002-9993-2814

Michael Felderer
German Aerospace Center
University of Cologne
Cologne, Germany
0000-0003-3818-4442

***Abstract*—Abrupt and unexpected terminations of software are termed as software crashes. They can be challenging to analyze. Finding the root cause requires extensive manual effort and expertise to connect information sources like stack traces, source code, and logs. Typical approaches to fault localization require either test failures or source code. Crashes occurring in production environments, such as that of SAP HANA, provide solely crash logs and stack traces. We present a novel approach to localize faults based only on the stack trace information, by fine-tuning open-source large language models (LLMs). As the number of historic crashes is insufficient to fine-tune LLMs, we augment our dataset by leveraging code mutators to inject synthetic crashes into the code base. By fine-tuning on 64369 crashes resulting from 4.1 million mutations of the HANA code base, we can correctly predict the root cause location of a crash with an accuracy of 66.9% while baselines only achieve 12.6% and 10.6%. We substantiate the generalizability of our approach by evaluating on two additional open-source databases, SQLite and DuckDB, achieving accuracies of 63% and 74%, respectively. Across all our experiments, fine-tuning consistently outperformed prompting non-finetuned LLMs for localizing faults in our datasets.**

***Index Terms*—fault localization, machine learning, large language models, mutation-based testing**

I. Introduction

Fault localization in software engineering entails identifying parts of code that lead to failures [1]. It is a crucial step in software debugging. The effort required to manually locate faults may increase proportionally to the size and complexity of the software projects. Automatically locating the part of the code containing the problem with reasonable effectiveness can significantly accelerate issue resolution.

In the case of SAP HANA, a database management system for enterprise applications developed by SAP [2], [3], the source code consists of about 40 million lines of code and is changed about 200 times a day, making it impossible for a single person to understand all parts and all modifications [4]. The tests in continuous integration infrastructure mostly aim at identifying bugs and possible regressions. However, software crashes occurring in production scenarios violate most of the assumptions in unit and functional tests. The crash, in most cases, is reported in the form of a bug report augmented with logs and stack traces resulting from a crash. In such a scenario, there are no failing test cases or source code elements accessible for automated fault analysis or localization. Even with the availability of source code, it still takes a lot of effort to narrow down exactly the faulty code element. Therefore, an automatic fault localization mechanism to identify the faulty code elements is beneficial.

Stack traces have continually proven to be helpful for fault localization [5]–[7]. Previous works have focused on empirically assessing the importance of stack traces in debugging, fault localization [5]–[8] and their effect on bug resolution times [9]. Stack traces are extensively utilized in addition to other factors like test cases, program code, etc., in automated fault localization [10]. Although manually analyzing stack traces is valuable for diagnosing problems it can be time-consuming. Automating this process improves efficiency by quickly identifying patterns and reducing debugging time and effort. In the bug benchmark Defects4J [11], only 3.33% of the crashes contain crash-triggering test cases [12], which further emphasizes an approach to localize faults based on stack traces.

Fault localization has conventionally been done using criteria, such as failing test cases and their coverage information [13], [14] or by applying specific mutation operators to the original code elements and analyzing the mutants' execution. However, the capabilities of large language models (LLMs) have been increasingly leveraged in the field of software engineering for various tasks, including but not limited to code summarization [15], code generation [16], test case generation [17], automatic program repair [18], or fault localization [7], [19]. Recent approaches to fault localization using LLMs regularly outperform conventional statistical and deep learning-based methods [7], [20]. Unlike classical machine-learning methods, LLMs excel in multi-modal learning with free-form text [21], including program text, stack traces, and SQL queries. They can be adapted to domain-specific tasks using one-shot or few-shot prompting, fine-tuning, and instruction tuning [21]. Effective LLM fine-tuning needs large, task-specific datasets. While authentic crashes from real-world systems are the gold standard to learn from, they are rare with very limited samples per code element, which is not enough to fine-tune LLMs. They may not cover all code elements. Therefore, for fault localization in crashes, mutation-based techniques can generate relevant training data in the form of stack traces of crashes.

In this work, we utilize pre-trained open-source LLMs to localize faults using stack traces. We generate a new dataset by mutating the source code of large database projects and triggering crashes during test suite execution. We make the following contributions:

- A novel approach to localize faults by solely utilizing stack traces.
- An evaluation with different LLMs across multiple codebases that shows the viability and generalizability of our approach.

The paper is structured as follows: Section II presents related work. Section III describes our approach using stack traces and fine-tuning open-source LLMs for fault localization. Section IV outlines our mutation-based process to generate failure data for fine-tuning. Section V presents prediction methods with both fine-tuned and non-fine-tuned LLMs. Section VI details experiments on HANA, SQLite, and DuckDB crashes. Section VII discusses results on synthetic and authentic crashes and the approach's limitations. Section VIII presents future directions, and Section IX concludes.

II. Related Work

A. Statistical and Mutation Based Methods

Statistical methods like Spectrum-based fault localization [22] utilize coverage metrics from failing test cases to assign suspiciousness scores to the elements in code. There are various methods [13], [14] to calculate these metrics. Localizing crashes by leveraging stack traces leads to better efficacy of spectrum-based methods [12]. Mutation-based fault localization methods apply mutations to the program, and the failures from test cases that cover the code path of the mutations are used to localize the faults [23], [24]. Both these methods are impacted by the test suite's comprehensiveness and its ability to detect a wide range of faults. Furthermore, these methods are contingent upon failing test cases, where flakiness can play a role [25]. False positives from these methods can complicate fault localization efforts further.

B. Deep Learning and LLM Based Methods

Learning-to-rank faulty code elements involves scores from other fault localization methods, including spectrum-based methods [26], mutation-based methods, or a hybrid of both [27]. Deep learning-based approaches may further combine multimodal factors like code complexities, test results, and code structures. DeepFL [28] combines scores from the spectrum and mutation-based methods and textual attributes of the code to localize faulty code elements. GRACE [29] encodes the coverage information through graph representation to rank the program elements. FixLocator [30] localizes co-changing faults across the function and statement level using a Graph Convolutional Network encoding program dependencies. DEEPRL4FL [31] find representation for coverage matrix and TRANSFER-FL [32] combines spectrum, mutation-based score, and semantic features from code to localize faults at the statement level. These approaches outperform conventional methods but need substantial data to train the models on. In contrast, pre-trained LLMs possess these abilities out-of-the-box and have better generalizability.

Pre-trained LLMs perform a wide range of software engineering tasks more effectively than traditional methods [33], [34]. LLM-based fault localization in AutoFL [7] outperforms both Spectrum-based [13], [14] and Mutation based [23], [24] methods by iteratively prompting the GPT-4 [35] with a combination of failing test case code, stack traces, and code coverage to localize faulty lines. LLMAO [19] extends an open-source LLM and exploits the naturalness of code for localizing faults based only on source code. Automatic program repair approaches perform fault localization as an intermediate step and leverage LLMs to generate repair. The Inferfix [36] framework proposes program repair utilizing an LLM and localizes fault using static analyzer information. RING [18] considers a suggestion of repair for an appropriate location as fault localization. OpenAI's closed-source models [37], [38] limit exploration to prompting strategies, and can be prohibitively expensive [39].

