Is Fault Localization Effective on Industrial Software? A Case Study on Computer-Aided Engineering Projects

ZHILEI REN, School of Software, Dalian University of Technology, China
YUE MA, School of Software, Dalian University of Technology, China
XIAOCHEN LI, School of Software, Dalian University of Technology, China
SHIKAI GUO, School of Information Science and Technology, Dalian Maritime University, and Dalian Key Laboratory of Artificial Intelligence, China
HE JIANG*, School of Software, Dalian University of Technology, Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, and DUT Artificial Intelligence Institute, China

In software engineering, empirical studies on automated fault localization (FL) methods mainly focus on general software, and substantial progress have been made. However, the applicability and efficacy of these methods in specialized, domain-specific software like industrial software remains under-explored. Such specialized software is usually characterized by complex inputs and iterative computing paradigms, which could significantly influence the effectiveness of existing FL methods. To address this gap, this study takes a typical categorical of industrial software (i.e., computer-aided engineering (CAE) projects) as a case study, to investigate the feasibility and effectiveness of state-of-the-art FL methods within CAE projects. Through the reproduction of 76 real-world bugs from three widely used CAE projects (i.e., FDS, deal.II, and MFEM), we find that even the most precise FL methods require developers to examine on average 467.18 statements before finding bugs, and can take 208.13 hours to execute. The complex inputs and long-term computation characteristics of CAE projects further increase the difficulty of FL. Moreover, FL on CAE also faces challenges, such as insufficient differentiation of coverage information and missing CAE-specific FL features. Based on our findings, we improve FL on CAE projects by proposing a set of CAE main module based features, which improve the best-performed FL method in this study (i.e., DeepFL) by 35.93% and 45%, in terms of *MAR* and *MFR*, respectively.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Software testing and debugging**.

Additional Key Words and Phrases: Fault localization, Computer-Aided Engineering, debug, empirical study, verification and validation

*He Jiang is the corresponding author

Authors' addresses: Zhilei Ren, School of Software, Dalian University of Technology, Dalian, China, zren@dlut.edu.cn; Yue Ma, School of Software, Dalian University of Technology, Dalian, China, yuema@mail.dlut.edu.cn; Xiaochen Li, School of Software, Dalian University of Technology, Dalian, China, xiaochen.li@dlut.edu.cn; Shikai Guo, School of Information Science and Technology, Dalian Maritime University, and Dalian Key Laboratory of Artificial Intelligence, Dalian, China, shikai.guo@dlmu.edu.cn; He Jiang, School of Software, Dalian University of Technology, Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, and DUT Artificial Intelligence Institute, Dalian, China, jianghe@dlut.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1049-331X/2024/0-ART0
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Zhilei Ren, Yue Ma, Xiaochen Li, Shikai Guo, and He Jiang. 2024. Is Fault Localization Effective on Industrial Software? A Case Study on Computer-Aided Engineering Projects. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 0 (2024), 37 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

As software systems become increasingly complex, quickly and accurately locating the source of faults has become a challenge. In recent years, fault localization (FL) methods have accomplished promising achievements in addressing this challenge for general software systems [25]. However, their effectiveness and adaptability in software within specific industry fields (i.e., industrial software) remain unclear. Such specialized software is usually characterized by complex inputs and iterative computing paradigms, which could significantly influence the effectiveness of existing FL methods. In modern manufacturing, industrial software helps efficiently analyze and process production data, and provides insights for decision-making. Industrial software covers a wide range of applications, such as computer-aided design (CAD), computer-aided engineering (CAE), and computer-aided manufacturing (CAM). These tools are essential for designing products in safety-critical fields such as aerospace, automotive manufacturing, and electronic devices [18]. Especially, CAE projects require interdisciplinary knowledge to implement efficient numerical simulation algorithms [46]. This character increases the likelihood of bugs being introduced during the development process. Bugs in CAE projects can affect the entire numerical simulation process, and may lead to inaccurate or misleading results [18]. This unreliable output may have a negative impact on the engineering design. Therefore, bug resolution is crucial in the field of industrial software.

Unfortunately, currently for CAE projects, the localization task for bugs is mostly conducted by developers manually. Once a bug is reported, the initial step for developers is to locate the buggy elements (e.g., source code files/statements) in the CAE projects. However, the task of FL is time-consuming and costly [16, 26, 30, 53, 64, 65]. For example, the Kratos Multiphysics (Kratos) project, a popular CAE project, has 8,488 files and 1,976,009 statements (i.e., lines of code). When locating bugs in the Kratos project, developers typically examine test cases and error messages to debug/locate the problem. This process is non-trivial, since the buggy location may not be clearly reported in the error messages. According to our preliminary investigation, the average time required to fix bugs in some mainstream CAE projects is: 28 days for Fire Dynamics Simulator (FDS), 91 days for Kratos, 149 days for deal.II, and 56 days for Modular Finite Element Methods (MFEM).

Figure 1 illustrates a report about issue#8360 in a CAE project (FDS) where the 3D pyrolysis model does not work properly under certain circumstances. Initially, the user tried to solve this problem by activating the BURN_AWAY option, but the pyrolysis reaction still failed to occur. The developer then determined through code review that the problem stemmed from a poorly placed *if* statement. This process shows that the initial location of the bug may not be accurate, and often requires multiple iterations to accurately find the bug location. In this case, it took seven days to locate the bug. Hence, it would be ideal if the localization task for buggy code in CAE projects could be automated.

However, the majority of automated FL studies focus on bugs in general-purpose software projects [22, 33, 49]. There have been some empirical studies on FL [24, 56], but they mostly use smaller-scale benchmark programs such as Siemens, SIR (Software-artifact Infrastructure Repository), and Defects4J for evaluation [4, 10]. These program collections, due to their lower scale and complexity, may not fully reflect the challenges and needs of FL in large, complex software systems. Although some studies have explored the effectiveness of FL methods on large software

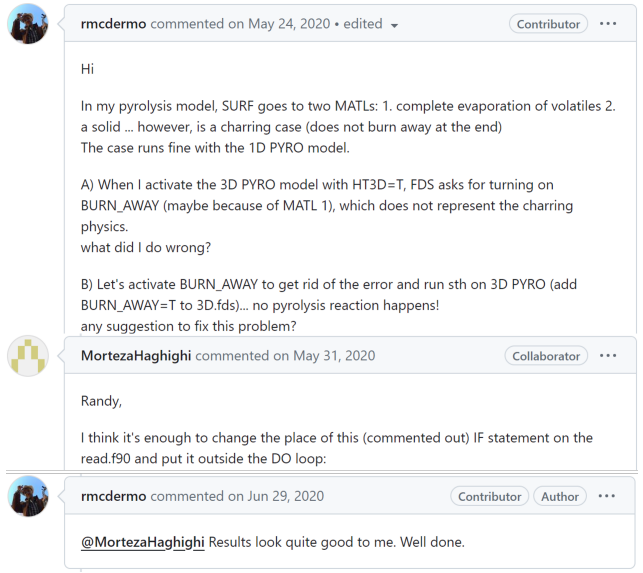


Fig. 1. The Process of Locating and Fixing a CAE Bug.

repositories [19, 28], these studies are generally restricted to a limited type of FL methods, and their studied repositories differ significantly from CAE projects.

Due to the interdisciplinary nature of CAE software, the bugs in CAE projects exhibit unique characteristics. (1) Compared to traditional software, CAE software typically adopts an iterative computation paradigm to realize various numerical simulations [7, 8, 13]. Consequently, similar code areas tend to be reached within repeated loops, regardless of different inputs. In this context, coverage may not be able to provide sufficient guidance for FL methods. (2) In addition, long execution cycles may cause multiple-file bugs to overlap each other, increasing the complexity of fault diagnosis. (3) The input parameters of CAE software are complex and diverse, making it more difficult to write test cases. (4) Meanwhile, CAE software often involves fine meshing and specific numerical algorithms, which not only increases the uncertainty and inaccuracy of simulation, but also brings greater challenges to the accuracy of FL methods. Therefore, there remains a question: *Can existing FL methods be applied to effectively and efficiently locate bugs in CAE projects*? To explore this, we frame the following research questions (RQs):

RQ1: *How effective do FL methods perform in CAE projects*?

RQ2: *How efficient do FL methods perform in CAE projects*?

RQ3: *How do single-file and multiple-file bugs in CAE projects affect the effectiveness of FL methods*?

The effectiveness and efficiency of FL are particularly important in CAE projects due to the high computational cost and time-consuming simulations. In RQ1 and RQ2, we aim to assess how effectively and cost-effectively these methods can locate bugs. RQ3 explores how the presence of single-file and multiple-file bugs affects the effectiveness of FL methods. Understanding this impact is critical for CAE projects with a high proportion of multiple-file bugs.

To answer these questions, this study conducts a systematic empirical analysis on the effectiveness of FL methods for real-world bugs for CAE projects. Due to the lack of publicly available datasets, we manually reproduce 76 real-world bugs from three widely used open-source CAE projects, namely

FDS[1], deal.II[2], and MFEM[3]. Then, we precisely label the code elements causing these bugs, and collect coverage information when the bugs are triggered. To automatically locate these bugs, we implement six state-of-the-art FL methods (i.e., Rogot2, Bugspots, Metallaxis, MUSE, SmartFL, and DeepFL), which belong to five different categories. By feeding these methods the necessary coverage information, a ranked list of suspicious code elements is generated for further investigation.

With comprehensive empirical analysis, a series of insights are gained into the effectiveness and efficiency of FL methods for CAE projects:

- The learning-based localization method (i.e., DeepFL) performs better overall than other FL methods. However, project developers still need to check 467.18 statements or 4.03 files to find buggy code elements, which are 23 times higher than those required for other general-purpose software.
- When locating CAE project bugs, there is an obvious trade-off between FL accuracy and efficiency. Many effective FL methods are slow to locate a bug (e.g., taking 208.13 hours), which is significantly higher than other software (e.g., Defect4J, Siemens, and SIR).
- The proportion of multiple-file bugs in CAE projects cannot be ignored. This type of bug is difficult to locate and requires 4.41 more files to be checked than single-file bugs.

The aforementioned findings highlight the need for CAE projects to design fine-grained, efficient, and domain-specific FL methods. Particularly, we observe that the majority of bugs are concentrated in "main module" files closely associated with CAE functionalities. This observation leads us to explore the potential relationship between the importance of files and the likelihood of fault occurrence. Inspired by this insight, we propose a set of CAE main-module-based features that improve FL accuracy by assessing the similarity between files and the main module documents. By integrating these new features into a learning-based FL method (i.e., DeepFL), we are able to improve *Mean Average Rank* (*MAR*) and *Mean First Rank* (*MFR*) by 35.93% and 45.00%, respectively.

In summary, the main contributions of this work are:

- To the best of our knowledge, this is the first study to investigate the feasibility of applying FL methods to CAE projects, which represents our initial exploration in examining FL issues within industrial software.
- Inspired by our findings, we improve FL accuracy by main-module-based features, which has successfully identified an additional 9.68% of bugs in the *Top-1* metric in CAE projects.
- We have collected 76 real-world CAE project bugs, and have made our dataset publicly available[4]. This dataset fosters further research and collaboration on CAE projects within the software engineering community.

