Finding the Needle in the Crash Stack: Industrial-Scale Crash Root Cause Localization with AutoCrashFL

Sungmin Kang
sungmin@nus.edu.sg
NUS
Singapore, Singapore

Sumi Yun
Jingun Hong
sumi.yun@sap.com
jingun.hong@sap.com
SAP Labs Korea
Seoul, Republic of Korea

Shin Yoo
shin.yoo@kaist.ac.kr
KAIST
Daejeon, Republic of Korea

Gabin An*
gabin_an@korea.ac.kr
Korea University
Seoul, Republic of Korea

Abstract

Fault Localization (FL) aims to identify root causes of program failures. FL typically targets failures observed from test executions, and as such, often involves dynamic analyses to improve accuracy, such as coverage profiling or mutation testing. However, for large industrial software, measuring coverage for every execution is prohibitively expensive, making the use of such techniques difficult. To address these issues and apply FL in an industrial setting, this paper proposes AutoCrashFL, an LLM agent for the localization of crashes that only requires the crashdump from the Program Under Test (PUT) and access to the repository of the corresponding source code. We evaluate AutoCrashFL against real-world crashes of SAP HANA, an industrial software project consisting of more than 35 million lines of code. Experiments reveal that AutoCrashFL is more effective in localization, as it identified 30% crashes at the top, compared to 17% achieved by the baseline. Through thorough analysis, we find that AutoCrashFL has attractive practical properties: it is relatively more effective for complex bugs, and it can indicate confidence in its results. Overall, these results show the practicality of LLM agent deployment on an industrial scale.

CCS Concepts

• **Software and its engineering → Maintaining software**; **Software testing and debugging**.

Keywords

Fault Localization, Crash Debugging, Large Language Models, Automated Debugging, Industrial Software, Root Cause Analysis

1 Introduction

Like all software systems, enterprise applications are susceptible to bugs and crashes that demand timely resolution. A crucial step in the debugging process is Fault Localization (FL) [32], i.e., identifying the code responsible for a failure. Empirical studies show developers spend a significant portion of their debugging time on this task [2]. To alleviate this burden, researchers have proposed a variety of automated FL techniques, including spectrum-based fault localization (SBFL) [12, 31] and mutation-based fault localization (MBFL) [22, 23].

More recently, LLM-based fault localization techniques have been proposed [14, 25, 36]. These approaches demonstrate strong performance on open-source projects using failing test cases or bug reports. LLM-based FL offers greater flexibility than traditional FL

---

*Corresponding Author

methods: unlike SBFL, which requires expensive test coverage, or MBFL, which involves running numerous program variants, LLM-based approaches can operate directly on textual descriptions of failure symptoms. This enables integration with lightweight tools (e.g., file content retrieval) and makes them especially appealing in scenarios where runtime instrumentation is infeasible.

We set out to answer a key open question in this paper: to what extent do recent advances in LLM-based FL translate into practically usable tools that can help developers in real-world debugging scenarios? One such scenario is debugging crashes, which represents a common and high-priority failure mode in enterprise software systems. Particularly in industrial contexts, crashes often lack reproducing inputs due to client privacy or operational constraints. Instead, often the only available information is the *crashdump*, which captures the program state at the time of failure. Even the coverage information of the crashed execution is often absent in production environments, due to the performance overhead of instrumentation. These limitations severely constrain conventional FL tools such as SBFL and MBFL, which require dynamic analyses and are inapplicable when such data is unavailable. However, they are rarely reflected in standard open-source benchmarks. This motivates our exploration of LLM-based FL in this more realistic and restricted setting. In contrast to conventional tools, LLMs offer the potential to reason over static contextual information [14, 34], such as stack traces, crash messages, and source code.

To this end, we evaluate the performance of the recent AutoFL [14] tool, an LLM-based FL technique, on crashdumps collected from SAP HANA [1], a large-scale commercial in-memory database system, consisting of more than 35 million lines of code, primarily implemented in C and C++. At a high level, AutoFL operates by providing an LLM with contextual information about a bug and allowing the model to autonomously invoke *functions* which retrieve additional evidence from the code repository and available coverage information, eventually synthesizing this information to identify the most likely location of the fault. We chose AutoFL as the foundation for our LLM-based crash FL approach due to its lightweight design, high degree of customizability, and language-agnostic architecture, which make it especially suitable for integration into our industrial context.

Nevertheless, applying AutoFL in our setting requires significant domain-specific adaptation. In its original form, AutoFL takes error messages and stack traces as input, and uses coverage information from failing test cases to guide and constrain the fault search space. In contrast, our target industrial environment requires the use of extremely large crashdump files, which differ significantly in both

scale and structure, while coverage data from the crash execution is not available. To bridge this gap, we replace the original function set used by AutoFL with a new suite of tools specifically tailored for crash debugging. These include functions to access different segments of the given crashdumps, to retrieve relevant source code snippets from the repository, and to resolve symbol locations in the codebase to enable deep, context-aware source exploration. These modifications allow the LLM to reason more effectively over complex industrial crash data. For clarity, we hereafter refer to this adapted, crash-oriented variant of the tool as AutoCrashFL.

From the CI/CD system of SAP HANA, we collected 454 crashes that were reported and debugged between April 2023 and March 2024. Using these crashes, we evaluate AutoCrashFL against stack-trace-based heuristic baselines, and report that AutoCrashFL correctly ranked the root cause at the top for 134 crashes (30%), outperforming the baselines, which achieved 49 (11%) and 75 (17%), respectively. In summary, our contributions are as follows:

- We identify a common industrial problem, FL for crashes, and detail how it differs from usual FL formulations;
- We propose AutoCrashFL, an FL technique specialized towards crashes, and which can handle an industrial scale;
- We perform a thorough evaluation of AutoCrashFL, identifying its efficacy and current limitations.

The remainder of this paper is organized as follows. Section 2 provides academic background. Section 3 presents the design of AutoCrashFL, and Section 4 details the experimental setup. Section 5 reports the evaluation results, and Section 6 analyzes the root-cause explanations. Finally, Section 7 concludes.

2 Background

This section provides the academic background of this work.

2.1 Fault Localization

Fault Localization (FL) is part of the debugging process, in which a developer or automated process discerns which part of the software is responsible for undesirable behavior. As the developer study of Böhme et al. [2] demonstrates, FL takes a significant portion of time for developers. As such, automating FL has been studied since the Tarantula [12] technique was introduced. Multiple different families of techniques have been developed by researchers, including spectrum-based fault localization (SBFL) [12, 31], mutation-based fault localization (MBFL) [22, 23], and learning based fault localization techniques [20, 21]. Relevant to this work, the strong performance of LLMs on many software engineering domains [6] has led to their utilization in automated FL as well [14, 25]. In particular, this work adapts the AutoFL [14] tool, described in greater detail in Section 2.3.