For proprietary software like HANA, there is insufficient usable failure data for these models to perform fault localization effectively. Most of the existing methods focus on scenarios where more than just the crash information is available and cannot leverage LLMs, especially for closed-source software systems like SAP HANA. Hence, it is valuable to evaluate the approach of fine-tuning open-source models for fault localization.

III. Approach

In this section, we provide an overview of our approach while motivating our choices. Essentially, our approach involves using an LLM fine-tuned with stack traces to predict the name of the function causing the crash. Additionally, this fine-tuned language model, when used in production environments, can learn to map a new crash that partially or fully resembles known instances, and then use this information to localize crashes by interpolating mappings to the nearest code elements.

We present a framework for using mutation-induced crash data to fine-tune LLMs for automated fault localization in software crashes. Authentic crashes that can be collected from real systems are a rather small set. To address the scarcity of real crash data, we propose generating diverse crash scenarios through code mutations based on test coverage. This dataset is then used to fine-tune an LLM, enhancing its ability to identify the code elements responsible for new crashes. Our approach compensates for data limitations and leverages the strengths of LLMs to improve automated fault localization.

A. Mutation of Source Code

Mutation testing relies on mutations to detect real faults and evaluate the robustness of the test suites. Researchers have already examined the effectiveness of mutations in identifying real faults in both developer-written and auto-generated test cases [40]. Additionally, simple mutations are capable of identifying complex failures [41]. Based on these findings, we adopt mutations as a means of inducing program crashes to

collect failure data. When programs are mutated, they execute code in unexpected ways, often violating internal assumptions and following unconventional code paths. An example of a mutation changing the condition in the ternary operator is shown in Fig. 2. The resulting stack traces from mutated code can be considered as an incomplete record of the abnormal program flow, showing only the current path in the execution of the program. If mutated program execution is framed as a function that maps code mutations to eventual crashes, our approach can be understood as a sparse sampling of this mapping and using an LLM as an interpolator to fill in the inverse i.e., crashes to mutations.

B. Stack Traces as Failure Information

Crashes caused by mutations of the source code or otherwise typically result in a stack trace. Stack traces are regarded as one of the important pieces of information available to developers to navigate the code to locate errors [9]. They contain a list of active stack frames during program execution. Each frame represents a function call that is still in progress. The stack trace may or may not contain the method name which may have been the root cause of the failure. The presence or absence of the root causes in the stack traces are symptoms of crash stacks being incomplete descriptions of program execution. They are only a snapshot of the call sequence leading to a crash but may miss the broader sequence of events and interactions that contribute to the issue. Despite this fact, we posit that we can leverage such incomplete information and localize crashes that occur long after an incorrect state is introduced in program execution. Such crashes are typically hard to localize manually.

C. Fine-tuning LLM

Pretrained LLMs encapsulate knowledge from their training data. Decoder-only LLMs [34], [38] consist solely of the decoder part of the transformer architecture and are used for sequence generation tasks in natural language processing. They are adept at completing documents by generating tokens based on contextual information. Fine-tuning adapts the LLMs to domain-specific tasks by further training the LLMs on data mapping useful for the domain. In our case, the context is stack traces and as a completion task, the LLM is expected to generate the function name.

IV. Data

We synthesize artificial data to fine-tune the LLMs to perform fault localization. The data creation process consists of mutating the source code and utilizing the test suites to execute the mutated source code.

Fine-tuning requires a large dataset with clear correspondence between code changes and their resulting stack traces. Defects4J [11] and BugsInPy [42] are popular open-source bug benchmarks containing bugs with fixes in singular files. The problem with using these datasets is that most LLMs are trained on these public datasets, incorporating knowledge from these benchmarks. Their sizes are too limited for fine-tuning. These datasets contain multiple codebases and provide a maximum of 174 bugs per codebase. These benchmarks have bugs that span diverse projects over extended lifetimes containing multiple build configuration changes. Therefore, automating mutation with bug benchmarks has proven to be highly challenging in our initial experiments. Instead to evaluate our approach, in addition to SAP HANA, we mutate two well-known open-source projects, SQLite and DuckDB, which offer quick build and test turnaround times. We synthesize a labeled dataset by artificially injecting faults into a codebase (i.e., mutating it) so that a clear correspondence between fault location and crash stack trace can be established and learned by the LLM.

Name	Operation	Set
Assignment	Replace with symbol from set	= += -= *= /= %=
Number	Increment by value	+255 +1 -1 -255 *(-1)
LineOrder	Swap line with random other	
BooleanAssignment	Replace with symbol from set	= &= |= ^=
Delete	Blank line	
Comparison	Replace with symbol from set	== != <> \<= \>=
Symbol	Replace random other symbol	
Arithmetic	Replace with symbol from set	+ - * / %
IncrementDecrement	Replace with symbol from set	++ –
BooleanArithmetic	Replace with symbol from set	& | ^ << \>>
Logical	Replace with symbol from set	&& and || or != not

Table I: Example of mutators.

A. Synthetic Data Creation

We perform coverage-guided mutations to trigger potential crashes and rely on test suites to execute mutated code as shown in Fig. 1. A coverage-guided mutation is effective only on code that is reachable by existing test cases. However, a limitation of this approach is that faults can exist in both tested and untested code paths, potentially leaving some crashes excluded from our dataset. Generally, more actively used code paths and those with a history of crashes tend to have better test coverage. Regardless, a dataset of crashes generated through source code mutations will generally present faults from a more varied range of code paths compared to datasets based solely on real-world crashes.

B. Mutators

We apply mutations to create variants of the program that are almost identical to the original but contain one minor defect. These mutators cause code-path divergences, resulting in errors of varying complexity. When the mutated method is present in the stack trace of the crash, it can be easier to locate such faults. However, Fig. 2 shows a simple mutation causing a complex failure that crashes outside the mutated method, which does not appear in the stack trace. Such failures may need extensive debugging to locate the faults.

The mutators (methods to perform mutations) implemented in our approach can be divided into three classes (i) simple replacements within a fixed set of symbols, (ii) replacements of symbols found in the function body, (iii) deletion of lines. The sets of symbols used for replacement are given in Table I. These mutators are *Assignment*, *BooleanAssignment*, *Comparison*, *Arithmetic*, *IncrementDecrement*, *BooleanArithmetic*, and *Logical*. Their purpose is to induce execution to diverge

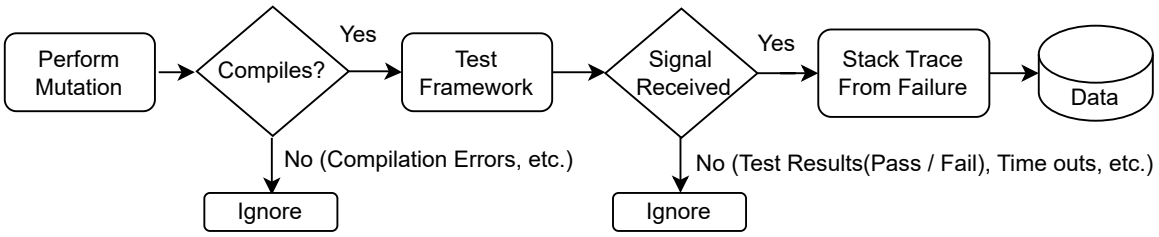


Figure 1: Data creation process.