Paper Organization. Section 2 introduces the characteristics of CAE and the main ideas of FL. Section 3 details the workflow of the empirical study, while Section 4 presents the experimental results. Section 5 explores the lessons learned and suggestions for future research. Section 6 introduces the improved FL method based on the research suggestions. Sections 7 and 8 discuss threats to validity and related work, respectively. Finally, Section 9 concludes this paper.

2 BACKGROUND

In this section, we explain the workflow and the characteristics of CAE, as well as the main idea of FL.

[1]https://github.com/firemodels/fds
[2]https://github.com/dealii/dealii
[3]https://github.com/mfem/mfem
[4]https://figshare.com/s/7e7f4b08240b2c215d67

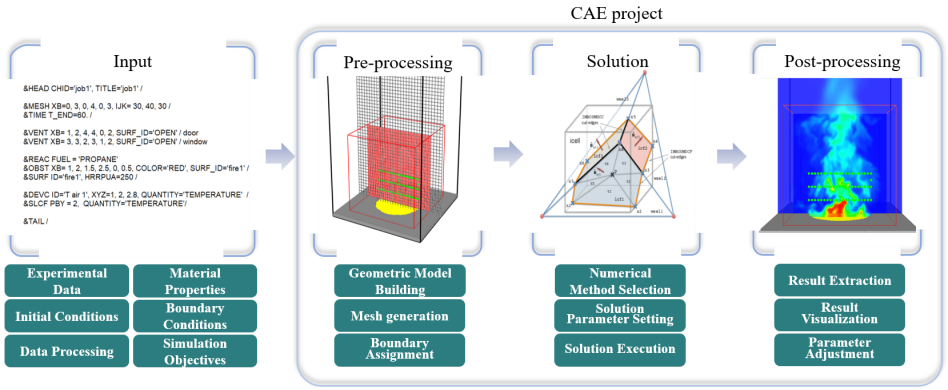


Fig. 2. Workflow of CAE Projects.

2.1 Workflow of CAE

CAE utilizes computer software to assist engineering analysis tasks, supporting a wide range of engineering applications from CAD to CAM. CAE software is commonly applied to perform complex mathematical analyses such as finite element analysis (FEA), computational fluid dynamics (CFD), and multibody dynamics (MBD). The workflow and main modules of CAE projects are shown in Figure 2, including input, pre-processing, solution, and post-processing.

The first step of CAE is to collect and prepare all necessary input data for simulation, which includes setting initial conditions, defining physical and material properties, and establishing boundary conditions. Additionally, the objectives of the simulation are defined to determine the physical phenomena to be studied, the scale of simulation, and the expected outcomes.

Following this, the process moves into pre-processing. Here, the geometric model is built and divided into small, finite elements or volume elements for numerical calculations. The mesh density and type can be adjusted based on the accuracy requirements and the available computational resources. Material properties and boundary conditions are then assigned to different regions of the model.

The next stage is named “solution”, which selects appropriate numerical methods, and executes the solving and computing process. Numerical methods like Finite Element Method (FEM) and Finite Volume Method (FVM) are chosen based on the specific requirements. Dynamic simulations necessitate the selection of suitable time steps and iterative solvers to assure numerical stability and precision. Upon the termination of simulation, the actual numerical solution is carried out, where key parameters are diligently monitored.

The final step is post-processing. In this step, computed results such as temperature distribution, velocity fields, and stress distributions, are extracted and presented in graphical or tabular form for detailed analysis and interpretation. Verification and calibration activities are conducted by comparing the simulation results with experimental data or theoretical predictions to confirm their accuracy. Adjustments or calibrations to the model or parameters may be necessary based on the results. This comprehensive method ensures the reliability and applicability of CAE software in various engineering scenarios.

In CAE projects, several main modules support the aforementioned steps, such as mesh generation, condition boundary assignment, numerical method selection, as presented in Figure 2.

These main modules are often specified in the official CAE module documentation (e.g., the official introduction of the MFEM project[5]).

Typical CAE projects include FDS, deal.II, and MFEM. FDS is a project for large-eddy simulation of low-speed flows focusing on the generation of smoke and heat transfer by fires. deal.II is a versatile C++ library for finite element methods, supporting adaptive mesh refinement and parallel computations. MFEM is a lightweight and scalable C++ library designed for high-performance simulation using FEM.

2.2 Characteristics of CAE Projects

Based on the general structure above, we have identified the following common characteristics of CAE projects.

CAE software adopts an iterative computing paradigm. Most CAE projects are typically designed to simulate evolving physical phenomena over time, necessitating the specification of a simulation duration. This process involves the iterative solving of physical equations, where each iteration builds upon the results of the previous one [43]. Due to the iterative nature of CAE, the variations in coverage for program execution on different inputs may not be significant, complicating the task of pinpointing exact bug locations through coverage information. Developers must, therefore, discern and comprehend the subtle effects that accumulate over lengthy iterations, which adds complexity to the FL process.

CAE software typically requires extensive, long-term computational efforts. Simulations involve a large number of physical equations and a huge amount of computation [18]. For instance, FDS requires at least approximately 82 hours to compute a relatively simple candle burning problem. FL methods, such as mutation-based FL, often require the execution of a large number of variants, which can significantly lengthen the diagnostic process. During long execution cycles, multiple-file bugs may overlap each other, which increases the complexity of fault diagnosis.

The inputs of CAE software are complex. Unlike general-purpose software, the input parameters of CAE software are more diverse and complex [52]. For example, there are over 600 parameters for FDS, which define the simulation scenario and conditions. Even a small-scale simulation scenario requires more than 20 parameters. Especially, some parameters are deeply influenced by specific domain knowledge. The meanings, acceptable value ranges, and inter-dependencies of these parameters may not be intuitive to users. The input complexity increases the difficulty of writing test cases for CAE software, which leads to an insufficient number of test cases in the current CAE field. Consequently, the scarcity impacts the efficacy of FL methods that depend on test cases.

Simulations in CAE software may be inaccurate. The accuracy of simulations largely depends on the quality and subdivision of the mesh. The finer the mesh, the more accurate the simulation results tend to be. Accordingly, the computational cost also significantly increases. Moreover, CAE software adopts specific numerical algorithms to solve physical equations. These algorithms may perform poorly under certain conditions, leading to inaccuracy in simulation results [21]. Under such circumstances, false positive FL suggestions provided by FL methods may have more negative impact on CAE developers, since deeper understanding and evaluation on the inaccurate results are required.

As a brief summary, the characteristics of CAE software pose great challenges for locating bugs in CAE software, in that various factors have to be considered such as the physical process of the

[5]https://mfem.org/

Table 1. Example Code for Fault Localization

	Code snippet with a bug at s_7	$a_0 = 0$	$a_1 = 1$	$a_2 = 2$	N_{CS}	N_{CF}	N_{US}	N_{UF}	Score	Ranking
s_1	input(a)	•	•	•	2	1	0	0	0.33	3
s_2	i = 1;	•	•	•	2	1	0	0	0.33	3
s_3	sum = 0;	•	•	•	2	1	0	0	0.33	3
s_4	product = 1;	•	•	•	2	1	0	0	0.33	3
s_5	if (i <a) {	•	•	•	2	1	0	0	0.33	3
s_6	sum = sum + i;			•	0	1	2	0	0.83	1
s_7	product = product × i;			•	0	1	2	0	0.83	1
	// *bug: product = product × 2i*									
s_8	} else	•	•		2	0	0	1	0	9
s_9	sum = sum - i;	•	•		2	0	0	1	0	9
s_{10}	product = product / i;	•	•		2	0	0	1	0	9
s_{11}	}	•	•		2	0	0	1	0	9
s_{12}	print(sum);	•	•	•	2	1	0	0	0.33	3
	Execution Results	Success	Success	Fail						

simulation, the applicability of the algorithm, and the complexity of the input parameters. Hence, it is important to evaluate the effectiveness of applying automated FL over CAE projects.

2.3 Automated fault localization

FL methods are vital in software engineering. By examining the coverage differences between passing and failing test cases, FL methods assign a suspiciousness score to each code element, thus pinpointing the most likely sources of bugs.

Take Table 1 as an example, we shall explain the idea of automated FL. In the table, there is a bug on line 7, that is, s_7 should be product × $2i$. Suppose we have two passing test cases a_0 and a_1, and one failing test case a_2. By analyzing the code coverage, we can identify the statements executed in each test case (indicated by the dots in columns 3–5). For example, for a_0, the value of a_0 (=0) is less than i (=1), so the lines from s_6 to s_7 are not executed. The principle of FL is that statements covered by multiple failing test cases are more suspicious than those covered by passing ones. To quantify this, we calculate four coverage metrics, namely N_{CS}, N_{CF}, N_{US}, and N_{UF}, where N_{CS} and N_{CF} are the number of passing and failing test cases covering a statement, and N_{US} and N_{UF} are the number of passing and failing test cases not covering a statement, respectively. Using these coverage metrics, we calculate the suspiciousness score (Score) of statements based on the Rogot2 formula and rank them according to their scores (Ranking).

Among all FL methods, spectrum-based FL (SBFL) is the most extensively researched and evaluated technique. SBFL relies on spectrum coverage information and the execution results of test cases to identify potential bugs in the code [54, 61, 62, 67]. In this method, a code element is often considered more suspicious if it is frequently executed in failing test cases and less in passing ones. Previous studies have shown that at least 37 different techniques for calculating suspiciousness have been proposed [19]. One of the state-of-the-art suspiciousness techniques is Rogot2, defined as $\frac{1}{4}\left(\frac{N_{CF}}{N_{CF}+N_{CS}}+\frac{N_{CF}}{N_{CF}+N_{US}}+\frac{N_{US}}{N_{US}+N_{CS}}+\frac{N_{US}}{N_{US}+N_{UF}}\right)$. In this formula, a statement will have a high suspiciousness score if it is only covered by failing test cases (i.e., $N_{CF} > 0$) and not covered by any passing test cases (i.e., $N_{US} = 0$). For example, in Table 1, test cases a_0, a_1, and a_2 all executed statement s_1. Therefore, for s_1, $N_{CS} = 2$, $N_{CF} = 1$, $N_{US} = 0$, and $N_{UF} = 0$ (see columns 6, 7, 8, and 9). Using the Rogot2 formula, the suspiciousness score for s_1 is calculated as 0.33 (see column 10), and

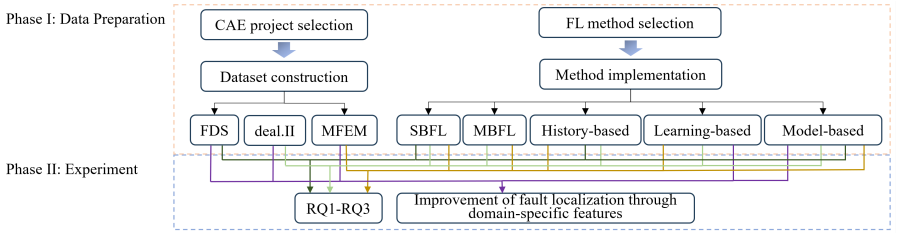


Fig. 3. The Workflow of the Empirical Study.

the same procedure is followed for the other statements. Finally, statements s_6 and s_7 are ranked as the most suspicious (see the last column).