2.2 LLM Agents for Automated Debugging

The flexibility of LLMs has allowed their adoption for many software engineering tasks, such as bug reproduction [16, 17], fault localization [14] and patch generation [4, 15, 26, 38]. While debugging requires inspection of a significant amount of information [3], LLMs have a limitation in that they can only process a limited amount of information at once. In response, LLM agents [7] which overcome this issue have rapidly been developed. LLM agents use tools to autonomously retrieve information while performing the task. Starting with examples such as AutoSD [15] and ChatRepair [35], the number of tools incorporated with LLM-based debugging techniques has increased [19]. While there are many LLM agents, we are unaware of agents that take crashdumps and employ LLM agents to identify the buggy files, as is done in this work.

2.3 AutoFL

AutoFL [14] is an FL technique that provides an LLM with tools that retrieve source code and documentation. This architectural decision aims to overcome a weakness of LLMs, in that they have a limited *context length*, i.e., the amount of text they can process at once is limited. By allowing LLMs to invoke tools, AutoFL allows the LLM to selectively retrieve important information without exceeding the context length.

In its original formulation, AutoFL begins with two inputs: (i) a failing test and (ii) its corresponding error message and call stack. With this information, AutoFL prompts the LLM to sequentially invoke tools and generate an explanation on how the bug occurs. The tools respectively show covered files, covered methods for a file, the source code of a particular method, and the documentation for a method (if it exists). Given the initial information, the LLM will iteratively use the tools to retrieve the relevant information for the bug, such as methods in the call stack. When the LLM judges that it has retrieved sufficient information to generate an explanation, it ceases to call a tool and generates a natural language description describing the mechanism of the bug. Upon this, the LLM is prompted to provide the precise signature of the buggy method. The resulting LLM response is treated as the predicted fault location, concluding a single 'run' of AutoFL. To improve performance and gauge confidence, multiple runs can be aggregated through a voting scheme. The aggregation through voting allows methods suggested over multiple runs to be suggested first, which is empirically correlated with performance.

In this work, we choose to modify AutoFL for fault localization of crashes as it can be adapted to the scenario of SAP HANA with relative ease. First, AutoFL already uses stack traces as part of its input, unlike other recent techniques [34, 38] which utilize developer-written bug reports. Second, its modular design allows easy adaptation for crashes: only the tools it uses need to be replaced, while the architecture can stay in place.

2.4 Crash Analysis

Meanwhile, program crashes have long been an issue for software [9], and as such, many techniques specialized in dealing with crashes have been proposed. First, crashdumps such as the ones used in this study are actually used by developers – Schröter et al. [27] find that bug reports accompanied with crash stacks are resolved more often. As crashes are a particularly visible type of bug, there are many techniques that attempt to reproduce crashes as well. EvoCrash [28] is one such technique that analyzes crash stacks and uses EvoSuite as a backbone to generate crash-reproducing tests. Relevant to this work, there have been many techniques that use crash stacks in conjunction with other analyses to perform FL, but they all fail at the scale of SAP HANA's codebase and its specific debugging scenario. F3 [11] generates a crash-reproducing

test, then performs SBFL and custom location filtering. In our industrial setting, tests are frequently unavailable and coverage is not collected, making such an approach infeasible. CrashLocator [33] performs static analysis and slicing to assign suspiciousness to functions, but without a test; we note that slicing at the scale of SAP HANA would be excessively time-consuming. The recent work of CrashTracker [37] similarly performs static analysis for FL, then uses LLMs to generate explanations. Unlike our work, the LLM is not directly involved in the FL process. Finally, Du et al. [5] evaluate the performance of ChatGPT in resolving crash bugs gathered from Stack Overflow. Most notably, in these cases the crash-causing code is already largely localized at the function level, making the work inapplicable to our scenario.

3 Methodology

Figure 1 shows the overall architecture of AutoCrashFL. We implement a parser for the crashdump, and a repository interface to access the code. The parser allows an LLM to easily access the information in the crashdumps by different segments – similarly to AutoFL, this is to mitigate the constraints from the context length. The repository interface allows an LLM to easily navigate the large code repository of SAP HANA. Based on these two toolkits, AutoCrashFL equips an LLM with four tools, allowing it to retrieve information and perform file-level fault localization. Finally, we run AutoCrashFL multiple times to generate independent reasoning trajectories, as prior work does [14], and aggregate the results into a final ranking of suspicious files. We describe each component of AutoCrashFL in more detail next.

3.1 Crashdump Parser

With SAP HANA, whenever an enterprise system crashes, the system information at the point of the crash is saved into a file known as a *crashdump*. A crashdump is typically organized into more than 50 sections, including BUILD, SYSTEMINFO, CRASH_EXTINFO, CRASH_STACK, and THREADS, each providing different kinds of contextual information about the system state at the time of the crash. Professional developers look into these crashdumps to understand how the program crashed and how it can be fixed. As such, we opt to give the information in the crashdump to AutoCrashFL. However, crashdumps are often too long for an LLM to handle – the median size of the crashdumps in our study is 6.6 megabytes, or about 3.1 million tokens, 24 times longer than the context length of gpt-4o. In fact, only one raw crashdump in our dataset fits in gpt-4o's context length of 128,000. The longest crashdumps contain hundreds of megabytes worth of information, making them difficult to deal with, despite increases in LLM context lengths.

To mitigate this issue, we develop a crashdump parser, which is used to extract the most relevant information for FL. The parser is based on the observation that many sections of the crashdumps contain information that is only useful for debugging in special cases. For example, the MOUNTINFO section contains a substantial amount of information about the hardware configuration of the execution environment, and is consequently only useful for rare hardware bugs. To prevent confusion from such sections and include only the most relevant information, the parser first divides

the crashdump into its constituent sections. Next, AutoCrashFL provides tools that retrieve the content of specific sections.

In particular, AutoCrashFL leverages two sections of the crashdump during operation: CRASH_EXTINFO and CRASH_STACK. Example excerpts of both sections are shown in Figure 1 within the Crashdump box. The CRASH_EXTINFO section provides background information about the crash, such as the process exit signal and system metadata (e.g., the accessed memory address in segmentation faults). This information enables the LLM to make informed decisions about the effect of the executed code. The CRASH_STACK section contains the symbolic stack backtrace at the point of failure, along with traces of unhandled pending exceptions. As AutoFL [14] begins FL from the failing test case, AutoCrashFL similarly requires concrete code locations from which AutoCrashFL can initiate its code search process using the repository interface. Among the available sections, we observe that CRASH_STACK provides the richest such information. Across all crashes contained in our dataset, an average of 37% of ground-truth buggy files had base filenames explicitly appearing in the stack trace, showing that while not exhaustive, stack traces expose a meaningful portion of buggy files. Prior crash fault-localization studies have likewise relied on stack traces as primary debugging cues [33].

In our experiment, when providing the contents of CRASH_EXTINFO and CRASH_STACK to the LLM through tools, we considered two settings: raw section content (default) and sanitized content. The sanitized CRASH_STACK retains only the source code locations (file path and line number) appearing in the stack trace while preserving their order, whereas the sanitized CRASH_EXTINFO is reformatted into a simple dictionary structure without verbose headers such as `--> Dump of siginfo contents <--`.