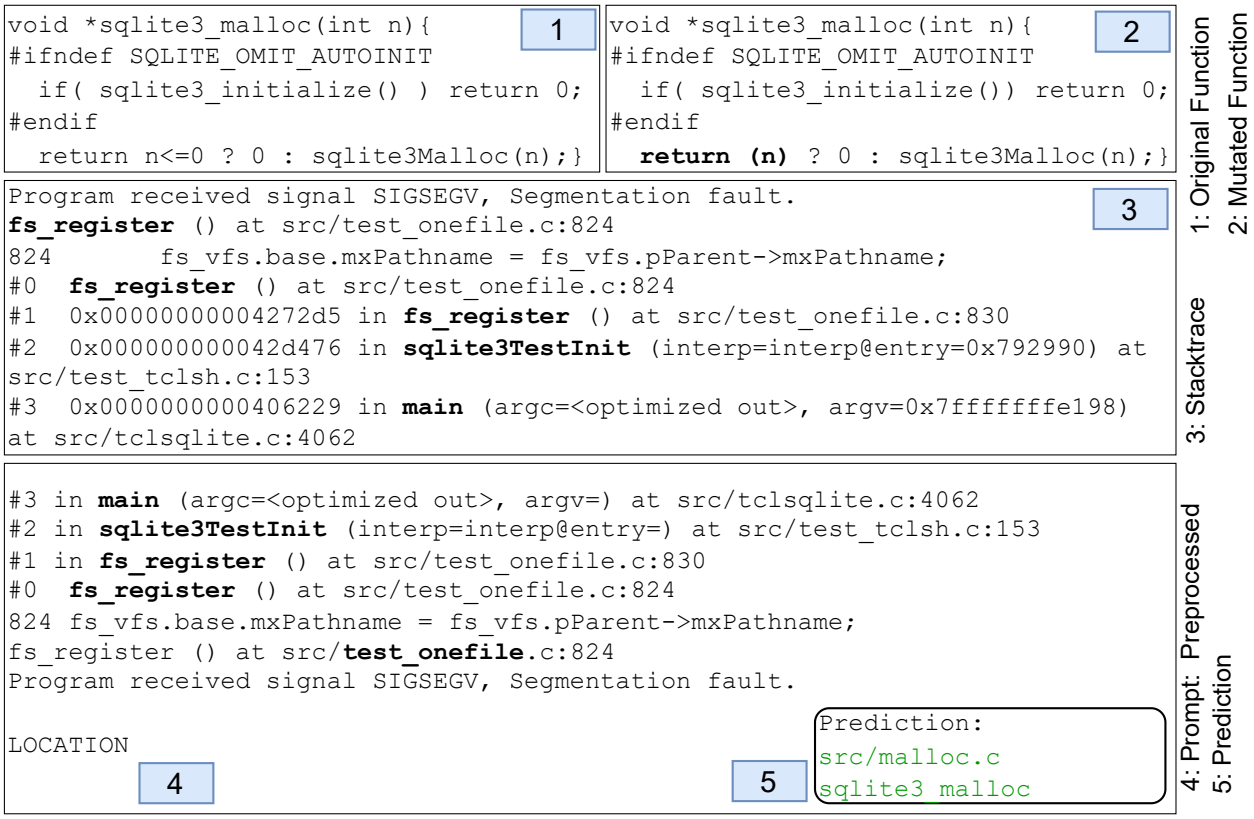


Figure 2: Example of Non-local Crash due to Mutation to *malloc* function in SQLite. This figure outlines mutation, preprocessing stack traces, fine-tuning prompt, label and prediction steps in our pipeline.

into paths where the current program state is invalid, e.g. by perturbing conditionals, and loop increments. We use the Python Tree-sitter[1] library to obtain the abstract syntax tree, identify nodes, and apply precise code mutations.

C. Mutation Execution

We utilize the test suites to execute the mutated source code. Therefore, we select the source code targets for mutation based on test case coverage information. For the selected mutation targets, we apply the set of mutators as described. We present an overview of our mutation process in Fig. 1. Then we build the source code. We discard all build failures. We then execute the code through test frameworks. If the mutation leads to a program receiving a signal like SIGSEGV, resulting in the abrupt termination of the program as illustrated in step 3 of Fig. 2, due to an invalid memory operation, we consider such a scenario as a crash. Only mutations that lead to crashes, defined as the executable receiving a signal from the operating system, are considered for evaluation. Once the mutated executable crashes, a stack trace is extracted with the debugger and recorded into our dataset. Finally, the mutation is rolled back in the source code, and we perform a new mutation.

For mutation, we primarily focus on SAP HANA [2] an enterprise-level ERP database system implemented in C++. Additionally, we choose two open-source database projects: SQLite [43] and DuckDB [44]. DuckDB, developed in C++, is optimized for in-memory analytical queries. SQLite, written in C, is a lightweight, serverless DBMS commonly used in embedded systems.

We enable debug symbols to compile the code for SQLite and DuckDB and run their respective test suites under GNU

[1] https://github.com/tree-sitter/tree-sitter

GDB to collect traces for crashes. HANA has its own internal unwinding framework that provides stack traces. We ignore the mutations that lead to i) an ordinary completion of the shell executable or ii) execution times that exceed a predetermined threshold. We determine the threshold duration for each codebase separately since successful build and test runtimes differ by project. We record the time for a successful build under the specific settings used for each project. This recorded time then is set as the build threshold. As some mutations can cause longer runtimes exceeding the threshold times, we terminate such builds. In the final dataset, the stack traces resulting from the crash are mapped to the mutated method.

D. Types of Crashes

Some crashes occur close to the code being executed, making the faulty function identifiable within the stack trace and relatively easy to localize. However, even in these cases, the faulty function may appear at any depth in the stack trace, adding complexity to localization. More challenging are cases where the faulty method doesn't appear in the stack trace at all, making localization significantly more difficult. Based on these aspects of the complexity of fault localization, we categorize the resulting crashes into two types. i) *Local crashes*: Crashes where the mutated function name is found within the resulting stack trace. ii) *Non-local crashes*: Crashes where the mutated function name is not included in the stack trace.

E. Preprocessing Stack Traces

We choose to fine-tune the open-source LLMs. With the open-source LLMs, we are constrained with respect to the available context length, and preprocessing allows us to retain the most important parts of the stack traces. Step 4 of Fig. 2 illustrates an example of the preprocessed SQLite stack trace. The stack trace shows the active function calls during a system crash and the exact location of the crash as shown in Fig. 2. Stack traces consist of static components like method names, class names, and line numbers, which are useful for identifying faults, and dynamic components like memory addresses, thread identifiers, timestamps, and register values that vary with each execution. We exclude the dynamic components during preprocessing because they do not significantly contribute to fault localization. The frame order can differ based on the collection method. The GNU GDB returns a different order than that of HANA's method. We ensure the most recent call is at the beginning of the stack trace. In the case of HANA, the crash dumps contain a large and dense amount of information on the database's state, runtime information, etc. We extract parts that contain the stack traces of the crash.