Beyond SBFL, the field of FL has developed diverse methods that harness various types of information to enhance the accuracy and efficiency of identifying bug locations within software systems. Mutation-based FL (MBFL) evaluates the effects of intentional code changes on test results, and it pinpoints specific code elements that influence software behavior [59]. History-based FL considers the bug-fix history of code elements, hypothesizing that elements frequently altered in the past are more likely to contain faults [29, 45]. Learning-based FL utilizes machine learning to analyze complex data patterns and historical bug information [31, 33, 37, 44]. Together, these methods form a powerful FL method family that utilizes different data to improve the accuracy and efficiency of FL in software development.

3 WORKFLOW OF EMPIRICAL STUDY

This study investigates three research questions. First, CAE projects often require the coupling of multiple physical fields, intensive numerical calculations, and complex input and output processing, which increases the difficulty and complexity of FL. Therefore, RQ1 explores the effectiveness of different FL methods in CAE projects. Second, the efficiency of FL methods determines the time developers wait for getting a ranking list of suspicious code elements. Ideally, this waiting time should be minimized to avoid unnecessary delays for fixing bugs. However, the long-term computational nature of CAE projects means that re-executing or debugging CAE projects during FL can be very time-consuming, which poses a challenge to efficiently fix bugs. Therefore, RQ2 studies the efficiency of different phases of FL on CAE projects. Third, when dealing with single-file bugs, like an error in calculating the elastic modulus in MaterialProperties.cpp, the developer only needs to check this file to locate and fix the issue. In contrast, in multiple-file bugs, an incorrect elastic modulus defined in MaterialProperties.cpp will lead to errors in displacement calculations in FiniteElementAnalysis.cpp and dynamic load calculations in DynamicLoad.cpp. Developers need to trace data flows and dependencies across multiple modules, which increases the difficulty of fault localization. Therefore, RQ3 studies the effectiveness of different FL methods on single-file bugs and multiple-file bugs.

Figure 3 shows the two phases of the empirical study. The first phase is data preparation, which is detailed in Subsections 3.1 to 3.4. The second phase involves conducting experiments to address the RQs, which is extensively covered in Sections 4 and 6. In particular, in Figure 3, the data and FL methods employed in each experiment are specified by various colored lines.

3.1 CAE Project Selection

To conduct our experiment, we have surveyed 62 projects in the field of CAE provided by CFD Support[6]. The project list is established by experts with a deep-rooted foundation in CAE. This background ensures that the CAE projects provided are both sophisticated and aligned with the latest industry standards, making them an ideal resource for research.

Eirini et al. [27] suggest researchers should filter out projects that are not suitable for analysis. Such unsuitable projects may only record minimal development activity or are used only for personal purposes. Therefore, we apply the following filtering rules:

- Open-source: We focus on projects hosted on open-source platforms because they readily provide access to development histories and community interactions. Currently, there are many such platforms, including GitHub, OpenLB, and Bitbucket. Notably, OpenLB is not a general code hosting platform but is limited to support for specific projects; Bitbucket often features a smaller number of third-party apps and integrations. Therefore, in this study, we primarily focus on GitHub. Compared to other open-source platforms, the increase in contributor numbers, enhanced community engagement, and visibility of activities of GitHub have led to greater participation and more review and feedback opportunities from the community [38].
- Duration: The project must contain at least 50 weeks of software development activity [5]. Prolonged development periods allow for the accumulation of diverse bug data and software revisions, providing a powerful dataset for locating bugs.
- Issues: The project must have more than 100 issues, as a lower number of issues indicates lower project activity and community engagement, which also affects the effective collection and reproduction of bugs [27].

After applying these criteria, the top five projects are OpenModelica, FDS, deal.II, MFEM, and Kratos. OpenModelica primarily uses the Modelica language, an object-oriented, declarative modeling language designed for multi-domain physical system modeling and simulation, such as electromechanical systems and thermo-fluid dynamics. However, Modelica lacks well-established mutation testing and instrumentation tools, which are essential for our experiments requiring extensive mutation testing and detailed code instrumentation. Given these limitations, we have decided to exclude OpenModelica from our selection and instead focus on FDS, deal.II, and MFEM, which offer better support for our experimental requirements.

As mentioned above, these projects are quite diverse, with different application areas, scales, programming languages, code complexity and maturity, which can help us evaluate FL more comprehensively.

- **Diversity in Application Domains:** FDS is widely used in fire safety engineering, allowing us to examine a project centered on computational fluid dynamics (CFD) and fire modeling. MFEM is a high-performance finite element method (FEM) library, supporting various scientific computing applications, from fluid dynamics to electromagnetics. deal.II stands out for its wide applicability across engineering fields such as solid mechanics, fluid mechanics, and thermal analysis.
- **Variety in Programming Languages:** FDS is developed in Fortran, a language traditionally associated with scientific computing and high-performance numerical simulations. Fortran remains a prominent choice in legacy systems, especially in projects that involve extensive numerical computation, like fluid dynamics and climate models. MFEM and deal.II are both written in C++, a language known for its performance in high-computation environments

[6]https://www.cfdsupport.com/cae-open-source-software.html

and its widespread use in modern CAE projects. This selection enables us to observe the advantages, disadvantages, and impact of various programming languages (particularly popular ones) on fault localization.

- **Varied Code Complexity and Maturity**: FDS, MFEM, and deal.II span a wide range of source lines of code (SLOC), from 147,657 for FDS to over 2.3 million for deal.II. This variation in codebase size offers us the opportunity to explore the scalability of FL methods. Projects like deal.II with larger codebases help assess whether FL techniques can handle extensive, highly modularized systems, while smaller, more focused projects like FDS allow us to study the efficiency of fault localization in more concise but computationally intensive environments. MFEM has one of the longest continuous development cycles among the available projects. The extended history offers insights into how bugs evolve and how fault localization performance might vary as a project matures.

Table 2 provides a comprehensive statistical summary of all the projects, detailing aspects such as the number of test cases, issues, lines of code, programming languages, data collection periods, code line distribution by category, and project descriptions.

Currently, the FDS, MFEM, and deal.II projects have accumulated 4,263, 1,974, and 3,315 issue reports, respectively, in their code repositories. The number of test cases ranges from 1,336 to 2,057, with MFEM experiencing a 220% increase in test cases after April 12, 2019, FDS seeing a 22% increase after November 19, 2021, and deal.II showing a 12% increase after March 21, 2020. This growth in test cases over time signifies active community involvement and ongoing testing efforts. The types and distribution of bugs vary across the projects. In FDS, bugs are predominantly associated with abnormal behaviors in specific physical simulation scenarios, occurring in 100% of the reported cases. Addressing these issues often requires a deep understanding of FDS's internal mechanics and the specific application contexts, highlighting the potential challenges FDS faces in managing complex simulation tasks effectively. For MFEM, 38% of reported bugs involve multiple issues, typically resulting from interactions between various software components. These complex, interdependent problems suggest that MFEM faces considerable challenges in ensuring smooth integration and robust interaction among its modules, particularly in scenarios involving intricate simulations. In the case of deal.II, 37.5% of the bugs are linked to data synchronization and distribution challenges in parallel computing environments. These issues frequently arise when managing distributed data across multiple processors, indicating significant hurdles for deal.II in achieving reliable data synchronization and consistency in large-scale simulations. Furthermore, problems related to the allocation and management of degrees of freedom (DoF), accounting for 25% of reported bugs, underscore additional difficulties in efficiently managing resources and maintaining accuracy in complex computational settings. In these three projects, users and testers provide feedback on bugs, new feature requests, and development suggestions through issue reports. The project development teams classify and tag these issue reports with labels like "bug", "feature", and "cmake", to distinguish the nature and purpose of each issue report.

3.2 Dataset Construction

To construct the FL dataset for CAE projects, the first step is to set up the development and testing environments for the FDS, MFEM, and deal.II projects. FDS relies on various libraries and external tools (such as MPI for parallel computing). During configuration, it is essential to ensure that FDS integrates correctly with system dependencies such as Fortran compilers and MPI libraries. Proper versions of libraries like HDF5 and OpenMPI must be installed, and the parallel computing environment should be adequately configured. MFEM depends on C++ compilers and several numerical computation libraries (such as LAPACK, BLAS, and MPI). To reproduce

Table 2. Project Details

Metrics	FDS	MFEM	deal.II
Test cases	2057	1336	882
Issues	4263	1974	3315
Lines of Code (KLOC)	148	188	2391
Programming Languages	Fortran	C++	C++
Data Collection Period	2016-2024	2016-2024	2016-2024
Code Line Distribution by Category	Definition of geometric model and boundary conditions (geom.f90): 27,115 Fluid Dynamics (ccib.f90): 23,508 Reading input files (read.f90): 15,549	FEM core algorithms (fem): 119,774 Auxiliary functions (general): 15,723 Manage finite element meshes (mesh): 43,310	Mesh management (grid): 62,355 Finite Element Method (fe): 52,713 Core modules (base): 49,263
Description	Large-eddy simulation of low-speed flows focused on smoke and heat transfer by fires.	Lightweight C++ library for high-performance simulation using FEM.	Versatile C++ library for finite element methods, supporting adaptive mesh refinement and parallel computations.

bugs in MFEM, it's crucial to ensure that all dependencies are correctly installed, with parallel or non-parallel computation modes enabled based on specific experimental requirements. deal.II is a complex finite element computation library, often integrated with Trilinos, PETSc, and p4est, among others. Ensuring the compatibility and correct linking of these dependencies with deal.II is a key prerequisite.

To ensure consistency and accuracy in the bug reproduction process, we adopt strict filtering criteria to collect fault information. First, following the previous study [50], we select issue reports that included keywords like "bug", "issue", "problem", "error", "fix", or "solve" in their commit messages. This initial filter reduces the number of relevant issue reports for FDS, deal.II, and MFEM to 980, 365, and 186, respectively. This significant reduction is expected as it removes unrelated or trivial issues.

Subsequently, we prioritize the issue reports that provide test cases or detailed steps for constructing test cases, as these reports ensure repeatable and consistent results across different testers. This further filtering reduces the number of issue reports for FDS, deal.II, and MFEM to 159, 76, and 83, respectively. The motivation of such prioritization can be attributed to the complexity of CAE projects, i.e., users may file issue reports without fully understanding how to use the software systematically, resulting in issue reports lacking relevant test cases. Additionally, problems caused by specific hardware configurations, operating systems, or dependencies are also less likely to include test cases.

Lastly, we focus on those issue reports that have already been fixed, to accurately identify the code containing bugs. Following the existing work, we treat the commits that fixed bugs as clean versions, and those before them as versions containing bugs [50, 51]. In this process, the non-bug-fixing issue reports are discarded. For instance, some commits with fixes are not merged into the main branch due to conflicts with ongoing development or decisions to prioritize other features over bug fixes. Some issues are resolved by only updating documentation to clarify usage or correct misunderstandings. Considering the time cost, simulations that time out are also discarded.