While not directly used by AutoCrashFL, we note that the BUILD section, which contains the git hash for the code version, is used to set up the environment for our experimentation. Furthermore, the CRASH_EXTINFO section is additionally used to distinguish bug types based on exit signal in Section 4 and 5.

3.2 Repository Interface

We provide two repository navigation tools to AutoCrashFL. The first tool takes as input a file path (within the target repository) and a line number, and retrieves the 10 preceding lines, the target line itself, and the 10 following lines. The second tool allows the LLM to point to a specific term within a line and use clangd, a compiler front-end for C/C++ implementing the Language Service Protocol (LSP), to automatically identify which file and line a term is defined in. By providing these tools to the LLM, we designed AutoCrashFL such that the LLM could navigate the repository as necessary and reach a conclusion on fault localization.

Note that, unlike AutoFL, we cannot rely on coverage information of the failing test, primarily due to the high overhead of coverage collection. SAP HANA spans more than 35 million LOC and 110 thousand files, occupying over 7GB. It is larger by orders of magnitude than commonly used open-source benchmarks: even the largest projects from the widely-used Defects4J [13], BugsInPy [30], or SWE-bench [10] benchmarks have less than a million LOC. Measuring coverage for every execution run would therefore incur prohibitively high costs, and in practice, it is only performed on a
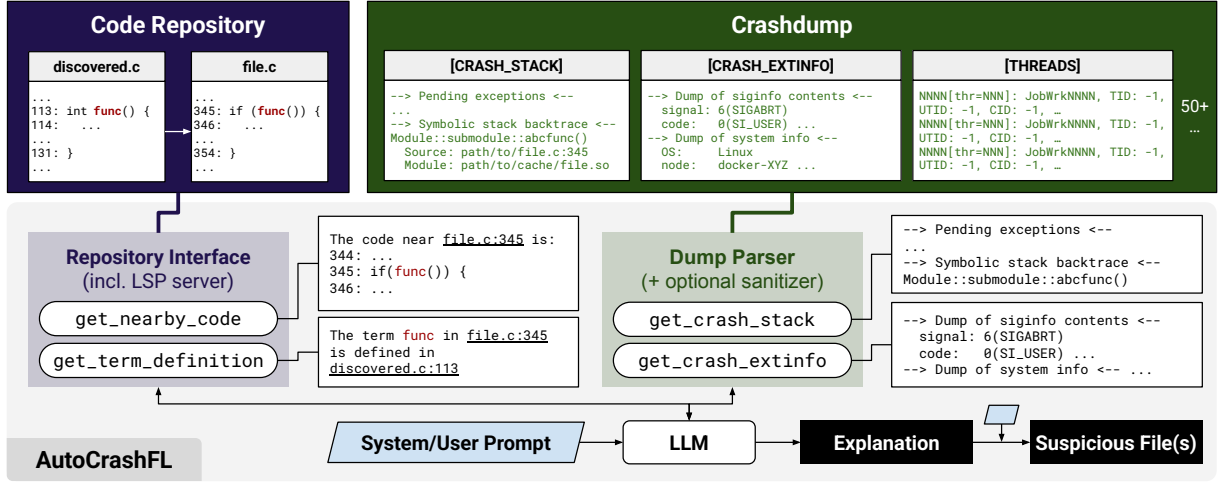
Figure 1: Diagram of AUTOCRASHFL.

weekly basis [1]. With coverage unavailable, we instead provide additional mechanisms that allow navigation of deeper parts of the repository that are not directly exposed in the stack trace. While we considered RAG (Retrieval Augmented Generation [18]) to provide code information, we decided against it due to the high cost of maintaining up-to-date indexing of the entire codebase, as well as the lack of control over the exact selection of code that is presented to the LLM. The clangd-based tool, on the other hand, fulfills this role by dynamically resolving symbol definitions and linking code references across the entire repository.

3.3 AUTOCRASHFL

Using the parsed information and the code navigation tools, AU-TOCRASHFL prompts an LLM to identify the file that is responsible for the crash. In particular, the LLM is given the following tools:

- get_crash_extinfo: Returns either the raw (or sanitized) summary of the crash signal (e.g., signal number and code), along with host/system metadata (e.g., OS and kernel release/version) and processor information.
- get_crash_stack: Returns either the raw (or sanitized) stack report extracted from the crashdump, including symbolic backtraces of relevant threads at the point of failure and any pending exceptions.
- get_nearby_code: as a function for code navigation, this function takes as input a target file path and line number, and returns the nearby 10 lines of code above and below the input line.
- get_term_definition: as another function for code navigation, this function takes as input a target file path, line number, and the name of an identifier. Based on this, it will return the location in the repository in which the identifier was defined (if the language server, clangd, can find the location).

Given this list of functions to call, the LLM autonomously retrieves information as it predicts is necessary. The prompts used to describe the role of the LLM are provided in Figure 2. The system prompt states the general objective of the agent, while the user

Figure 2: Prompts used by AUTOCRASHFL, lightly truncated for presentation.

prompt provides instance-specific information such as the budget and instructs the LLM to start calling tools. Guided by these prompts, AUTOCRASHFL iteratively calls tools to gather information until it reaches a conclusion with an explanation of why the bug happened. As is the case for AUTOFL, after the explanation is generated, the LLM is prompted once again to suggest a list of culprit files (as paths) responsible for the bug.

3.4 Ranking Aggregation

LLM behavior can be stochastic, yielding different results in different runs. To stabilize and improve the performance of AUTOCRASHFL, given the same crash, we run the tool-calling process from start to finish R times, and aggregate the results. In particular, for an LLM run that returns a set of suspicious files S, each file in the set is given a suspiciousness score of $1/|S|$; the scores for suggested files in all runs are added to give the final suspiciousness score for each file. Based on the score, we can derive two results. First, the score allows a ranking of all files that were suggested by the LLM in any run. Second, a 'confidence score' can be calculated, defined as
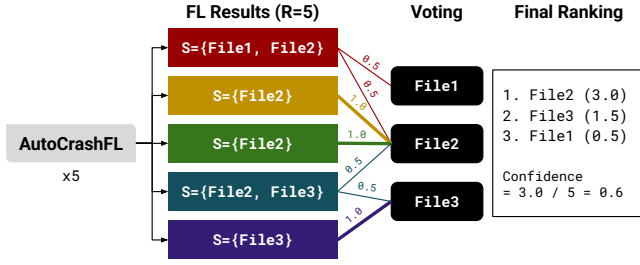
Figure 3: Rank Aggregation based on Voting

Table 1: Distribution of usable crash instances by crash type (Apr 2023–Mar 2024).