After preprocessing, SQLite and HANA retain the full depth of stack traces, while in DuckDB, truncation mostly eliminates frames pertaining to the test execution framework. The preprocessing steps we apply may cause stack traces resulting from different mutation locations to appear identical, thereby reducing the effectiveness of the fine-tuning process. Therefore, we filter the dataset to retain distinct stack traces to function mappings. In the SQLite and DuckDB datasets, the average occurrence of such cases is 1, whereas in the HANA dataset, the average occurrence is 2.37. In HANA, we deduplicate the dataset based on the preprocessed stack traces.

V. Prediction Task

Prompt

ROLE: You are a software engineering assistant who can analyze stack traces from a crash and assist a C developer in localizing the fault to a method in the stack trace.
TASK: Given a stack trace, identify the function name that most likely caused the crash. The faulty method is not necessarily closer to the last frame in the stack trace.
INPUT: Preprocessed stack traces that resulted from a crash in the open-source Database project SQLite, which has been written in C programming language.
[CRASH STACK]:
#3 in main .. at src/tclsqlite.c:4062
#2 in sqlite3TestInit ..
..
fs_register() at src/test_onefile.c:824
Program received signal SIGSEGV, Segmentation fault.
OUTPUT: Strictly provide only the function name without any additional information in the format:
<function_name>

Response

<sqlite3_malloc>

Listing 1: Example Prompt for SQLite and Llama Instruct.

LLMs are trained on publicly available code and related data. They have been used for fault localization and understanding code and failures through prompting techniques. Code-based LLMs are not as effective at comprehending instructions in a prompt as their instruct-tuned versions. Code-based LLMs often generate tokens unrelated to the expected output when not explicitly trained for a particular task as ours. Therefore, predictions can be made using prompting techniques with general-purpose language models like GPT4-o. Alternatively, we can expect an accurate inference from fine-tuned LLMs.

A. Prompting LLMs

We use zero-shot prompting to query faulty methods from instruction-capable LLMs. The general structure of our prompts has a role definition, task description, input description, and output structure constraints. Initially, we define a role for the LLM as an 'assistant' to the developer who is capable of analyzing stack traces of a software crash to identify the faulty method that may have most likely caused the crash. We provide information on the programming language and name of the database project from which the stack trace has been collected. We describe the input information as preprocessed stack traces collected from a crash. We define the output structure and instruct the LLM to strictly adhere to producing a function

name without any additional explanation or sentences. Listing 1 is an example prompt and response for an SQLite crash caused by mutating *malloc* function.

B. Inference with Finetuned LLMs

Our fine-tuning task involves a text prediction task, where the LLM predicts the faulty function name with a file path given a preprocessed stack trace. While fine-tuning an LLM we train the target as a file name with a function name that can comprise multiple tokens. Various methods control token generation, such as setting a maximum token limit or using an end-of-generation token selected during fine-tuning. We set the end-of-generation token to '<|ENDOFTEXT|>'. For the prediction phase, we generate the tokens where the model greedily chooses the next most probable token as completion until it encounters an end-of-generation token.

VI. Experiments and Evaluation

In this section, we discuss our experiment setup, dataset and define the evaluation metric for each experiment.

A. Dataset

1) Synthethic Crash Dataset: The dataset of synthetic crashes in our experiments is presented in Table II. The types of mutations applicable vary depending on the programming language the project is implemented in. Therefore, we implement custom mutators that are applicable to specific libraries in HANA. The types of mutations across SQLite and DuckDB remain the same. We do not balance the synthetic dataset w.r.t the function names or the mutation types. Although the process of mutating source code may be influenced by the size of the target functions in terms of lines of code, our resulting dataset includes a large number of unique target functions and shows no significant bias toward specific targets. We deduplicate the dataset before splitting it into training and validation sets, ensuring no data leakage. The data is randomly split into a 90% training set for fine-tuning the model and the remaining 10% as a validation set. We use the same validation dataset across all our experiments pertaining to synthetic crashes.

	Training			Validation			Authentic		
Project	#C	#T	%L	#C	#T	%L	#C	#T	%L
HANA	64369	8609	33.1	7152	3123	32.4	175	82	100
SQLite	13299	1183	47	1477	569	46	-	-	-
DuckDB	1635	345	55.8	181	120	57	-	-	-

Table II: #C: count of crashes #T: unique targets - file path with function names, %L: percentage of local crashes in data

2) Authentic Crash Dataset: Over several years, we have collected data on crashes in HANA. For these crashes, fixes are often complex. Fixes may involve changes to multiple files and functions unrelated to the root cause of the failure. Additionally, changes to code over a decade have resulted in methods that were changed as a part of a fix but no longer exist. To ensure our approach has even a chance of predicting the correct locations, we significantly trim the authentic crashes to retain only crashes where fixes entail changes to a single file but multiple methods, and one of the fixed methods is present in the stack trace. Consequently, we have 175 authentic local crashes spanning five years.

B. Prediction Task

The task is to predict the faulty method causing the crash.

1) On Synthetic Crashes: In fine-tuned models, the prediction must accurately match the file name and the method name. We introduce this output structure to pre-trained LLMs through fine-tuning. In non-fine-tuned LLMs, the models must predict only the function name.

2) On Authentic Crashes: Authentic crashes have complex fixes, which consist of modifications to n different methods. If at least one of the n methods is predicted as a fault, we consider it accurately localized.

C. Models and Setup

1) Non-fine-tuned LLMs: Non-fine-tuned LLMs may not recognize the necessary file names or understand the required output structure. Additionally, non-instruct versions of LLMs typically are not designed for comprehending instructions in prompts, where we define a task and desired output format. Therefore, we choose to use the instruct variations of LLMs for our non-fine-tuned LLM experiments - specifically, Mistral-7b-Instruct (Mistral-I) [45], LLAMA-70B-Instruct (Llama-I) [46], and GPT-4o[2], one of the best LLMs across several tasks. Mistral-I outperforms Llama-I across several benchmarks[3], Llama-I is an instruct model well suited for coding tasks in the Llama family[4].

Non-fine-tuned models trained on public datasets may possess an inherent understanding of stack traces and may be inclined to pick a function name from within the stack trace. We noticed this pattern with our manual experiments with prompting. Therefore, we modify our prediction task to identify the faulty function name and also limit our dataset to the *local crashes* where the faulty method is present in the stack trace. We consider a target accurately predicted when the LLM picks the right function name from the input stack trace. Every LLM may comprehend instructions differently and require separate prompt optimization across each codebase to obtain the desired output. Therefore, we manually optimize the prompt for each LLM and project by examining 10 sample cases. We set the temperature to 0 across all these models.