During the dataset construction process, reproducing bugs in CAE projects involves challenges on multiple levels. The operation of CAE software relies on the integration of complex physical, engineering, and mathematical models, and the detailed and precise inputs required by these models can vary widely. For example, FDS has up to 600 different input parameters. This complexity requires precise configuration of the simulation environment to ensure accurate reproduction of the reported issues. Additionally, small differences in environment settings or unaccounted changes between the issue report and the test settings can prolong the reproducibility process. For example, differences in operating systems, version conflicts in dependent libraries, and differences in hardware configurations may affect the results of simulation. Moreover, the long computational nature of CAE projects further lengthens the reproduction process. In the end, it took us five months to reproduce all the bugs.

Eventually, we collect 31, 11, and 34 bugs from FDS, deal.II, and MFEM, respectively. The sizes of these two project datasets are comparable with existing FL datasets. For example, Dao et al. utilized data from five projects (Chart, Math, Mockito, Time, and Lang) in Defects4J, with an average of 34.8 real faults per project [14].

Following the previous research [34, 39, 47], we identify buggy statements as those modified in the Git diffs corresponding to each commit. The files containing these altered statements are designated as buggy files. For each bug, we execute failing test cases on the buggy version of the CAE projects, while also running passing test cases written by the project developers, to obtain code coverage information for each test case. Additionally, we collect the bug-fixing history of each code element in that specific version. Therefore, each bug is associated with the corresponding buggy CAE project version, code coverage information, bug-fixing history, and the specific bug location.

3.3 FL Method Selection

According to the empirical study by Zou et al. [67], FL methods are categorized into seven types, including SBFL, MBFL, dynamic program slicing [6, 48], stack trace analysis [55, 57], predicate switching [63], information-retrieval-based FL (IR-based FL) [66], and history-based FL. Our focus is on test case-based FL methods; therefore, we exclude stack trace analysis and IR-based FL, which primarily rely on issue report information. Given that predicate switching is less commonly used, we choose to employ the more popular learning-based FL methods. Additionally, we observe a lack of suitable open-source projects for slicing-based FL. Since the number of model-based papers ranks in the top three [56], we additionally introduce model-based methods. Based on these considerations, we select five FL method categories for in-depth experimentation: SBFL, MBFL, history-based FL, model-based, and learning-based FL. Table 3 shows these methods and their input types.

SBFL is the most widely studied and evaluated FL method. The idea of SBFL methods has been explained in Section 2.3. Among the different suspicious degree techniques, Rogot2 has the best overall performance (including accuracy and robustness) on our dataset. We finally chose Rogot2 in our experiments[7].

MBFL evaluates the impact of small changes on test results by introducing mutations into the code. The method generates mutants by changing code elements and evaluates how these mutations affect test results, allowing for more precise pinpointing of faults. Among MBFL methods, MUSE [40] and Metallaxis [41] are the two most widely studied. These methods mutate each statement using mutation operators to generate mutant code fragments. They aggregate the suspiciousness scores

[7]A comparison of the accuracy of different suspicious degree techniques on our dataset can be seen in the supplementary materials.

Table 3. A List of FL Methods Used in Experiment

Methods	Category	Input types
Rogot2	SBFL	Test coverage
MUSE	MBFL	Test results from mutating the program
Metallaxis	MBFL	Test results from mutating the program
Bugspots	History-based	Development history
SmartFL	Model-based	Static analysis results(control dependencies), dynamic analysis results(data dependency) and test results
DeepFL	Learning-based	Various suspiciousness-value-based, fault-proneness-based, and textual-similarity-based features

calculated by spectrum-based methods with the number of passing and failing test cases affected by the mutant code fragments (i.e., from pass to fail or from fail to pass) and rank the buggy elements.

History-based FL is particularly suitable for large projects requiring long-term maintanence, and is therefore suitable for CAE projects. This method leverages the bug fix history of code elements, operating on the premise that areas that are frequently modified to address bugs are more likely to hide future defects. We adopt the latest history-based FL method - Bugspots [45], which is based on the "hotspot" analysis concept. Its core assumption is that bugs tend to cluster in specific areas of the codebase, which would show frequent modifications in the version control system's commit history.

Model-based fault localization methods construct models and infer fault locations by analyzing a program's structure, control flow, data dependencies, and abnormal behaviors. SmartFL, as an advanced model-based FL technique, probabilistically models the propagation of errors through program dependencies. Unlike traditional approaches that either overlook or fully model program semantics, SmartFL finds an optimal balance between accuracy and scalability [60]. It achieves this by selectively modeling program behavior using a probabilistic graphical model, capturing essential aspects of program semantics without the computational overhead of complete modeling.

Learning-based FL represents the frontier of FL methods, leveraging machine learning to handle complex data patterns found in software systems. We attempted to reproduce some state-of-the-art learning-based methods: DEEPRL4FL (unpublished source code) [33] and GNet4FL (incomplete source code provided) [44]. Therefore, we finally chose to use DeepFL [31]. DeepFL converts code into multidimensional feature vectors, encapsulating aspects such as code complexity, historical data, and mutation effects. The method uses a multi-layer perceptron (MLP) to learn these features, and predict where bugs are most likely to occur.

3.4 Implementation Details

The implementation of the FL method is divided into two main phases: data collection and ranking. The data collection phase involves collecting all relevant data required for FL, such as the code coverage information and commit history. The ranking phase analyzes this information and assigns suspiciousness scores to different code elements.

Common levels of granularity for program elements are statements, methods, and files. The FL methods used in this study consider different levels of granularity. Rogot2, Metallaxis, and MUSE operate at the statement level; Bugspots operates at the file level, and DeepFL operates at the method level. We choose file-level and statement-level granularity, because file-level FL can quickly

narrow down the possible scope of bugs, while statement-level FL can provide more precise bug locations. To enable conversion between these granularities, our study follows practices established by previous research [50, 67]. When converting from coarse-grained to fine-grained, we extend the file-level suspiciousness score to all executable statements and methods in the file. For example, if Bugspots assigns a file a suspiciousness score of 0.7, then every executable statement and method in that file inherits a score of 0.7. Instead, to move from fine-grained to coarse-grained levels, we adopt the maximal method. For example, the suspiciousness score of a file or method can be derived from the maximum suspiciousness score of its constituent statements.

Spectrum-based localization. We use FDS, deal.II, and MFEM projects to evaluate the spectrum-based FL method.

(1) Data Collection: We run the test cases of each bug, and use coverage tools (i.e., gcov and Coverage) to obtain the coverage information of bugs. Considering the time cost and the average runtime, we set a timeout threshold of 1,200 seconds.

(2) Ranking: Following the principles of existing studies [11] and [12], each passing test case should share a similar coverage information (i.e., executed statements) with a given failing test cases. In this way, according to the idea of SBFL, the suspicion of more bug-free files can be reduced. We apply the commonly used Jaccard formula to calculate the similarity between the coverage information of failing and passing test cases. In order to reduce time cost, we filter and retain only the top 100 passing test cases that exhibit the highest similarity to the failing test cases. The similarity is defined as: $\text{Sim}(a, b) = \frac{|\text{Cov}_a \cap \text{Cov}_b|}{|\text{Cov}_a \cup \text{Cov}_b|}$, where $\text{Sim}(a, b)$ represents the similarity of the coverage information of the two test cases a and b, Cov_a and Cov_b respectively represent the statements covered by the two test cases.

Mutation-based localization. MBFL requires creating a series of mutants by mutating the original code. Since the two projects involve different development languages, we needed to find different mutation tools. For the FDS project (developed in Fortran), we identify four mutation tools compatible with Fortran (i.e., PIMS, EXPER, FMS.3, and Mothra) [23]. The most sophisticated of these tools dates back to 1987. Unfortunately, all of them are not working or unobtainable. MFEM and deal.II only uses C++ language, and an open-source mutation tool MuCPP is available[8]. Therefore, regarding MBFL methods, we choose to use MFEM and deal.II projects for evaluation.

(1) Data collection: The collection of coverage information is the same as the SBFL method. We then adopt MuCPP to make subtle adjustments to the original program, thereby generating a series of mutants (mutation programs). The mutation operators include arithmetic operator and conditional operator replacement/insertion/deletion, relational operator replacement, logical operator replacement, and short-cut assignment operator replacement. At last, we execute the mutation program using selected test cases for data collection. Table 4 shows in detail the 12 traditional mutation operators we use.

(2) Ranking: We use Metallaxis and MUSE techniques to calculate the suspiciousness score for each buggy element. The Metallaxis formula is: $S(m) = \frac{\text{failed}\ (m)}{\sqrt{\text{totalfailed} \cdot (\ \text{failed}\ (m) + \text{passed}\ (m))}}$, where totalfailed is the total number of test cases that failed on the original program, failed(m) is the number of test cases that failed on the original program but passed on mutation program m, passed(m) represents the number of test cases that passed on the original program but failed on mutation program m. The suspiciousness score of a buggy element (denoted as s) is the maximum suspiciousness score among all mutants of the element. The MUSE formula is: $S(m) = \text{failed}\ (m) - \frac{f2p}{p2f} \cdot \text{passed}(m)$, where failed($m$) and passed($m$) follow the same definition as Metallaxis, and $f2p/p2f$ is the ratio of test cases changing from "fail" to "pass" and from "pass" to "fail". The suspiciousness score of a

[8]https://ucase.uca.es/mucpp/download.html

Table 4. Mutation Operators Employed in MuCPP

ID	Operators	Description
1	ARB	Arithmetic Operator Replacement (Binary: +, -, *, /, %)
2	ARU	Arithmetic Operator Replacement (Unary: +, -)
3	ARS	Arithmetic Operator Replacement (Short-cut: ++, −)
4	AIU	Arithmetic Operator Insertion (Unary: -)
5	AIS	Arithmetic Operator Insertion (Short-cut: ++, −)
6	ADS	Arithmetic Operator Deletion (Short-cut: ++, −)
7	ROR	Relational Operator Replacement (<, ≤, >, ≥, == , ≠, not_eq)
8	COR	Conditional Operator Replacement (&&, and, ‖, or)
9	COI	Conditional Operator Insertion (!, not)
10	COD	Conditional Operator Deletion (!, not)
11	LOR	Logical Operator Replacement (&, |, ^)
12	ASR	Short-Cut Assignment Operator Replacement (-=, +=, *=, /=, %=)

buggy element (denoted as s) is the average of the suspiciousness scores among all mutants of the element.

History-based localization. We use FDS, deal.II, and MFEM projects to evaluate history-based FL methods.

(1) Data Collection: We clone the Git repositories for FDS, deal.II, and MFEM into our local environment.

(2) Ranking: We analyze the Git commit history of each project by running Bugspots. Following the default settings of Bugspots, we search for the keywords "fix" and "bug" in the commit messages to identify commits related to bug repairs. Additionally, Bugspots particularly emphasizes the importance of recent bug repair commits, and assigns higher weights to these commits. Ultimately, Bugspots combines a time decay factor with the modification history of each file, and ranks all buggy elements with a unique "Bugspot score" for each file.