Crash Type	Count	%
SIGABRT	289	63.7
SIGSEGV (NPE)	118	26.0
SIGSEGV (Non-NPE)	43	9.5
SIGBUS	3	0.7
SIGFPE	1	0.2
Total	**454**	**100**

the suspiciousness of the file with the highest score divided by the number of runs, R. This confidence score is an indication of how consistently the LLM behaved. As we demonstrate in our results, confidence is correlated with better FL performance; thus, it can be used to present the highest-quality results to developers.

4 Experimental Setting

We conduct an empirical evaluation of AutoCrashFL on the large C/C++ codebase SAP HANA to assess its feasibility for enterprise-scale crash debugging. This section provides details of the experimental setting.

4.1 Research Questions

Our empirical evaluation is designed to answer the following research questions.

- **RQ1. Effectivenss of AutoCrashFL:** How effective is AutoCrashFL at localizing the root cause of collected crashes? We compare its localization accuracy to the heuristic baselines (described in Section 4.5).
- **RQ2. Performance Across Different Crash Types:** Are there specific types of crashes against which AutoCrashFL performs better or worse? We compare the performance of AutoCrashFL against different crash types and investigate the reason.
- **RQ3. Impact of Repeated Runs:** Do the repeated runs of AutoCrashFL and the voting-based aggregation contribute to its overall performance? To examine the cost–benefit trade-offs between repeated runs and accuracy, we compare the performance of AutoCrashFL under different numbers of runs (R).
- **RQ4. Contribution of Provided Functions:** Do all functions provided to AutoCrashFL contribute to its performance? We look at the patterns of tool usage, and present case studies of good and bad tool uses.

4.2 Dataset Collection

We collect actual SAP HANA crash reports that had been previously debugged by developers and thus could be linked to their corresponding fixing commits. In total, we investigate 527 crash instances reported during the one-year period from April 2023 to March 2024, which represents a recent yet sufficiently mature time window at the time of initiating our study (i.e., enough time had passed for fixes to be produced by developers). After excluding 56 instances with invalid Git hashes in the crash dumps (e.g., dummy or temporary commit identifiers not present in the remote repository) and 17 instances for which the language server fails to initialize

(mainly due to the old `clang` version), 454 crashes remain usable for evaluation. On average, these usable crash instances require changes to 4.37 files in their corresponding fixing commits.

We categorize the crash instances by crash type and present their distribution in Table 1. The most frequent categories are SIGABRT (abort signal) and SIGSEGV (segmentation fault), whereas SIGBUS (bus error) and SIGFPE (computational error) are rarely observed. Through interviews with developers who regularly debug such crashes, we learn that SIGSEGV crashes are typically further classified based on the *faulting address*, which is usually the first detail examined during debugging. If the recorded address is either `0x00` (Null) or a close-to-zero value (e.g., `0x3d`, representing `Null+offset`), the crash is identified as a *null-pointer dereference*, aka a Null-Pointer Exception (NPE). Otherwise, it is treated as a non-NPE segmentation fault. Following this practice, we subdivide the collected SIGSEGV crashes into NPE and non-NPE categories.

4.3 AutoCrashFL Parameters

We evaluate AutoCrashFL with the maximum number of function interactions set to $N = 25$, a setting that ensures sufficient exploration of the code context without incurring excessive inference costs. Each crash is analyzed over $R = 10$ repeated runs, allowing us to aggregate rankings across stochastic LLM outputs and obtain more stable results, as described in Section 3.4. By default, the functions `get_crash_extinfo` and `get_crash_stack` return the **raw** contents of their respective sections. For all experiments, we use OpenAI's `gpt-4o` as the underlying LLM.

For RQ3, we investigate the effect of repeated runs by varying R from 1 to 10 and comparing the FL accuracy of the resulting aggregated rankings. For RQ4, we keep all other settings identical to the default configuration, but configure `get_crash_extinfo` and `get_crash_stack` to return sanitized contents of their corresponding crashdump sections. In addition, we disable the deep-search functionality (i.e., we do not provide the `get_term_definition` tool to the LLM).

4.4 Evaluation Metric

We use the widely adopted FL accuracy metric, acc@k (or top-k accuracy), here defined as the number of crash instances for which at least one actual faulty file (contained in the corresponding fixing commit) appears within the top-k ranked results. We evaluate acc@k for $k \in \{1, 2, 3, 5, 10\}$. We also report the corresponding percentage of the total number of evaluated crash instances, for easier comparison of FL performance between different crash types.
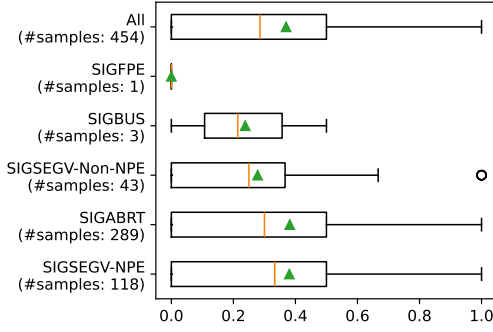
Figure 4: Ratio of buggy files whose base filenames explicitly appear in the `CRASH_STACK`. The green triangle denotes the mean value, while the orange line indicates the median.

4.5 Heuristic Baselines

As discussed in Section 3.1, the crash stack in a crashdump file often provides useful hints about the location of the buggy file. In our scenario, a typical stack trace, provided by the CRASH_STACK section of the crashdump, contains two main components: (1) a *symbolic stack backtrace*, representing the call stack at the point where the crash occurs, and (2) a list of *pending exceptions*, which captures the stack trace of unhandled exceptions that were pending at the time of the crash. Figure 4 shows the ratio of buggy files whose base filenames explicitly appear in the CRASH_STACK, both overall and by crash type. The value of AutoCrashFL lies in providing additional benefit beyond simply investigating the files appearing in the stack trace. Therefore, to more rigorously assess its accuracy, we compare AutoCrashFL against the following baselines, designed through consultation with SAP HANA developers.

- **Baseline 1: Symbolic Stack Backtrace Only.** This baseline prioritizes files based solely on their position in the symbolic stack backtrace. Files are ranked in the order they appear in the stack trace, from top (most recent call) to bottom. This follows the observation of Wu et al. [33], who note that functions closer to the crash point are more likely to be faulty. Duplicate files are removed while preserving order. The resulting list serves as the FL ranking.
- **Baseline 2: Pending Exceptions First, Symbolic Stack Backtrace Second.** In addition to the top-to-bottom assumption, Baseline 2 further utilizes files that appear in the pending exceptions stack trace. It assumes they are more indicative of the root cause as they reflect the initial deviation from expected behavior. Accordingly, we first extract files from the pending exceptions trace, followed by those from the symbolic stack backtrace, mapping each to its source file and removing duplicates. The concatenated list is used as the FL ranking.

5 Results

This section provides the results for each research question.

Table 2: Comparison of AutoCrashFL and heuristic baselines in terms of localization accuracy, reported as acc@k (a@k). Results are based on 454 crash instances. The second column (Len.) denotes the average length of FL rankings produced by each technique.