2) Fine-tuned LLMs: We conduct experiments with three open source LLMs: Mistral-7B [47], Santacoder-1B [48], and CodeLlama-7B [49]. Santacoder-1B has not been trained on C and C++ code or failures. However, CodeLlama and Mistral may have been trained with C and C++ code. Santacoder-1B and CodeLlama-7B are particularly suited for code-related tasks, and Mistral-7B is capable of comprehending instructions and performs suitably well on code-related tasks. We fine-tune CodeLlama-7B and Mistral-7B with LoRA [50] adapters which

[2]https://openai.com/index/hello-gpt-4o/
[3]https://mistral.ai/news/mixtral-of-experts/
[4]https://deepinfra.com/meta-llama/Llama-2-70b-chat-hf

results in training only about 1% of the full model parameters during fine-tuning. We limit the input to a maximum of 1024 tokens. This token limit is sufficient to fit the required tokens from pre-processed stack traces across all our models, ensuring uniform data across experiments for model comparison.

Baselines: i) Innerframe: A method that chooses the *'innermost frame'* in the stack trace. In a sequence of calls, it is likely the most relevant fault location. ii) fasttext with cosine similarity: We train a *fasttext* [51] embedding model and vectorize the pre-processed stack traces to predict function name using 1-nearest-neighbor(*1-NN*) classifier using cosine similarity. iii) Non-fine-tuned instruction capable LLMs mentioned in Section VI-C1.

D. Metrics

1) Perfect Match / Accuracy: We present the *Accuracy* of the models as our evaluation metric. For fine-tuned LLMs, we consider a target accurately predicted when the predicted tokens, excluding the end-of-generation, perfectly match the target tokens. For non-fine-tuned LLMs, if the model picks the right function from within the stack trace, we consider it accurately predicted.

2) Average Precision: We choose average precision as a metric to assess an LLM for partially identifying the fault location. We consider that correctly locating the right component, class, or file can be helpful in pinpointing faulty code. Therefore, we evaluate the similarity between the predicted and target strings. To determine the precision, we convert both the prediction and target strings into individual terms. For example, in C++ target string `'X/Y/Z.cpp A::B::C'` becomes [X, Y, Z, A, B, C]. We disregard the order of terms and compute the percentage of terms in the prediction that match terms in the target. This is similar to calculating the BLEU-1 score [52] without the brevity penalty. The average precision of the terms produced by the LLM across all instances is calculated as shown in Eq. (1).

Average Precision (AP), is calculated as:

$$\text{AP} = \frac{1}{N}\sum_{i=1}^{N}\frac{|\text{terms}(P_i) \cap \text{terms}(T_i)|}{|\text{terms}\ (P_i)|} \tag{1}$$

where P_i and T_i are the Prediction and Target of the ith instance, respectively. N is the total number of predictions.

VII. Discussion

In this section, we discuss the results of fault localization across fine-tuning LLMs and prompting instruct capable LLMs.

A. Fault Localization with Fine-tuned LLMs

1) Performance on Synthetic Crashes: Table III presents an overview of the overall performance for all the fine-tuned LLMs and the baselines *fasttext* and *Innermost* across all the codebases. While fine-tuning Santacoder-1B on HANA crashes, we observed evolving validation set accuracies as seen in Fig. 3. Continued training improved the accuracy for non-local crashes, but these remain more challenging compared to local crashes. This trend is consistent across all our projects and LLM combinations. As shown in Table III, non-local crashes consistently have lower accuracies than local crashes.

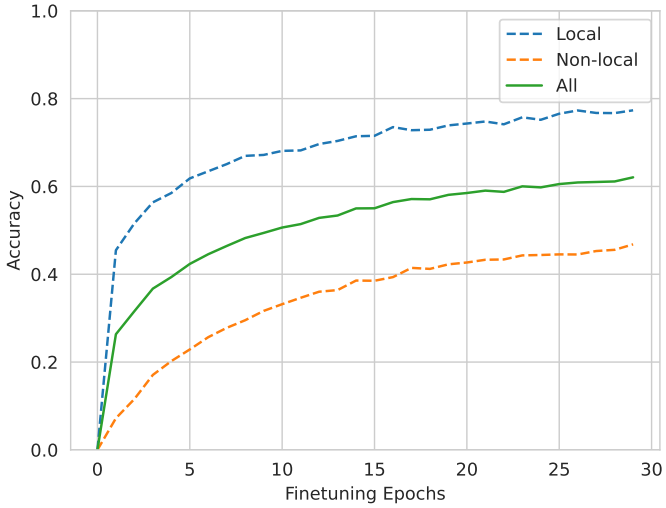


Figure 3: HANA Validation Accuracy on Santacoder-1B

In our experiments, we observed no correlation between the depth at which the mutated function name is present and the LLMs' effectiveness in localizing faults within the validation data. To determine if the LLM relies heavily on learning associations with function names present in the stack trace, we analyzed the misclassifications to identify if the predicted function names were included in the prompt. However, this was not the case. All LLMs tended to make predictions without a particular bias towards function names from the stack trace, across both local and non-local crash misclassifications. We also found no correlation between the length of stack traces and the accuracy of fault localization across all our experiments with fine-tuned models. We observed that LLMs generated accurate file paths for some of the misclassified instances. As shown in Fig. 2, LLMs are required to generate both the file path and function names. Therefore, when considering file-level localization, we observed an average increase in overall localization accuracy by 5% in DuckB (to 62%), 34% in SQLite (to 89%), and 10% in HANA (to 76%) across all LLMs. This improvement can be due to the ease of a coarse-grained task of predicting file names.

The LLMs we fine-tuned are of different sizes, pre-trained on different data, and possess different inherent capabilities. For instance, Santacoder-1B is not inherently aware of C and C++ style code and stack traces. In our experiments with HANA and SQLite on fine-tuned Santacoder-1B, a model with just 1 billion parameters, we observe that it performs on par with larger models like CodeLlama-7B and Mistral-7B. We can partly attribute the comparatively better performance across HANA and SQLite to the relatively larger dataset we were able to gather for fine-tuning.