Model-based localization. We use FDS, deal.II, and MFEM projects to evaluate model-based FL methods.

(1) Data Collection: We start by instrumenting the programs to collect execution traces from both passing and failing test cases. These traces capture detailed information on instruction execution sequences, including memory reads/writes, control flow transitions, and program state changes. We then applies both static and dynamic analyses to build a comprehensive dependency model of the program. Static analysis focuses on identifying control dependencies within the program, determining how conditional branches and control structures influence statement execution. Dynamic analysis leverages the execution traces to map data dependencies, tracking error propagation through variables and memory locations by identifying how read and write operations interact. These combined analyses enable SmartFL to construct a probabilistic graphical model (factor graph) that captures both control and data dependencies.

(2) Ranking: The test case outcomes (i.e., passing or failing) are fed into the probabilistic model as evidence. SmartFL performs marginal inference on the factor graph to compute the likelihood of each statement being faulty. The statements are then ranked based on their calculated fault probabilities, with higher probabilities indicating greater suspicion of fault. The resulting ranked list is provided as the fault localization outcome.

Learning-based localization. DeepFL requires the suspiciousness values calculated by MBFL methods as feature inputs. Since FDS is not considered for MBFL experiments, to ensure fair comparison, we use MFEM and deal.II projects to evaluate the learning-based FL method.

(1) Data Collection: DeepFL integrates the following dimensions of fault diagnosis information for localization:

- Spectrum-based features: DeepFL takes the suspiciousness scores computed by 34 SBFL techniques (e.g. Ample, M1, Goodman) as features.
- Mutation-based features: DeepFL collects features by executing Metallaxis and MUSE. Metallaxis utilizes 34 traditional spectrum-based techniques to compute suspiciousness scores. DeepFL follows previous research to collect four different types of test results on 34 Metallaxis scores and one MUSE score, including pass/fail information, exception type, exception message, and stack trace [32]. This gives (34 + 1) × 4 = 140 suspect values. Note that due to the difference in programming languages, we could not obtain the 35 values that require full Java stack trace information. Therefore, we finally obtain 105 mutation-based features.
- Complexity-based features: DeepFL uses 37 complexity features, including 16 Java ASM bytecode instruction statistics and 21 code complexity measures. Since our projects use C++ and Fortran, we only obtain 21 complexity-based features.
- Textual similarity features: DeepFL collects different fields as queries and documents. Query fields come from failed tests, including the name of failed tests, the source code of failed tests, and the complete failure message. Document fields come from source code methods, including the fully qualified name of the method, accessed classes, method invocations, used variables, and comments. DeepFL computes the similarity between queries and documents, and obtains 15 features.

Finally, we use a total of 175 features as input.

(2) Ranking: We use DeepFL to predict potential fault locations by integrating the above 175 features. Following the settings of DeepFL, we perform leave-one-out cross-validation on the faults for each buggy version.

4 EXPERIMENT RESULTS

4.1 RQ1: How effective do FL methods perform in CAE projects

Methodology. We run FL methods on datasets to locate buggy code elements. Following previous studies [42, 67], we use three accuracy-based evaluation metrics (i.e., *Top-N*, *MFR*, and *MAR*) to assess the effectiveness of FL methods at both file level and statement level. *Top-N* counts the exact position of buggy code elements in a ranking list. To provide effective guidance to developers, we set N as 1, 5, 10, 20 [42, 67]. It is noteworthy that even if multiple code elements of a bug are located at *Top-N*, it is only counted once.

$$\textit{Top-N} = \sum_{i=1}^{B} \mathbb{I}\left(\min(R_i) \leq N\right) \tag{1}$$

where B denotes the total number of bugs. R_i represents the set of ranks of all buggy code elements for the i-th bug in the ranking list. $\min(R_i)$ selects the highest-ranked buggy code element for the i-th bug. $\mathbb{I}(\cdot)$ is the indicator function, which returns 1 if $\min(R_i) \leq N$, and 0 otherwise.

MFR calculates the average rank of the first buggy code element in the ranking list for each buggy version.

$$MFR = \frac{1}{B}\sum_{i=1}^{B} \min(R_i) \tag{2}$$

MAR calculates the average rank of all buggy code elements in the ranking list for each buggy version. Higher *Top-N* or lower *MFR* and *MAR* indicate more effective FL.

$$MAR = \frac{1}{B}\sum_{i=1}^{B} \frac{1}{|R_i|}\sum_{r\in R_i} r \tag{3}$$

where $|R_i|$ is the number of buggy code elements in the i-th bug. $\sum_{r\in R_i} r$ represents the sum of the ranks of all buggy code elements for the i-th bug. The inner term $\frac{1}{|R_i|}\sum_{r\in R_i} r$ calculates the average rank of all buggy code elements for the i-th bug.

Let us explain these metrics using two problematic code segments A and B: A has two buggy code elements, and B has one. Suppose after running the FL method, the buggy code elements in A are ranked at the 3rd and 15th position in the list, while the buggy code element in B is ranked at the 6th position. We then calculate the Top-N, *MFR*, and *MAR* values as follows:

Top-1 = 0: none of the buggy code elements appear in the first position. Top-5 = 1: one buggy code element from segment A is ranked within the top 5 (at position 3). Top-10 = 2: both buggy code elements from segment A and segment B are ranked within the top 10 (at positions 3 and 6). Top-20 = 2: buggy code elements from both segments A and B are ranked within the top 20 (at positions 3, 15, and 6). Among these, positions 3 and 15 belong to segment A, but are counted only once. The first buggy code elements in segment A are ranked at the 3rd in the list, while the first buggy code element in segment B is ranked at the 6th position. Thus, we compute *MFR* as $(3+6)/2 = 4.5$. For segment A, the ranks of the buggy code elements are 3 and 15, and for segment B, the rank of the buggy code element is 6. Thus, we compute *MAR* as $((3+15)/2+6)/2 = 7.5$.

Results. As shown in Table 5, FL in CAE projects is far from satisfactory. In these three projects, the minimum *MFR* indicates that project developers need to examine on average 467.18 statements or 4.03 files in a ranking list suggested by FL methods to find the location of buggy code elements. This stands in stark contrast to Defects4J, where developers only need to check an average of 20.32 statements (i.e., *MFR*=20.32) to identify bugs, as shown in prior studies [33]. The 23-fold increase in the number of statements to be checked in CAE projects underscores the significant challenges posed by industrial-scale complexities, such as larger codebases and computationally intensive tests. While Defects4J provides a controlled benchmark for evaluating FL methods, CAE projects highlight the limitations of current techniques and the urgent need for more robust, scalable solutions tailored to real-world, domain-specific environments.

At the statement level, DeepFL outperforms other methods on most evaluation metrics. Although all methods fail to find many bugs within *Top-20*, the *MFR* value of DeepFL is much lower. This means that compared to other methods, DeepFL can place buggy statements at higher positions in the ranking list. The higher accuracy of DeepFL is due to its comprehensive application of different types of coverage information (such as spectrum-based and mutation-based information) to train FL models.

At the file level, according to *Top-N*, Bugspots outperforms other methods on most evaluation metrics; it locates 55% of bugs at the *Top-1* position. By examining the top 20 most suspicious files, developers can find 94% of bugs in the ranking list generated by Bugspots. Bugspots performs well in these CAE projects for the following reason: As shown in Figure 2, CAE projects have the main modules to implement the key functions of numerical simulation (e.g., mesh generation and numerical methods implementation). These main modules are frequently to be modified and likely

Table 5. Effectiveness of FL Methods at Statement and File Levels

Projects	Method	Statement Level						File Level					
		Top-1	*Top-5*	*Top-10*	*Top-20*	*MAR*	*MFR*	*Top-1*	*Top-5*	*Top-10*	*Top-20*	*MAR*	*MFR*
FDS	Rogot2	0	0	0	1	9973.45	9824.08	3	11	23	31	9.56	8.00
	Bugspots	0	0	0	0	1600.83	1049.17	17	26	28	29	5.54	4.03
	SmartFL	0	0	0	0	20314.24	20045.53	4	12	21	29	9.56	8.74
MFEM	Rogot2	0	1	2	3	7801.97	7384.35	1	7	8	9	72.24	71.82
	Metallaxis	0	1	1	2	9226.72	8276.61	1	9	10	12	67.94	61.74
	MUSE	0	1	1	1	9231.05	8236.47	1	8	8	10	70.38	69.62
	Bugspots	0	0	0	0	36718.67	23228.58	8	17	19	25	61.82	28.15
	SmartFL	0	0	1	5	576.88	570.82	1	10	12	14	60.31	59.55
	DeepFL	0	1	1	3	485.46	467.18	1	9	13	14	58.67	54.03
deal.II	Rogot2	0	0	0	0	3243.39	3515.11	1	1	1	1	86.00	85.13
	Metallaxis	0	0	0	0	6294.04	4547.25	0	1	1	1	182.64	161.49
	MUSE	0	0	0	0	8293.75	7274.19	0	0	1	1	190.38	169.62
	Bugspots	0	0	0	0	15251.00	13442.67	0	0	1	3	121.73	113.91
	SmartFL	0	0	0	0	658.07	645.82	0	1	1	2	153.85	136.90
	DeepFL	0	0	0	0	564.86	513.75	1	1	2	3	78.68	71.73

to have bugs. Bugspots happens to mark files that have been frequently modified historically as "hotspot". To understand this reason, we analyze the modification frequency of CAE projects. The results show that the average modification frequencies of the main modules of MFEM, deal.II, and FDS are 4281, 9399, and 4182, respectively, which are 1.0 to 7.5 times more than other modules. Therefore, Bugspots can correctly locate the bugs in such frequently modified modules. However, its reliance on historical modification frequency may not be applicable to all CAE projects, especially those with different development practices or low update frequency of critical modules.

For other FL methods, SmartFL performs closest to DeepFL at both granularities. Metallaxis and MUSE find more bugs at *Top-5* than Rogot2. This is consistent with the conclusion of the previous research: "the accuracy of SBFL methods is often too low to localize faults in large real-world programs" [20]. To understand the reason of the low accuracy of Rogot2 in CAE projects, we use the Jaccard coefficient to calculate the path similarity of test case coverage information, and find that the differences of the code coverage among test cases are small in the FDS, deal.II, and MFEM projects. Let's take a specific issue in the FDS project (issue#6646) as an example. When we compute the similarity between the failing test case and the 100 most similar passing test cases, we have only three distinct similarity values: 0.96, 0.95, and 0.91. On average, 33 test cases have the same similarity value, showing a high degree of consistency. Furthermore, we evaluate the test case density of the projects, i.e., the ratio of the number of test cases to the lines of code. The results show that the test case density is generally low, which is 0.0139, 0.0014, and 0.0071 for FDS, deal.II, and MFEM projects, respectively.