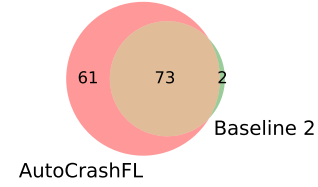| Technique	Len.	a@1	a@2	a@3	a@5	a@10
Baseline 1	14.76	49 (11%)	88 (19%)	142 (31%)	192 (42%)	255 (56%)
Baseline 2	16.49	75 (17%)	138 (30%)	196 (43%)	243 (54%)	**295** (65%)
AutoCrashFL	3.83	**134** (30%)	**195** (43%)	**227** (50%)	**245** (54%)	255 (56%)

Figure 5: Venn diagram of crash instances localized at the top by AutoCrashFL and Baseline 2

5.1 RQ1: Effectiveness of AutoCrashFL

Table 2 shows the performance of AutoCrashFL compared to the heuristic baselines. AutoCrashFL clearly outperforms both baselines in terms of acc@k for $k \in \{1, 2, 3, 5\}$. In particular, the strong acc@1 demonstrates that AutoCrashFL can pinpoint the buggy file even when it does not appear at the top of either the symbolic backtrace or the pending-exception list. Between the two baselines, Baseline 2 consistently outperforms Baseline 1, suggesting that prioritizing files appearing in the pending exceptions is more effective in practice than considering only the files in the symbolic stack at the crash point.

Figure 5 further analyzes the overlap between AutoCrashFL and Baseline 2 for top-1 localization. Out of 454 crashes, AutoCrashFL uniquely localizes 61 crashes at the top rank, while Baseline 2 uniquely localizes 2 crashes. The two methods agree on 73 crashes, indicating that AutoCrashFL covers most of Baseline 2's successes while also expanding coverage to additional crashes that the heuristic fails to capture. This demonstrates that AutoCrashFL not only achieves higher accuracy overall, but also contributes unique localizations beyond what stack-based heuristics can provide.

The only exception is acc@10, where Baseline 2 achieves better performance. This can be explained by the fact that AutoCrashFL produces rankings that are, on average, less than one-third the length of those generated by the baselines (3.83 vs. 14–16 files). While it is notable that AutoCrashFL improves accuracy while simultaneously reducing developer effort by providing shorter rankings, this conciseness may also omit valuable contextual information available in the original stack trace.

Table 3: Performance of augmented AutoCrashFL rankings. "Aug." denotes that the AutoCrashFL ranking is extended with the files from the corresponding baseline's ranking.

Technique	Len.	a@1	a@2	a@3	a@5	a@10
AutoCrashFL	3.83	**134** (30%)	195 (43%)	227 (50%)	245 (54%)	255 (56%)
AutoCrashFL Aug. w/ Baseline 1	15.96	**134** (30%)	**198** (44%)	234 (52%)	258 (57%)	292 (64%)
AutoCrashFL Aug. w/ Baseline 2	16.49	**134** (30%)	**198** (44%)	**235** (52%)	**263** (58%)	**308** (68%)

In the original AutoFL study, this limitation was mitigated by augmenting the generated rankings with additional candidates that were not directly identified by AutoFL but were covered by the failing test. Following this strategy, we also augment the AutoCrashFL-generated rankings with the files included in each baseline. For example, if the stack trace contains five files f_1, f_2, f_3, f_4, f_5, in order from top to bottom, and AutoCrashFL produces the ranking $[f_1, f_5]$, the remaining files are appended to the rankings, resulting in the augmented ranking $[f_1, f_5, f_2, f_3, f_4]$.

Table 3 reports the performance of AutoCrashFL when its rankings are augmented with the file rankings from each baseline. When augmented, the acc@1 remains unchanged at 134, but top-2 and top-3 improve slightly. At larger k values, the augmented versions yield substantial gains. In particular, when AutoCrashFL is augmented with Baseline 2, acc@10 reaches 308 (68%), the best among all settings. Overall, these results confirm that augmentation helps recover the contextual coverage that may be lost in AutoCrashFL's concise rankings, especially at larger k. While the augmented versions surpass the baselines, they do so by increasing the ranking length and thus the developer's inspection cost. In practice, this highlights a trade-off: unaugmented AutoCrashFL offers concise, high-precision rankings (lower effort), while augmented versions achieve higher recall at the cost of longer candidate lists. **In the remaining sections, we focus only on the original AutoCrashFL-generated rankings to clearly demonstrate its performance in isolated settings.**

5.2 RQ2: Performance Across Crash Types

Table 4 shows the performance of AutoCrashFL compared to the heuristic baselines across different crash types. Since the sample sizes of SIGBUS and SIGFPE were too small, we report them together. Overall, AutoCrashFL outperforms both baselines in terms of acc@k for smaller values of k, indicating that AutoCrashFL more accurately pinpoints the actual faulty location. As shown in RQ1, performance for larger k can be further improved by augmenting the ranking list with baseline results; however, we do not consider such augmentation here.

Notably, AutoCrashFL's relative benefit is most prominent for SIGABRT crashes. For example, AutoCrashFL achieves acc@1 of 82 cases (28%) on SIGABRT, compared to only 6 (2%) and 33 (11%) for Baseline 1 and Baseline 2, respectively. Between the two baselines, Baseline 2 significantly outperforms Baseline 1, suggesting that for SIGABRT crashes, the true buggy location is

more often contained in the pending-exception list rather than the symbolic backtrace. This is because SIGABRT is frequently triggered by explicit calls to abort() or by runtime checks (e.g., failed assertions), where the symbolic backtrace only shows the abort site, while the pending exceptions preserve the context of the original faulty operation.

For SIGSEGV (NPE) crashes, the performance gap narrows. While Baseline 1 and Baseline 2 achieve competitive results at larger k values (as also shown in RQ1), AutoCrashFL still provides the best top-1 accuracy (38 cases, 32%) and maintains steady improvements through acc@10. Interestingly, in this category Baseline 1 slightly outperforms Baseline 2. This is because when a program dereferences a null pointer, the segmentation fault occurs immediately, and the symbolic backtrace typically records the precise function and location of the invalid access. Indeed, for SIGSEGV (NPE), many crashes had no pending exceptions at all, as reflected by the relatively small difference in the ranking lengths produced by the two baselines compared to other crash types. Overall, SIGSEGV (NPE) represents the easiest category for fault localization, both for the baselines and for AutoCrashFL.