2) Perfomance of fasttext: The fasttext baseline identifies the root cause functions on par with the LLMs across SQLite and DuckDB synthetic crashes. However, this approach has

Models			Accuracy												
			Santacoder-1B			Mistral-7B			CodeLlama-7B			Fasttext Cosine			InnerM
Project	Count	Max Depth	Local	Non Local	Total	Local	Non Local	Total	Local	Non Local	Total	Local	Non Local	Total	Total
SQLite-Syn	1082	13	0.758	0.375	0.557	0.832	0.422	**0.634**	0.744	0.271	0.515	0.558	0.434	0.558	0.192
DuckDB-Syn	181	7	0.597	0.261	0.453	0.854	0.587	**0.741**	0.725	0.283	0.537	0.709	0.565	0.629	0.204
HANA-Syn	7152	59	0.795	0.489	0.588	0.884	0.545	0.655	0.879	0.568	**0.669**	0.133	0.093	0.106	0.126
HANA-Auth	175	104	0.354	-	**0.354**	0.297	-	0.297	0.297	-	0.297	0.00	-	0.00	0.185

Table III: Accuracy of predicting file and faulty function name on Validation Data - Baseline and fine-tuned LLM (Auth: Authentic crashes, Syn: Synthetic crashes, InnerM: InnerMost Baseline)

		#Correct Predictions on UT		
Project	#UT	Santacoder-1B	Mistral-7B	CodeLlama-7B
SQLite-Syn	25	4	2	2
DuckDB-Syn	14	1	4	0

Table IV: Correct predictions for targets that were not present in the training set. UT: Unseen Targets during fine-tuning

a low accuracy of 10% on HANA synthetic crashes and is unable to localize any of the authentic HANA crashes despite having a complete overlap of the target functions of the authentic crashes with the training set. *fasttext* primarily treats text as a bag-of-words, disregarding its sequential structure. Hence it doesn't effectively capture long-range dependencies. LLMs, being transformer-based architectures, excel at learning representations for long sequences, effectively preserving the structural nuances of stack traces [53]. As shown in Table III, the performance disparity between the LLMs and fasttext on HANA highlights the influence of stack trace structure on HANA compared to SQLite and DuckDB.

3) Unseen Targets: Random sampling without stratification for targets resulted in validation sets for DuckDB and SQLite with stack traces of certain mutated functions and, consequently, the function names are not present in the training set. As shown in Table IV, LLMs predicted some of such unseen targets correctly. Our approach benefits from the generative aspect of LLMs being capable of abstracting inherent patterns and interpolating unseen but correct locations, as shown in the case of SQLite and DuckDB.

4) Performance on Authentic Crashes: Even with a limited set of authentic data, we show in Table III that LLMs fine-tuned on synthetic crash data can localize authentic bugs with an accuracy of 35.4%. Despite relatively weaker performance on synthetic crashes, Santacoder-1B outperforms CodeLlama-7B and Mistral-7B on the authentic crash dataset. This could be an indication of larger models overfitting our synthetic crash dataset. Fine-tuned LLMs produced more useful information on average across all experiments in comparison to prompting non-fine-tuned LLMs as indicated by average precision in Table VII.

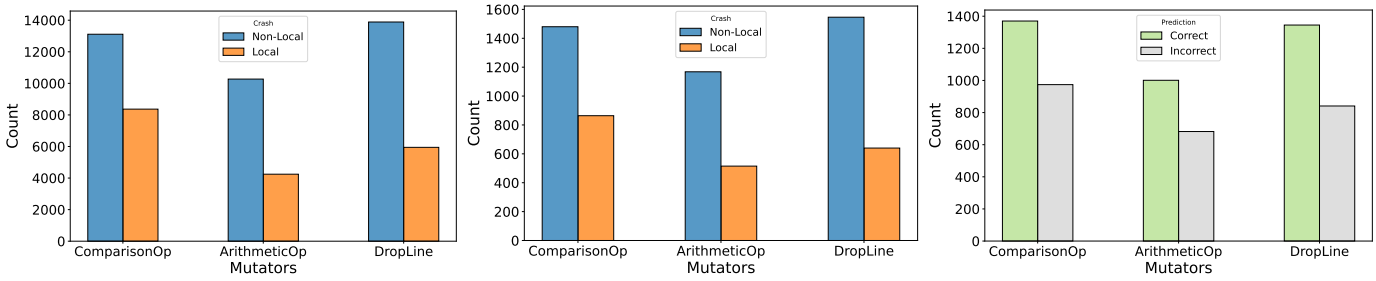
B. Analysis across Mutation Operators

We analyze the crashes caused by different mutators and evaluate the accuracy of localizing them. Our data shows that certain mutators cause more crashes and complex failures than others. In the HANA dataset, the distribution of crashes from different mutation operators is consistent across training and test splits as shown in Fig. 4. The top 3 mutators result in 86.8% of all crashes and 85% of wrongly localized crashes in the HANA dataset. 78% non-local crashes of comparison mutators are misclassified in HANA. Research indicates that Drop Line mutants often reflect real faults [40]. Drop Line operator constitutes 30% of all crashes in the HANA dataset. 72% of the Drop Line mutation crashes are non-local crashes. In HANA, 52% of the wrong predictions belong to non-local crashes of the Drop Line operator. We found that mutations to lines of code containing assignments, method calls, and pointer operations resulted in more non-local crashes than other types of code. Similarly, in both SQLite and DuckDB datasets, Drop Line and comparison mutators lead to the most non-local crashes. Across SQLite and DuckDB datasets, non-local crashes resulting from Drop Line have an accuracy of 0%, whereas local crashes are localized with an accuracy of 100%. Across all the other mutators, we do not see a distinct pattern of misclassification in our datasets. DropLine and comparison operators induce complex failures and stack traces alone may be insufficient to localize such faults.

C. Fault Localization with Prompting Non-fine-tuned LLMs

We evaluate the performance of the Instruct LLMs for localizing faults i) Based only on stack traces, ii) Based on stack traces augmented with function implementations, and iii) Based on obfuscated stack traces. We also evaluate them for the usefulness of their output with an average precision.

1) Prompts with only Stack Traces: We prompt the LLMs as shown in Listing 1 with preprocessed stack traces. Llama-I outperformed Mistral-I and GPT-4o on the HANA dataset of local crashes. Across DuckDB Mistral has 12% while the other LLMs have 0%. Despite considering only local crashes, the accuracy across all the LLMs and projects is consistently poor as shown in Table V in comparison to their fine-tuned counterparts. Table VI shows the average depths at which the LLMs selected the faulty method. If the LLM chose the first frame, the depth would be considered as 0, and if it picked



(a) Training data (b) Test data (c) Accuracy with Santacoder-1B

Figure 4: Top 3 Mutator distribution in Synthetic HANA Dataset

		Accuracy Non-Fine-tuned			Accuracy Fine-tuned		
Models		GPT-4o	Mistral-I	Llama-I	Santacoder-1B	Mistral-7B	CodeLlama-7B
Project	Count	Local			Local		
SQLite-Syn	680	**0.567**	0.357	0.496	0.758	**0.832**	0.744
DuckDB-Syn	103	0.0	**0.161**	0.0	0.597	**0.854**	0.725
HANA-Syn	2317	0.212	0.241	**0.422**	0.795	**0.884**	0.879
HANA-Auth	175	0.091	0.12	**0.229**	**0.354**	0.297	0.297
HANA-Syn-FD	1469	0.094	-	-	-	-	-

Table V: Accuracy of Non-fine-tuned and Fine-tuned LLMs on local crashes. Only Local crashes are used for this experiment. (Auth: Authentic crashes, Syn: Synthetic crashes, FD: Prompts augmented with function source code)

the last, then the depth would be 1. We exclude instances where the LLMs returned unparseable outputs for average depth calculation. The stack traces include functions from the test framework and exception-handling frameworks that may have caught the failures. In SAP HANA, such frames tend to be at the beginning and, at times, also at the end of the stack traces. In DuckDB and SQLite, such frames tend to be at the beginning. In most cases, LLMs bypassed such tests and stack unwinding related frames. We analyzed how the length of stack traces affects accuracy and found no correlation between them.