In traditional software fault localization, common features such as code coverage and test cases are widely utilized. However, in CAE projects, the effectiveness of these conventional features is limited due to typically low code coverage and the complexity of test case design. Moreover, CAE projects have unique characteristics—such as iterative computation paradigms, long-term computations, complex input, and simulation inaccuracies—that not only increase the complexity of fault localization but also restrict the applicability of general features. These limitations suggest that relying solely on traditional features may not sufficiently support effective fault localization in

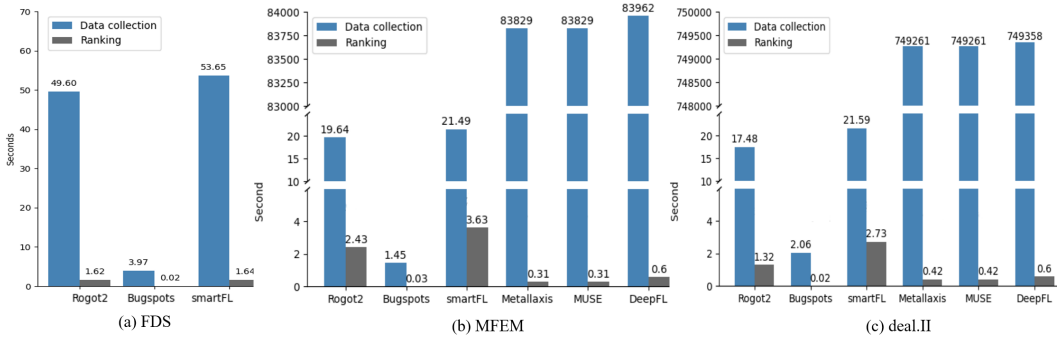


Fig. 4. The Time of Data Collection and Ranking for FL Methods (in Seconds).

CAE. It is crucial to introduce or develop specialized features tailored for CAE projects to enhance the accuracy and efficiency of fault localization.

Conclusion. The current FL method performs unsatisfactorily in CAE projects, requiring developers to check on average 467.18 statements or 4.03 files to locate buggy code elements. At the statement level, DeepFL exhibits high accuracy. At the file level, Bugspots obtains the lowest *MFR* and *MAR* values.

4.2 RQ2: How efficient do FL methods perform in CAE projects

Methodology. We measure the average time for data collection and ranking for each FL method. During the data collection phase, we measure the time spent on collecting the code coverage information, executing test cases on mutants (for mutation-based methods), collecting historical information (for history-based methods), static and dynamic execution information (for model-based), and training neural networks (for learning-based methods). During the ranking phase, we focus on the time required to calculate the scores of suspicious code elements. In this subsection, we present the file-level results, since the statement-level results have the same data collection time and the ranking phase is also similar.

Results. As shown in Figure 4, Bugspots shows the fastest execution speed, which completes the data collection and ranking of suspicious code elements in 1.45 seconds and 0.03 seconds, respectively[9]. Bugspots is particularly effective in CAE projects where key numerical simulation modules (e.g., mesh generation and numerical method implementations) are frequently modified and tend to contain bugs. This enables Bugspots to locate 55% of bugs at the *Top-1* position and 94% within the *Top-20* most suspicious files. However, its reliance on commit history may limit its applicability in projects with less frequent updates or those lacking detailed historical data.

Other methods spend most of the time on data collection. For instance, Metallaxis and MUSE require significant time to generate mutation-based information, which involves systematically applying mutation operators to the codebase. Taking issue#1230 in the MFEM project as an example, MBFL methods generate 2998 mutants using 12 mutation operators. This process is inherently highly demanding on computational resources, especially for large projects like FDS and MFEM, where the data collection time can exceed 208.13 hours (i.e., 749261 seconds).

DeepFL, despite its slower execution time due to the integration of spectrum-based, mutation-based, complexity-based, and textual similarity features, excels in accuracy, particularly at the statement level. By combining multiple coverage features into a multidimensional representation

[9]The execution time presents the average time consumed per fault over the whole dataset.

and training an FL model, DeepFL effectively ranks buggy statements higher in the list. This makes DeepFL a better choice for offline debugging scenarios where accuracy is prioritized over runtime.

In contrast, Rogot2 and SmartFL require moderate execution times as they focus on simpler data collection processes, such as test case coverage or selective program behavior modeling, which are less resource-intensive.

A significant difference from previous research on Defects4J [67], Siemens, and SIR datasets [4] is the long execution time of MBFL and SBFL methods on CAE projects. Although in this study we execute FL methods with more powerful computational resources[10], the execution time of the same MBFL and SBFL methods on CAE projects is 10 to 62 times longer than the reported execution time in the previous study on the other datasets. We analyze the reason as follows. First, the projects used in the previous studies have an average code size of approximately 31–64 KLOC, whereas the CAE projects in this study range from 148 to 2391 KLOC. Larger code sizes directly increase the data collection time for FL methods, particularly those relying on coverage or mutation analysis, as the number of test cases and code elements to process grows exponentially.

Second, CAE projects involve intensive iterative computations for solving physical equations during test case execution. This significantly impacts methods like MBFL and SBFL, which depend on repeated test case executions to collect execution traces or generate mutants. The iterative nature of these computations amplifies runtime costs for every additional test case or mutation, further exacerbating the scalability challenges of these methods.

In addition, our experiments show that memory limitations will not be a bottleneck in our study. For example, in the deal.II project, Rogot2 requires about 180MB, Bugspots requires about 40MB, DeepFL requires about 60MB, and SmartFL requires about 50MB. The variation in memory consumption mainly stems from the difference in the amount of analyzed data rather than inherent method inefficiency. We will also explore memory and energy efficiency optimizations in future work.

Conclusion. Among all FL methods, mutation-based and learning-based FL methods use the longest data collection time due to the complexity (in terms of code size) and the long-term computation characteristic of CAE projects. Bugspots is the fastest FL method, with SmartFL and Rogot2 closely following, while DeepFL is the slowest. The trade-off between accuracy and efficiency is more obvious when locating bugs in CAE projects.

4.3 RQ3: How do single-file and multiple-file bugs in CAE projects affect the effectiveness of FL methods

Methodology. Inspired by the previous research [17, 54], we classify bugs into single-file bugs and multiple-file bugs. Single-file bugs mean that the unusual behavior or bug of programs can be directly traced to a single source code file location. In contrast, multiple-file bugs mean that the bug is caused by a combination of logical errors in multiple files. To ensure consistency and minimize potential bias, we employed a two-step classification process:

Automated Initial Classification: We developed scripts to automatically analyze Git diffs and identify the number of files modified in each commit. Based on this analysis, bugs were initially categorized as single-file or multiple-file bugs.

Manual Verification and Refinement: To enhance classification accuracy, two independent researchers manually reviewed each automatically classified bug. They examined the context of code modifications to confirm whether the bug truly affected a single file or spanned multiple files. In cases of disagreement, a third researcher made the final determination to resolve discrepancies.

[10]Our experiments use a 2.7GHz GHz Intel Xeon Platinum 8374C with DDR4 256GB of memory, while previous research utilizes a 2.3 GHz Intel Pentium-6 CPU with 4GB memory [4].

Table 6. Comparison of Effectiveness of FL Methods on Single-file Bugs (_s) and Multiple-file Bugs (_m)

Projects	Method	*Top-1*	*Top-5*	*Top-10*	*Top-20*	*MAR*	*MFR*
FDS_s	Rogot2	1(4%)	5(22%)	15(65%)	23(100%)	9.43	9.43
	Bugspots	11(48%)	18(78%)	20(87%)	21(91%)	5.00	5.00
	SmartFL	4(17%)	10(43%)	16(70%)	21(91%)	8.96	8.96
FDS_m	Rogot2	2(25%)	6(75%)	8(100%)	8(100%)	9.93	3.88
	Bugspots	6(75%)	8(100%)	8(100%)	8(100%)	7.09	1.25
	SmartFL	0(0%)	2(25%)	5(63%)	8(100%)	11.3	8.12
$MFEM_s$	Rogot2	0(0%)	2(10%)	2(10%)	3(14%)	75.81	75.81
	Metallaxis	1(5%)	6(29%)	6(29%)	7(33%)	52.86	52.86
	MUSE	1(5%)	5(24%)	5(24%)	6(29%)	64.52	64.52
	Bugspots	6(29%)	11(52%)	11(52%)	15(71%)	32.95	32.95
	SmartFL	1(5%)	8(38%)	8(38%)	9(43%)	56.65	56.655
	DeepFL	0(0%)	5(24%)	9(43%)	9(43%)	53.00	53.00
$MFEM_m$	Rogot2	1(8%)	5(38%)	6(46%)	6(46%)	67.85	65.37
	Metallaxis	0(0%)	3(23%)	4(31%)	5(38%)	123.91	81.46
	MUSE	0(0%)	3(23%)	3(23%)	4(31%)	136.12	96.37
	Bugspots	2(15%)	6(46%)	8(62%)	10(77%)	108.46	20.38
	SmartFL	0(0%)	2(15%)	4(31%)	5(38%)	64.76	64.00
	DeepFL	1(8%)	4(31%)	4(31%)	5(38%)	60.89	57.41
$deal.II_s$	Rogot2	0(0%)	0(0%)	0(0%)	0(0%)	100.32	100.32
	Metallaxis	0(0%)	0(0%)	0(0%)	0(0%)	191.05	191.05
	MUSE	0(0%)	0(0%)	0(0%)	0(0%)	196.80	196.80
	Bugspots	0(0%)	0(0%)	1(25%)	1(25%)	110.80	110.80
	SmartFL	0(0%)	1(25%)	1(25%)	1(25%)	133.05	133.05
	DeepFL	1(25%)	1(25%)	1(25%)	2(50%)	69.52	69.52
$deal.II_m$	Rogot2	1(14%)	1(14%)	1(14%)	1(14%)	80.83	82.41
	Metallaxis	0(0%)	1(14%)	1(14%)	1(14%)	179.80	159.63
	MUSE	0(0%)	0(0%)	1(14%)	1(14%)	188.62	167.75
	Bugspots	0(0%)	0(0%)	0(0%)	2(29%)	137.67	115.67
	SmartFL	0(0%)	0(0%)	0(0%)	1(14%)	157.72	140.25
	DeepFL	0(0%)	0(0%)	1(14%)	1(14%)	81.04	73.06

As a result, we identified 23 single-file bugs and 8 multiple-file bugs in the FDS project, 4 single-file bugs and 7 multiple-file bugs in the deal.II project, and 21 single-file bugs and 13 multiple-file bugs in the MFEM project. In this RQ, we assess the effectiveness of FL methods on the file level, because the differences between the two types of bugs occur at this level.

Results. Table 6 shows the impact of single-file bugs and multiple-file bugs on FL methods in CAE projects. In the FDS project, the evaluation metrics in the multiple-file bug scenario are generally better than those in the single-file bug scenario. This may be because the majority of bugs in the FDS project are concentrated in a module named "source", and bugs in the same module are easier to locate. The "source" module exhibits high code coverage across test cases, and its functionalities are relatively cohesive compared to other modules. Consequently, faults within this module tend to share similar execution paths, making them easier to localize even when they span multiple files. However, this pattern appears to be specific to FDS and may not generalize to other projects. For example, in MFEM and deal.II, bugs are distributed more evenly across different modules, increasing the complexity of multiple-file bug localization.