For SIGSEGV (non-NPE) crashes, baseline methods perform worse than for SIGSEGV (NPE) despite both being segmentation faults, likely due to the inherent difficulty of this category. As shown in Figure 4, the average ratio of ground-truth buggy files whose base filenames explicitly appear in the stack trace is relatively low, even though more files tend to be included in the trace (as reflected by the longer FL rankings of the baselines). A plausible explanation is that while these crashes are also segmentation faults, the faulting address is often a large arbitrary value, making its connection to the intended memory access ambiguous. In such cases (e.g., buffer overflows, dangling pointers, or incorrect offset calculations), the top frame in the crash stack may only indicate where the invalid memory access was detected, whereas the true defect lies earlier in the execution path. This ambiguity makes SIGSEGV (non-NPE) crashes more difficult to localize than SIGSEGV (NPE) crashes when relying solely on the crash stack, thereby explaining the baselines' lower accuracy. Nevertheless, AutoCrashFL substantially improves top-1 accuracy to 31%, reaching a level comparable to NPE bugs and highlighting its value. Finally, for SIGBUS and SIGFPE crashes, only four samples were available, making them rare crash types and limiting the ability to draw generalized conclusions from the observed performance. For these crashes, AutoCrashFL performed on par with or better than the baselines.

Taken together, these results suggest that AutoCrashFL generalizes well across different crash types, with particularly large advantages for SIGABRT and SIGSEGV (non-NPE) cases, while still providing competitive performance for SIGSEGV (NPE).

5.3 RQ3: Impact of Repeated Runs

In its original paper, AutoFL reported improved performance when the number of repeated runs R used for aggregation was increased [14]. We investigate whether AutoCrashFL exhibits a similar trend; specifically, whether aggregating more runs for each crash improves FL accuracy. Figure 6 shows that FL accuracy consistently improves as R increases. For example, acc@1 rises from 123 crashes with $R = 1$ to 134 crashes with $R = 10$, and acc@10 increases from

Table 4: Performance of AUTOCRASHFL compared to heuristic baselines across different crash types, reported is acc@k (a@k). Each block corresponds to one crash type: SIGABRT, SIGSEGV due to Null Pointer Exceptions (NPE), SIGSEGV due to other errors (non-NPE), and two minor types (SIGBUS/SIGFPE). The column Len. denotes the average length of FL rankings produced by each technique, and numbers in parentheses indicate percentages relative to the total number of crashes in each category.

Technique	SIGABRT (total = 289)						SIGSEGV (NPE) (total = 118)					
	Len.	a@1	a@2	a@3	a@5	a@10	Len.	a@1	a@2	a@3	a@5	a@10
BASELINE 1	14.58	6 (2%)	11 (4%)	53 (18%)	92 (32%)	145 (50%)	14.41	33 (28%)	**61** (52%)	70 (59%)	**80** (68%)	**83** (70%)
BASELINE 2	17.01	33 (11%)	64 (22%)	108 (37%)	143 (49%)	**184** (64%)	14.91	32 (27%)	58 (49%)	69 (58%)	80 (68%)	84 (71%)
AUTOCRASHFL	4.13	**82** (28%)	**115** (40%)	**135** (47%)	**148** (51%)	155 (54%)	3.01	**38** (32%)	**61** (52%)	**72** (61%)	74 (63%)	75 (64%)

Technique	SIGSEGV (Non-NPE) (total = 43)						SIGBUS/SIGFPE (total = 4)					
	Len.	a@1	a@2	a@3	a@5	a@10	Len.	a@1	a@2	a@3	a@5	a@10
BASELINE 1	17.00	9 (21%)	15 (35%)	18 (42%)	19 (44%)	**25** (58%)	13.50	**1** (25%)	**1** (25%)	**1** (25%)	1 (25%)	**2** (50%)
BASELINE 2	17.60	9 (21%)	15 (35%)	18 (42%)	19 (44%)	**25** (58%)	13.50	**1** (25%)	**1** (25%)	**1** (25%)	1 (25%)	**2** (50%)
AUTOCRASHFL	3.90	**13** (31%)	**18** (43%)	**19** (45%)	**21** (50%)	23 (55%)	3.25	**1** (25%)	**1** (25%)	**1** (25%)	**2** (50%)	**2** (50%)

Figure 6: Effect of R on AUTOCRASHFL's FL performance

Figure 7: Success ratio of AUTOCRASHFL across different confidence ranges. Success is defined as ranking the actual buggy file at the top position.
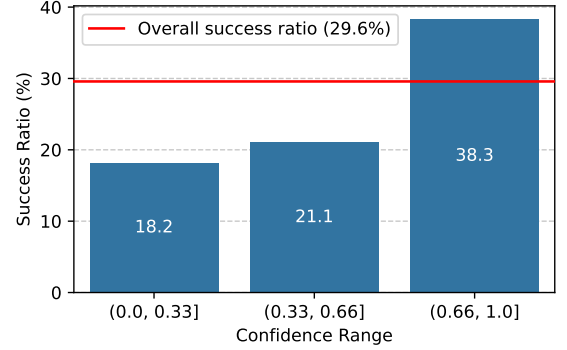
182 to 255. These results confirm that aggregating multiple runs yields more stable and accurate rankings. Because the computational cost increases approximately linearly with R, it is advisable to perform as many runs as the available budget allows and apply voting-based aggregation to improve reliability. However, the marginal gains diminish as R grows, aligning with the diminishing returns observed in the original study.

As described in Section 3.4, after aggregation, we also measure AUTOCRASHFL's confidence, defined as the maximum suspiciousness score among its FL outputs, following the strategy of AUTOFL. To assess whether this confidence score correlates with actual performance, we partition the score range into three buckets: (0, 0.33] (*low*), (0.33, 0.66] (*medium*), and (0.66, 1.00] (*high*), ensuring that each bucket contains at least 30 samples. Figure 7 shows the success rate for each bucket, where success means that AUTOCRASHFL ranks the correct buggy file at the top. While the overall success rate

is 29.6%, it rises to 38.3% in the *high*-confidence bucket and drops to 21.1% and 18.2% in the *medium* and *low* buckets, respectively. These results suggest that developers may benefit from filtering out lower-confidence predictions to prioritize higher-quality results.

To quantify the relationship between confidence and actual FL success more precisely, we compute the Point-Biserial correlation between AUTOCRASHFL's continuous confidence scores and the binary success labels. The resulting correlation is 0.23 ($p < 0.001$), which is weaker than the 0.57 reported in the original AUTOFL paper, yet still indicates a statistically significant positive association. We further assess the calibration of AUTOCRASHFL's confidence score, i.e., its alignment with the true probability of FL success, using the Brier score [8], which quantifies the mean squared error between
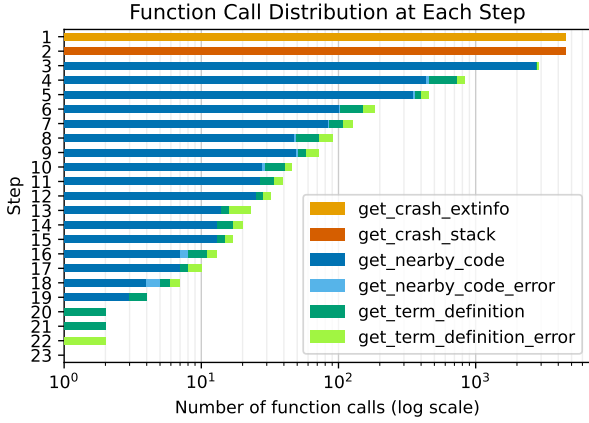
Figure 8: Distribution of function calls made by AutoCrashFL across reasoning steps.

predicted confidence values and binary success labels (lower is better). AutoCrashFL yields a Brier score of 0.357, notably worse than a naive predictor that always outputs 0.5 (Brier score of 0.25), indicating that the raw confidence values are poorly calibrated.