	Avg. Stack Depth of Prediction					
Models	GPT-4o		Mistral-I		Llama-I	
Project	Local	Non Local	Local	Non Local	Local	Non Local
SQLite-Syn	0.93	0.91	0.15	0.01	0.27	0.28
DuckDB-Syn	0.93	0.93	0.27	0.21	0.27	0.23
HANA-Syn	0.49	0.50	0.47	0.51	0.37	0.42

Table VI: Average depth at which Non-fine-tuned LLMs predicted the faulty function from the input stack traces. (Syn: Synthetic crashes)

2) Prompts Augmented with Function Source-code: We assessed if adding function implementation code to pre-processed stack traces could improve performance, as SAP HANA is a closed-source system. Function implementations may provide useful comments that enhance the model's understanding. We ignore the function implementations of unit tests and stack unwinding frameworks. We retain 1469 local and 234 non-local crashes where we could reliably extract all important function definitions. Since adding function implementations to the prompt requires longer context lengths than the context length supported by Mistral-I and Llama-I, we only conducted these experiments with GPT-4o. GPT-4o accurately localized only 9.4% of the faults in local crashes as shown in HANA-Syn-FD Table V. This approach requires accurate source code elements and performs worse than prompts with only stack traces. With the average precision, GPT-4o provided useful information for 720 local crashes as shown in Table VII, which is worse than only using the stack traces approach. Overall, adding the function implementations did not aid in fault localization.

3) Prompts with Stack Trace Obfuscation: We have seen in our experiments that LLMs understand the basic structure of a stack trace. However, we wanted to investigate if LLMs derive meaning from human-readable elements, specifically method names within the stack trace. To test this, we obfuscated stack traces by using the SHA-256 [54] algorithm to hash each line. We informed the LLMs that the stack trace content had been hashed. We tested GPT-4o on 10 SAP HANA local crash examples with these obfuscated stack traces. GPT-4o consistently identified the most frequently repeating line as the culprit method for the crash and provided the same explanation each time. This behavior remained the same even when we separated and hashed each term within the lines individually while keeping the overall line structure. The accuracy of predictions was 0% across both these experiments. While with unobfuscated prompts the LLM was able to predict the locations correctly. This suggests that LLMs derive information from human-readable and meaningful terms in stack traces.

4) Performance on Authentic Crashes: Llama-I outperformed all other non-fine-tuned models and accurately local-

Models		Avg. Precision Non Fine-tuned %						Avg. Precision Fine-tuned %					
		GPT-4o		Llama-I		Mistral-I		Santacoder-1B		CodeLlama-7B		Mistral-7B	
Project	Count	Local	Non Local	Local	Non Local	Local	Non Local	Local	Non Local	Local	Non Local	Local	Non Local
SQLite-Syn	1082	**0.566**	0.0	0.495	0.0	0.351	0.0	0.894	0.705	0.881	0.674	**0.936**	**0.761**
DuckDB-Syn	108	**0.427**	0.014	0.308	0.014	0.278	**0.018**	0.851	0.699	0.881	0.711	**0.922**	**0.829**
HANA-Syn	7152	0.37	0.42	**0.642**	**0.450**	0.379	0.429	0.924	0.759	0.940	0.802	0.959	0.792
HANA-Auth	175	0.308	-	**0.534**	-	0.302	-	**0.640**	-	0.612	-	0.610	-
HANA-Syn-FD	1469	0.49	0.37	-	-	-	-		-	-	-	-	-

Table VII: Average precision of prediction measuring the usefulness of the predictions across fine-tuned and non-fine-tuned LLMs. (Auth: Authentic crashes, Syn: Synthetic crashes, FD: Prompts augmented with function source code)

ized 22.9% of our authentic crash dataset, while GPT-4o and Mistral-I were at 9.1% and 12%, respectively. However, non-fine-tuned models do not match the performance of any of the fine-tuned models across all our experiments as seen in Table V with an accuracy of 35.4%.

D. Evaluation with Average Precision

Average Precision, as described in Section VI-D2, evaluates the LLM's ability to match terms within the target function name. This allows us to measure performance for both local and non-local crashes for non-fine-tuned models. Even though the accuracy across DuckDB is 0% for both GPT-4o and Llama-I as presented in Table V, the models were able to correctly predict parts of the fault location, as indicated by their average precision match in Table VII. Fine-tuned LLMs consistently achieve higher average precision than non-fine-tuned models across all experiments.

E. Limitations

Our data generation process uses simple mutations, leading to a dataset biased towards less complex failures. Additionally, our authentic dataset of crashes is filtered to include only changes to a single function and function names present in the stack trace, biasing the real-world evaluation towards easier cases. As a result, our evaluation outcomes may be an overestimation of the efficacy of our approach.

Certain concerns come with a supervised learning setting, like fine-tuning. Models are prone to overfitting and may become invalid as the source code evolves. Therefore, we may need to create new data to retrain and adapt the fine-tuned models periodically.

VIII. Future Work

By exploring the inner workings of open-source models' attention mechanisms, we can understand how models link tokens in stack traces to predictions. Preprocessing stack traces and performing ablation studies would allow us to identify key elements crucial for fault localization. Incorporating log information and leveraging the multimodal aspects of LLMs can enhance this approach. Additionally, a pretraining step on the entire project code base before fine-tuning can further improve the model's effectiveness. With this research we aim to develop a general approach to handle stack traces across different programming languages, enhancing the adaptability of LLMs. Given their capability in code generation, effective fault localization naturally extends to automatic program repair.

We currently have made fault localization available to SAP HANA developers. They can provide the stack traces of crashes as input to the tool, and we provide the faulty method as a prediction. However, we are still in the early process of gathering feedback and optimizing our service. Therefore, we cannot yet derive conclusions about the performance of our solution in day-to-day work.

IX. Conclusion

There are failure scenarios where stack traces are the sole information available for localizing faults. We use mutations as a data generation process and fine-tune LLMs to model this cause and effect on program execution. Despite stack traces being incomplete records of failures, we show the viability and generalizability of exploiting them for fault localization by evaluating our approach across multiple LLMs and multiple code bases developed in different programming languages.

Refining and chaining prompts may serve as an alternative to improving the performance of instruct-capable LLMs. However, even when the information is present in the input, the LLMs are not able to clearly associate it with the source of failure. Our experiments with obfuscation indicate dependency on the inherent meaning of the human-readable information in the stack traces. Additionally, regardless of the prompting strategy, the underlying LLMs never learn the domain-specific knowledge required for our purpose. While fine-tuning involves creating data and training an LLM, it produces more consistent results as it derives from the patterns of mapping between a failure and the fault location learned during the training phase.