In the MFEM and deal.II projects, Rogot2 performs well in the multiple-file bug scenario, identifying 8% and 14% more bugs in *Top-1* than in the single-file bug scenario, respectively. This advantage can be attributed to Rogot2's reliance on test coverage data to calculate suspiciousness scores, which benefits from the increased interactions between files in multiple-file bugs. The cooperative behavior of test cases in triggering multiple files provides more comprehensive coverage data for Rogot2 to analyze. However, in single-file bugs, where coverage data may be more localized, Rogot2's performance is less dominant compared to methods like DeepFL. Rogot2 demonstrates strong

performance in multiple-file scenarios while maintaining moderate execution times by leveraging simpler data collection processes, such as test case coverage. This approach is significantly less resource-intensive compared to mutation-based methods like Metallaxis or MUSE, making Rogot2 a practical choice for CAE projects with large-scale computational demands. However, its reliance on basic coverage data restricts its adaptability in single-file bug scenarios, where richer feature integration may be required to achieve optimal accuracy.

In comparison, Metallaxis, MUSE, Bugspots, SmartFL, and DeepFL perform better in the single-file bug scenario. For example, in the single-file bug scenario, DeepFL finds 43% and 50% of bugs in the top 20 most suspicious files. However, when we compare the *MFR* value of DeepFL in the single-file and multiple-file bug scenarios of the MFEM and deal.II projects (i.e., $MFEM_s$ vs. $MFEM_m$, and $deal.II_s$ vs. $deal.II_m$), we find that the *MFR* value increases from 53.00 and 69.52 to 57.41 and 73.06. This means that developers need to check more files on average before locating multiple-file bugs compared to that in the single-file bug scenario. In multiple-file bugs, interdependencies between modules introduce noise and ambiguity, with failures in one module potentially causing cascading effects and coverage overlaps. This complexity weakens DeepFL's ability to isolate buggy files. In contrast, single-file bugs are localized, allowing DeepFL to effectively leverage its features, leading to better performance.

Multiple-file bugs are common and difficult to localize in CAE projects. CAE projects often simulate complex tasks involving multiphysics phenomena. Such simulation requires close cooperation among various CAE modules, and also requires the collection and processing of large amounts of input data from different physical fields to achieve the overall simulation goal. Suppose our goal is to simulate the structural response of a bridge. The scenario involves structural mechanics, earthquake engineering, and wind action analysis. For example, in the MFEM project, this process requires collaborative work among the Finite Element Analysis module, the Material Properties module, and the Dynamic Loading module. At the same time, parameters such as elastic information, damping ratio, density, and wind speed of the material need to be dealt with. Such close cooperation between modules complicates multiple-file bug localization.

Multiple-file bugs are likely to be a common challenge in other CAE projects as well, especially those involving complex and interrelated simulation modules. The inherent nature of CAE systems, which often requires the integration of multiple physical domains and the management of large datasets, suggests that Rogot2's robust performance in multiple-file bug scenarios may extend to other similar CAE projects. However, the effectiveness of other methods, such as Metallaxis, MUSE, Bugspots, SmartFL, and DeepFL, may vary depending on the specific characteristics of the CAE system and the interactions among its modules.

Conclusion. Metallaxis, MUSE, Bugspots, SmartFL, and DeepFL perform well at targeting single-file bugs, while Rogot2 shows its strong potential in the multiple-file bug scenario. Effective methods for multiple-file FL in the CAE field still need to be developed.

5 DISCUSSION

This section summarizes the lessons learned and the potential research directions for future studies.

5.1 Lessons Learned

Test density is generally low. According to RQ1, the test density of the CAE projects used in this study is low. For example, in the MFEM project, there are only 3.4 test cases per 1000 lines of code, which means that many functions and code paths in the project are not fully tested (i.e., low test coverage). FL relies heavily on test coverage information, and when the coverage is too sparse, the algorithm struggles to accurately pinpoint the potential fault areas. This leads to:

- **Increased false positives:** FL may flag irrelevant code as potential fault locations.
- **Longer fault localization time:** Due to insufficient coverage, developers require more time to manually inspect and verify possible fault locations.

Insufficient differentiation of coverage information. When we compute the similarity between the code coverage of the failing test case and the passing ones, among the 100 most similar test cases for a certain bug in the FDS project, 33 test cases have the same similarity value. Highly similar test cases may lead to misleading coverage information, making it difficult for FL methods to accurately identify the true bug location. This issue is particularly critical in CAE projects, which often involve subtle differences between numerous parameter configurations in simulation scenarios. As a result:

- **Reduced fault localization accuracy:** When test case differences are subtle, the algorithm struggles to identify the specific code sections responsible for the fault.
- **Increased complexity in fault analysis:** Developers must perform additional analysis to filter out truly relevant test cases, adding complexity to the problem-solving process.

Difficulty in generating test cases. The inputs of CAE projects are usually complex. For example, the input of the FDS project has up to 600+ parameters to define the simulation scenarios and conditions. CAE projects often involve complex physical equations, mathematical models, and a large number of parameter configurations, making it challenging to automatically generate high-quality test cases. The lack of quantity and quality of test cases adversely affects the effectiveness of FL, leading to:

- **Insufficient test samples:** FL relies on ample test data to accurately infer the fault location. A lack of test samples causes the algorithm to fail to converge on the correct fault area.
- **Failure to cover edge cases:** When test cases do not cover enough edge cases or specific scenarios, certain complex or rare conditions may be missed, reducing the comprehensiveness of FL.

Long execution time. Testing CAE projects typically needs complex calculations and large-scale data processing, resulting in long execution time for individual test cases. In the MFEM project, the data collection time for one of the bugs is as long as 208.13 hours. This decreases the efficiency of FL methods; it also increases the probability that multiple-file bugs overlap each other during a long execution periods, resulting in:

- **Difficulty analyzing concurrency issues:** Long-running test cases may involve concurrent tasks, with failures occurring across multiple files, making it difficult for FL to determine which specific file or module is faulty.
- **Reduced real-time feedback:** Long test execution time hinders the ability of the development team to receive prompt feedback, affecting the real-time performance of software development and debugging, and ultimately impacting project timelines.

Insufficient utilization of domain-specific features. CAE projects are closely tied to specific engineering fields. Typical FL methods, even the learning-based ones, fail to fully exploit domain-specific knowledge, such as understanding the importance of certain parameters in simulations or how physical laws affect software behavior. This lack of feature utilization can lead to:

- **Lack of context awareness:** General FL methods may not recognize critical parameters or physical constraints in certain domains, making it difficult to associate specific errors with related code or model behavior.
- **Error accumulation:** In physical simulations, small errors can accumulate, leading to significant problems. FL methods that fail to capture these subtle differences may miss gradually amplifying errors.

5.2 Future Research Suggestions

Improving existing FL methods. Many existing FL methods are not suitable for CAE projects due to the scalability issues and the incompatibility of programming languages [58]. For example, the mutation tool MuCPP is only applicable to C++, while FDS is developed by Fortran. It is urgent to adapt existing methods and projects to the computational environment and programs of CAE projects.

Possible solutions: (1) *Cross-language method development*: Develop a cross-language fault localization methods for large and complex CAE projects based on existing multi-language parsing tools such as LLVM, GCC, ANTLR, and Roslyn. These tools have demonstrated scalability and effectiveness in engineering practice, allowing us to leverage their mature ecosystems and technical foundations, thereby avoiding the need to build parsers from scratch. Additionally, adaptive technologies will be integrated to ensure the performance and compatibility of the method in a multi-language environment, thereby meeting the demands of CAE projects. (2) *Mining domain-specific FL features*: It is important to research and develop FL methods for specific engineering domains that understand and leverage CAE-specific knowledge, such as specific design parameters and engineering constraints. We can integrate such knowledge as features into FL methods to improve the FL accuracy.

Development of CAE domain benchmark datasets. The CAE domain lacks publicly available datasets. However, the complexity of the engineering problems involved in CAE makes it difficult to obtain the datasets.

Possible solutions: (1) *Industry collaboration*: Industry and academia can collaborate to create and maintain a public dataset of CAE projects. This includes the activities of standardizing data formats, defining common engineering problems, and providing thorough documentation and usage guidance. (2) *Fault injection*: An alternative solution is to artificially introduce some common faults into CAE projects, such as algorithm errors, parameter configuration errors, and boundary condition setting errors, which helps the creation of a CAE domain benchmark dataset with artificial faults.

Development of high-quality test cases. The test density in CAE projects is generally low, and the differentiation of coverage information is insufficient. The complex physical models and algorithms of CAE projects, the large number of input parameters and configuration options, and parallel execution characteristics increase the complexity of test case design.

Possible solutions: (1) *Automated test case generation*: We can utilize a model-driven approach to automatically generate test cases to cover complex physical models and algorithms. The test case generator can also automatically adjust input parameters to explore the boundary conditions of the program. (2) *Introducing more coverage dimensions*: By adding additional coverage dimensions (such as branch coverage, path coverage, and data-flow coverage), the similarity between test cases is no longer just based on code execution but also includes richer context information (like branches, paths, and data flows). This helps to reduce the issue of having too many similar test cases and improves the accuracy of fault localization. (3) *Weighted similarity calculation*: By giving different types of coverage information varying importance, the key coverage data will have a bigger impact in the similarity calculation. This allows fault localization methods to focus more accurately on the code blocks that are related to the fault.

Dealing with multiple-file bugs in CAE projects. The output of CAE projects is affected by various factors, including the physical and mathematical models, the discretization of mesh division, and the numerical calculations. When a CAE project has bugs, it is particularly crucial to identify and locate which part causes the bugs, since the bugs are usually caused by multiple files. However,

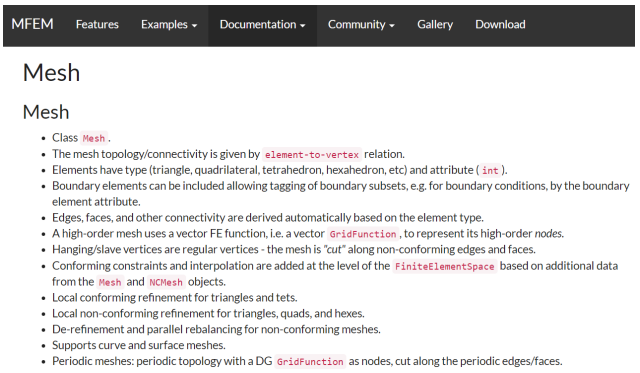


Fig. 5. Documentation of the Mesh Module in MFEM Main Modules.

how to effectively analyze and address bugs across multiple modules and files in complex CAE projects remains further exploration.

Possible solutions: (1) *Inter-module bug analysis*: The future study can develop methods that analyze and locate bugs across multiple modules and files. This may include methods such as improved fault propagation analysis and enhanced dependency analysis. (2) *Comprehensive bug effect analysis*: It is important to conduct a comprehensive analysis of approximation errors in physical and mathematical models, discretization errors in meshing, and floating point errors in numerical calculations to determine their impact on output. Such analysis can guide FL accordingly.