To address this, following recent LLM calibration work [29], we apply Platt Scaling [24], a post-hoc calibration technique that fits a logistic regression model to map predicted probabilities (here, confidence scores) to observed outcomes. We perform 5-fold cross-validation to obtain rescaled confidence scores across all crash instances. This calibration reduces the Brier score to 0.199, demonstrating that rescaling can significantly improve the alignment between confidence and actual success, making the scores more interpretable and actionable for developers.

5.4 RQ4. Contribution of Provided Functions

To understand AutoCrashFL's patterns of tool usage across all 4,540 AutoCrashFL runs (454 crashes × 10 repeated runs), we analyze which function the LLM invoked at each step of the reasoning process. On average, AutoCrashFL issued 3.1 function calls per run (maximum=23), indicating that it frequently terminates early (i.e., stops requesting further function calls and directly provides a root-cause explanation), far before exhausting its function-calling budget. Figure 8 shows the distribution of function usage across the reasoning steps over all AutoCrashFL runs. We observe that, as advised by the input prompt, `get_crash_extinfo` and `get_crash_stack` are invoked in the early steps, after which the LLM begins to examine additional information, such as the nearby code referenced in the stack trace. As intended, once it inspects the code, the LLM also attempts to locate the definitions of specific terms found within the inspected lines by calling `get_term_definition`, which in most cases (99.6% in our evaluation) naturally triggers an additional call to `get_nearby_code`.

Figure 8 also shows that some function invocations failed. Specifically, `get_nearby_code` failed in 77 cases (2%), and `get_term_definition` failed in 313 cases (40%). We further categorized the causes of these failures. All 77 `get_nearby_code` errors resulted from the LLM requesting code for files that could not be located. Among

Table 5: Distribution of `get_term_definition_error` cases.

Error Type	Count
Term does not appear in the given line	263
File not found	28
Term appears in the line, but navigation failed	15
File found but invalid line reference	3
Invalid argument format	4
Total	**313**

Table 6: Ablation results for AutoCrashFL under two configurations: disabling deep search by omitting `get_term_definition`, and using sanitized sections instead of raw contents.

Technique	Len.	a@1	a@2	a@3	a@5	a@10
AutoCrashFL	3.83	134 (30%)	195 (43%)	227 (50%)	245 (54%)	255 (56%)
AutoCrashFL w/o Deep Search	3.59	135 (30%)	199 (44%)	224 (49%)	243 (54%)	248 (55%)
AutoCrashFL w/ Sanitization	3.88	130 (29%)	189 (42%)	212 (47%)	244 (54%)	250 (55%)

these, 26 and 21 cases involved absolute and relative paths that were not resolvable within the project context, respectively. The remaining 30 cases appeared to involve hallucinated paths, where the requested filenames were not found and not even present in the stack trace. The 313 failures of `get_term_definition` are summarized in Table 5. In most cases, the LLM attempted to resolve the location of a symbol that did not exist at the specified code line. A contributing factor appears to be difficulty in interpreting the line-numbered code format produced by `get_nearby_code`, where line numbers are prepended to each line of the snippet. Additional failure cases included requests for non-existent files and language server being unable to resolve the symbol location.

To more accurately assess the role of deep search, we disable AutoCrashFL's `get_term_definition` functionality and examine the resulting performance change. Interestingly, as shown in Table 6, accuracy at lower k values slightly improves in this configuration. This can be attributed to the previously observed fact that the LLM may struggle to effectively leverage the deep search tool in certain cases. Without the tool, the LLM may instead focus more narrowly on analyzing the stack trace itself, leading to improved precision for top-ranked predictions. However, performance at higher k values declines when deep search is disabled. Upon further inspection, we found that with deep search enabled, AutoCrashFL included the correct buggy file in its FL rankings in 8 cases where the file's basename did not appear in the original stack trace. In these cases, the LLM identified relevant symbols in stack-trace contained files and successfully traced their definitions to external files, which were then marked as suspicious. Although these files were not ranked at the top, their inclusion demonstrates the potential value of deep search in uncovering non-obvious fault locations. These findings indicate that while the current interface between the LLM and the deep search tool may not always be effective, further refinement, e.g., improving symbol resolution prompts or tool response formats,

could enhance the utility of deep search and ultimately improve AutoCrashFL's overall performance.

As discussed in Section 3.1, we also evaluate the effect of providing AutoCrashFL with sanitized crashdump content via `get_-crash_stack` and `get_crash_extinfo`, instead of the raw section contents. This experiment is motivated by the observation that sanitization substantially reduces the number of input tokens, thereby lowering inference cost and improving compatibility with context length limits. For instance, the raw `CRASH_STACK` section contains an average of 9,326 tokens, which drops to 1,873 tokens after sanitization. Similarly, the `CRASH_EXTINFO` section is reduced from 240 to 205 tokens. As shown in Table 6, overall FL performance is slightly lower when using sanitized inputs, but the drop is negligible. These results suggest a trade-off between cost and performance. Sanitized content may be preferable when resource constraints are a concern or when raw inputs exceed the model's context length. Indeed, among the 454 crash instances, we encountered one case where the raw `CRASH_STACK` input triggered a context limit error, and sanitization was necessary for AutoCrashFL to function properly.

6 Explanation Reliability Assessment

For resolved crashes at SAP HANA, post-mortem root cause summaries are generated by LLMs. These summaries synthesize information from multiple failure-related artifacts, including the issue report (title, description, and comments), the ground-truth fix diff, and contents from relevant crashdump sections such as `BUILD`, `CRASH_STACK`, and `CRASH_SHORTINFO`. We refer to such summaries as *postmortem analyses*.

Since AutoCrashFL produces root cause explanations before pinpointing the actual file location (as illustrated in Figure 1), a natural question arises: *Do the explanations generated by AutoCrashFL semantically align with the postmortem analysis?* If they do align, explanations from AutoCrashFL could provide significant insight to developers, as if they were hearing a description of the bug after it was resolved. To answer this, we generate a consolidated explanation for each file suggested by AutoCrashFL. More specifically, for every flagged file, we collect all explanations generated across runs where that file appears in the final prediction, and aggregate them into a single summary using `gpt-4o`. For example, suppose we run AutoCrashFL three times for a given bug, and obtain predictions: $\{F_a\}$ (with explanation E_1), $\{F_a, F_b\}$ (with E_2), and $\{F_c\}$ (with E_3). In this case, the explanation for suspecting file F_a is derived by summarizing E_1 and E_2.