References

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016. (Cited on page: 1)

[2] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database–An Architecture Overview." *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 28–33, 2012. (Cited on pages: 1 and 4)

[3] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: data management for modern business applications," *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012. (Cited on page: 1)

[4] T. Bach, A. Andrzejak, C. Seo, C. Bierstedt, C. Lemke, D. Ritter, D. W. Hwang, E. Sheshi, F. Schabernack, F. Renkes *et al.*, "Testing very large database management systems: The case of SAP HANA," *Datenbank-Spektrum*, vol. 22, no. 3, pp. 195–215, 2022. (Cited on page: 1)

[5] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *2014 IEEE international Conf. on software maintenance and evolution*. IEEE, 2014, pp. 181–190. (Cited on page: 1)

[6] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for Java runtime exceptions," in *Proc.gs of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 153–164. (Cited on page: 1)

[7] S. Kang, G. An, and S. Yoo, "A Preliminary Evaluation of LLM-Based Fault Localization," 2023. (Cited on pages: 1 and 2)

[8] L. Gong, H. Zhang, H. Seo, and S. Kim, "Locating crashing faults based on crash stack traces," *arXiv preprint arXiv:1404.4100*, 2014. (Cited on page: 1)

[9] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do Stack Traces Help Developers Fix Bugs?" in *2010 7th IEEE Working Conf. on Mining software repositories (MSR 2010)*. IEEE, 2010, pp. 118–121. (Cited on pages: 1 and 3)

[10] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE TSE*, vol. 47, no. 2, pp. 332–347, 2019. (Cited on page: 1)

[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. of the 2014 ISSTA*, 2014, pp. 437–440. (Cited on pages: 1 and 3)

[12] L. Barreto Simedo Pacheco, A. R. Chen, J. Yang *et al.*, "Leveraging Stack Traces for Spectrum-based Fault Localization in the Absence of Failing Tests," *arXiv e-prints*, pp. arXiv–2405, 2024. (Cited on pages: 1 and 2)

[13] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proc.of the 20th IEEE/ACM International Conf. on ASE*, 2005, pp. 273–282. (Cited on pages: 1 and 2)

[14] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013. (Cited on pages: 1 and 2)

[15] T. Ahmed and P. Devanbu, "Few-shot training LLMs for project-specific code-summarization," in *Proc. of the 37th IEEE/ACM International Conf. on ASE*, 2022, pp. 1–5. (Cited on page: 1)

[16] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of GitHub copilot's code generation," in *Proc. of the 18th International Conf. on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 62–71. (Cited on page: 1)

[17] V. Guilherme and A. Vincenzi, "An initial investigation of ChatGPT unit test generation capability," in *8th Brazilian Symposium on Systematic and Automated Software Testing*, 2023, pp. 15–24. (Cited on page: 1)

[18] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, "Repair is nearly generation: Multilingual program repair with LLMs," in *Proc. of the AAAI Conf. on Artificial Intelligence*, vol. 37, no. 4, 2023, pp. 5131–5140. (Cited on pages: 1 and 2)

[19] A. Z. Yang, C. Le Goues, R. Martins, and V. J. Hellendoorn, "Large Language Models for Test-Free Fault Localization," 2024. (Cited on pages: 1 and 2)

[20] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large Language Models in Fault Localisation," *arXiv preprint arXiv:2308.15276*, 2023. (Cited on page: 1)

[21] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of Large Language Models," *arXiv preprint arXiv:2303.18223*, 2023. (Cited on page: 1)

[22] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *arXiv preprint arXiv:1607.04347*, 2016. (Cited on page: 2)

[23] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh ICST*. IEEE, 2014, pp. 153–162. (Cited on page: 2)

[24] M. Papadakis and Y. Le Traon, "Metallaxis-FL: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015. (Cited on page: 2)

[25] Q. I. Sarhan and Á. Beszédes, "A survey of challenges in spectrum-based software fault localization," *IEEE Access*, vol. 10, pp. 10 618–10 639, 2022. (Cited on page: 2)

[26] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *2014 IEEE ICSME*. IEEE, 2014, pp. 191–200. (Cited on page: 2)

[27] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proc.of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017. (Cited on page: 2)

[28] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proc.of the 28th ACM SIGSOFT ISSTA*, 2019, pp. 169–180. (Cited on page: 2)

[29] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*, 2021, pp. 664–676. (Cited on page: 2)

[30] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization to detect co-change fixing locations," in *Proc. of the 30th ACM Joint European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*, 2022, pp. 659–671. (Cited on page: 2)

[31] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd ICSE*. IEEE, 2021, pp. 661–673. (Cited on page: 2)

[32] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proc. of the 44th ICSE*, 2022, pp. 1169–1180. (Cited on page: 2)

[33] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large Language Models for Software Engineering: A Systematic Literature Review," 2023. (Cited on page: 2)

[34] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large Language Models for Software Engineering: Survey and Open Problems," *arXiv preprint arXiv:2310.03533*, 2023. (Cited on pages: 2 and 3)

[35] OpenAI, "GPT-4 technical report," *ArXiv*, vol. abs/2303.08774, 2023. [Online]. Available: https://arxiv.org/abs/2303.08774 (Cited on page: 2)

[36] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "InferFix: End-to-end program repair with LLMs," *arXiv preprint arXiv:2303.07263*, 2023. (Cited on page: 2)

[37] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with GPT-4," *arXiv preprint arXiv:2303.12712*, 2023. (Cited on page: 2)

[38] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. (Cited on pages: 2 and 3)

[39] Y. Deldjoo, "Fairness of ChatGPT and the role of explainable-guided prompts," *arXiv preprint arXiv:2307.11761*, 2023. (Cited on page: 2)

[40] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665. (Cited on pages: 2 and 8)

[41] G. Gay and A. Salahirad, "How Closely are Common Mutation Operators Coupled to Real Faults?" in *2023 IEEE Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 129–140. (Cited on page: 2)

[42] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, "BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies," in *Proc. of the 28th ACM joint meeting on european software engineering Conf. and symposium on the foundations of software engineering*, 2020, pp. 1556–1560. (Cited on page: 3)

[43] D. R. Hipp, "SQLite - A Small, Fast, Self-Contained, High-Reliability, Full-Featured, SQL Database Engine," *ACM SIGMOD Record*, vol. 29, no. 2, pp. 1–12, 2000. [Online]. Available: https://dl.acm.org/doi/10.1145/354805.354806 (Cited on page: 4)

[44] M. Raasveldt and H. Mühleisen, "Duckdb: An embeddable analytical database," in *Proc. of the 2019 International Conf. on Management of Data*, 2019, pp. 1981–1984. (Cited on page: 4)

[45] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024. (Cited on page: 6)

[46] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023. (Cited on page: 6)

[47] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023. (Cited on page: 6)

[48] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey *et al.*, "Santacoder: don't reach for the stars!" *arXiv preprint arXiv:2301.03988*, 2023. (Cited on page: 6)

[49] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023. (Cited on page: 6)

[50] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021. (Cited on page: 6)

[51] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017. (Cited on page: 7)

[52] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2002, pp. 311–318. (Cited on page: 7)

[53] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017. (Cited on page: 8)

[54] F. Pub, "Secure hash standard (shs)," *Fips pub*, vol. 180, no. 4, 2012. (Cited on page: 9)