To assess the semantic alignment between the aggregated explanations and the corresponding postmortem analyses, we forgo lexical similarity metrics (e.g., token overlap, BLEU), which fail to capture deeper semantic correspondence. Instead, we adopt an LLM-as-a-judge protocol [39], leveraging the reasoning capabilities of LLMs to evaluate semantic alignment. For each crash file, we present the postmortem analysis alongside the AutoCrashFL-generated explanation summary and prompt the LLM (here, `gpt-4o`) to issue a categorical judgment: `aligned` or `misaligned`. To enhance robustness, we collect five independent judgments (each from a separate LLM invocation) and assign the final label based on majority vote. We report two alignment metrics. First, the overall alignment rate, defined as the proportion of crashes where at

least one file-level explanation is judged as being aligned with the postmortem analysis, is 37%. For example, the postmortem analysis for one SIBABRT crash was as follows:

> *"The crash was caused by a **race condition** in the job execution system (...). The issue arises when an exception occurs in one job while multiple jobs are being processed in parallel (...)"*

While AutoCrashFL produced the following aligned explanation:

> *"<BUGGY_FILE> could exhibit problematic behavior if the job queue management system does not properly synchronize access to job queues (...). Potential issues might include **race conditions** or (...)."*

Second, acknowledging that developers are most likely to consult higher-ranked files, we report the top-1 and top-3 alignment rates, i.e., the fraction of crashes for which at least one explanation among the top 1 or top 3 ranked files is judged aligned with the postmortem analysis. This occurs in 113 out of 454 crashes for the top-1 explanation, yielding a top-1 alignment rate of 25%. For the top-3 explanations, alignment is observed in 154 crashes, corresponding to a top-3 alignment rate of 34%.

Overall, it is encouraging that AutoCrashFL could generate explanations that aligned with postmortem analyses – which were generated with full information about the crash – for a non-negligible portion of bugs. At the same time, it leaves significant room for improvement. While the LLM-as-a-judge procedure provided promising initial results, we note it is not clear what practitioners need from explanations; this knowledge would help improve the utility of explanations in future work.

7 Conclusion

In this work, we introduced AutoCrashFL, an LLM-driven FL approach that extends AutoFL and is tailored for large-scale industrial crash debugging. Unlike traditional SBFL or MBFL techniques that rely on costly runtime data, AutoCrashFL operates directly on crashdumps and source code, making it practical in production environments where coverage and reproduction are often unavailable. Our evaluation on 454 real-world SAP HANA crashes shows that AutoCrashFL consistently outperforms heuristic baselines across diverse crash types, with particularly strong results for SIGABRT and non-NPE segmentation faults. These findings highlight the potential of LLM-based agents as effective crash debugging assistants. At the same time, challenges remain in calibrating confidence scores and improving deep-search interactions. We see AutoCrashFL as a first step towards reliable integration of LLM-based FL into industrial crash-debugging workflows, and we hope this study inspires further research on LLM agents in realistic, information-constrained settings.

References

[1] Thomas Bach, Artur Andrzejak, Changyun Seo, Christian Bierstedt, Christian Lemke, Daniel Ritter, Dong Won Hwang, Erda Sheshi, Felix Schabernack, Frank Renkes, et al. 2022. Testing very large database management systems: The case of SAP HANA. *Datenbank-Spektrum* 22, 3 (2022), 195–215.

[2] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is It Fixed? An Experiment with Practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE*

2017). Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/3106237.3106255

[3] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is it Fixed? An Experiment with Practitioners. In *Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2017).* 1–11.

[4] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. arXiv:2403.17134 [cs.SE] https://arxiv.org/abs/2403.17134

[5] Xueying Du, Mingwei Liu, Juntao Li, Hanlin Wang, Xin Peng, and Yiling Lou. 2023. Resolving crash bugs via large language models: An empirical study. *arXiv preprint arXiv:2312.10448* (2023).

[6] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE).* IEEE, 31–53.

[7] Robert Feldt, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Towards autonomous testing agents via conversational large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 1688–1693.

[8] W Brier Glenn et al. 1950. Verification of forecasts expressed in terms of probability. *Monthly weather review* 78, 1 (1950), 1–3.

[9] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* 103–116.

[10] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations.* https://openreview.net/forum?id=VTF8yNQM66

[11] Wei Jin and Alessandro Orso. 2013. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis.* 213–223.

[12] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) *(ICSE '02).* Association for Computing Machinery, New York, NY, USA, 467–477.

[13] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis.* 437–440.

[14] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1424–1446.

[15] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. 2025. Explainable Automated Debugging via Large Language Model-driven Scientific Debugging. *Journal of Empirical Software Engineering* 30, 45 (2025), 1–28.

[16] Sungmin Kang, Juyeon Yoon, Nargiz Askarbekkyzy, and Shin Yoo. 2024. Evaluating Diverse Large Language Models for Automatic and General Bug Reproduction. *IEEE Transactions on Software Engineering* 50, 10 (2024), 2677–2694.

[17] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023).*

[18] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[19] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. 2025. The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering. arXiv:2507.15003 [cs.SE] https://arxiv.org/abs/2507.15003

[20] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019).* Association for Computing Machinery, New York, NY, USA, 169–180. doi:10.1145/3293882.3330574

[21] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21).* IEEE Press, 661–673. doi:10.1109/ICSE43902.2021.00067

[22] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation.* 153–162. doi:10.1109/ICST.2014.28

[23] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-Based Fault Localization. *Softw. Test. Verif. Reliab.* 25, 5–7 (aug 2015), 605–628. doi:10.1002/stvr.1509

[24] John Platt et al. 1999. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers* 10, 3 (1999), 61–74.

[25] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362* (2024).

[26] Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. 2025. Evaluating Agent-based Program Repair at Google. arXiv:2501.07531 [cs.SE] https://arxiv.org/abs/2501.07531

[27] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE working conference on mining software repositories (MSR 2010).* IEEE, 118–121.

[28] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2020. Search-Based Crash Reproduction and Its Impact on Debugging. *IEEE Transactions on Software Engineering* 46, 12 (2020), 1294–1317. doi:10.1109/TSE.2018.2877664

[29] Yuvraj Virk, Premkumar Devanbu, and Toufique Ahmed. 2025. Calibration of Large Language Models on Code Summarization. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2944–2964.

[30] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering.* 1556–1560.

[31] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability* 63, 1 (2014), 290–308. doi:10.1109/TR.2013.2285319

[32] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[33] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* 204–214.

[34] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. doi:10.1145/3715754

[35] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024).* Association for Computing Machinery, New York, NY, USA, 819–831.

[36] Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. 2025. Flexfl: Flexible and effective fault localization with open-source large language models. *IEEE Transactions on Software Engineering* (2025).

[37] Jiwei Yan, Jinhao Huang, Chunrong Fang, Jun Yan, and Jian Zhang. 2024. Better debugging: Combining static analysis and llms for explainable crashing fault localization. *arXiv preprint arXiv:2408.12070* (2024).

[38] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024).* 1592–1604.

[39] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems* 36 (2023), 46595–46623.