Scalability of CAE domain-specific FL features to other industrial software domains. While this study focuses on enhancing FL in CAE projects, there is considerable potential to extend these techniques to other industrial software domains. The features developed for CAE-specific FL can serve as a foundation for addressing the unique needs of these other domains.

Possible solutions: (1) *Generalizing domain-specific knowledge*: Many industrial software projects share commonalities with CAE projects, such as the use of simulation models, large-scale parallel computing, and complex input configurations. By generalizing the CAE-specific FL features to encompass other industrial domains, we can develop methods that are more broadly applicable. (2) *Cross-domain collaborations*: Collaborating with industries that face similar computational challenges can help develop benchmark datasets and FL methods that are applicable across domains. Cross-domain research can facilitate the sharing of best practices and the adaptation of successful FL techniques from one domain to another.

6 IMPROVEMENT OF FAULT LOCALIZATION IN CAE PROJECTS

The results in RQ1 reveal the unique advantages and disadvantages of the current FL methods on CAE projects. However, due to the simulation characteristics of CAE projects (e.g., iterative execution, inaccurate results, and complex computation), the precision in pinpointing bugs within CAE projects remains unsatisfactory. In this section, we investigate how to improve the effectiveness of FL methods based on the above lessons.

6.1 Fault Localization by Main Module Similarity

The main modules of CAE projects are designed for simulation (e.g., mesh generation, numerical methods implementation) as discussed in Section 2. Since developers frequently work on these main modules during the CAE project development, an interesting observation is to investigate

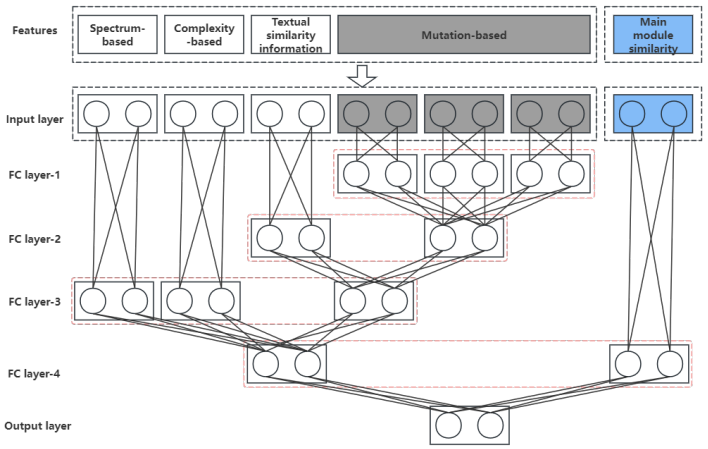


Fig. 6. The Framework of DeepFL$_{main}$.

the potential correlation between the relevance of a file to CAE simulation characteristic and its propensity for harboring bugs.

Figure 5 shows part of the documents of the "Mesh" module in the MFEM project. The document introduces mesh topology and connectivity, element types and properties, higher-order meshes, mesh refinement and derefinement, etc. The document clearly explains the diverse applications and technical details of grid processing in CAE projects, emphasizing the importance of grids as the core component of numerical simulation and engineering analysis.

Our hypothesis is that if the text content of a file is highly similar to the description of main modules in CAE documentation, the file plays a more important role in the entire CAE projects and is more likely to become a hotspot for bugs. Therefore, we propose a new similarity-based feature to improve the accuracy of FL. We study the similarity between each source code file in the CAE project and the main module documentation, which comes from the official introduction (e.g., the official introduction of the MFEM project[11]). The main module documentation details key components and functions within the different modules, including the various mesh types, mathematical components, and computational tools. Additionally, the documentation covers the different types of solvers and their classes, domains, and dependencies. When computing the similarity, we treat each source code file as a query, and the description in the official documentation as a document. We represent them using the TF-IDF vectorization, and compute the cosine similarity between a source code file (i.e., query) and the official documentation. Then, we treat this similarity score as a new feature group.

Since the experimental results in Section 4 show that compared with other FL methods, the learning-based method DeepFL exhibits higher accuracy, we integrate the new feature group into DeepFL for experiments.

Figure 6 shows the framework of the new methods (denoted as DeepFL$_{main}$). The lower part of the figure is the DeepFL model. The blue neurons are the new features we proposed. The upper part of the figure is the input features, which correspond to the neurons in the input layer. Each feature group is first connected with a fully-connected (FC) layer to extract the useful debugging information within this specific feature group. Then, the extracted debugging information of each

[11]https://mfem.org/

Algorithm 1 The pseudocode of DeepFL$_{main}$

Input: Program P with code elements to be analyzed, a set of test cases T providing test outcomes.
Output: Ranked list of code elements based on fault likelihood.

1: Initialize feature groups: SBFL, MBFL, Complexity Metrics (CM), Textual Similarity (TS), Main Module Similarity (MMS)
2: Initialize weights (W) and biases (b) for all layers.
3: **for** each feature group g in SBFL, MBFL, CM, TS, MMS **do**
4: Process features with a dedicated fully connected layer: $h_g = \text{Activation}(W_g \cdot x_g + b_g)$ {h_g: Hidden layer output for group g, x_g: Input features for group g}
5: **end for**
6: Concatenate all feature group outputs to form a complete layer: $h = [h_{\text{SBFL}}, h_{\text{MBFL}}, h_{\text{CM}}, h_{\text{TS}}, h_{\text{MMS}}]$ {h: Combined feature layer}
7: Apply deep layers to integrate features from the complete layer: $z = \text{Activation}(W_{\text{deep}} \cdot h + b_{\text{deep}})$ {z: Integrated deep layer output}
8: Predict fault likelihood using softmax output layer: $y = \text{softmax}(W_{\text{output}} \cdot z + b_{\text{output}})$ {y: Probability distribution of fault likelihood}
9: Rank code elements based on predicted likelihood y
10: Return the top-ranked code elements as potential fault locations.

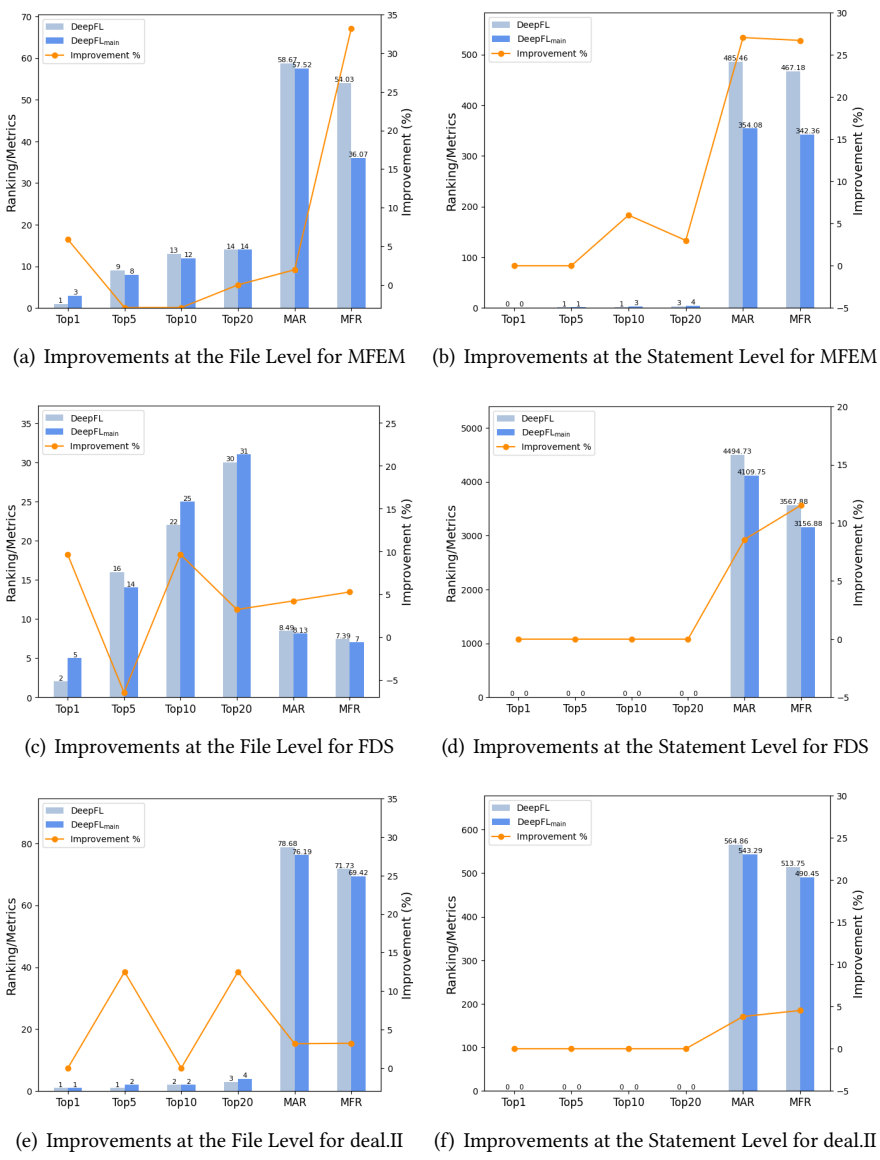
Table 7. Comparison of DeepFL and DeepFL$_{\text{main}}$ on Single-file Bugs (_s) and Multiple-file Bugs (_m) on MFEM, FDS, and deal.II.

	DeepFL						DeepFL$_{\text{main}}$					
Projects	*Top-1*	*Top-5*	*Top-10*	*Top-20*	*MAR*	*MFR*	*Top-1*	*Top-5*	*Top-10*	*Top-20*	*MAR*	*MFR*
$MFEM_s$	0	5	9	9	53.00	53.00	1	5	9	9	35.94	35.94
$MFEM_m$	1	4	4	5	60.89	57.41	2	3	3	5	58.72	37.20
$deal.II_s$	1	1	1	2	69.52	69.52	1	1	1	3	67.02	67.02
$deal.II_m$	0	0	1	1	81.04	73.06	0	1	1	1	79.74	72.20
FDS_s	1	10	14	22	7.97	7.97	3	9	17	23	7.47	7.47
FDS_m	1	6	8	8	8.57	4.92	2	5	8	8	8.52	4.38

feature group is concatenated together through several FC layers. Finally, DeepFL$_{main}$ connects the fully-connected layer to the output layer to perform prediction. Algorithm 1 shows the pseudocode of DeepFL$_{main}$.

6.2 Experiment Setup

Since in the experiments of Section 4, only MFEM and deal.II projects applies the DeepFL method, we integrate new features into the original four feature groups in DeepFL (i.e., the spectrum-based, mutation-based, complexity-based, and text similarity-based features), to form a new feature vector. By comparing the changes regarding various evaluation metrics before and after adding new features to DeepFL, we can assess whether the new features can improve the FL accuracy of CAE projects. For the FDS project, it does not apply the DeepFL method, because of the missing of mutation-based features (see Section 3.4 for details). Therefore, we only use three features based on spectrum, complexity, and text similarity information to compare the accuracy before and after adding new features.



(a) Improvements at the File Level for MFEM (b) Improvements at the Statement Level for MFEM

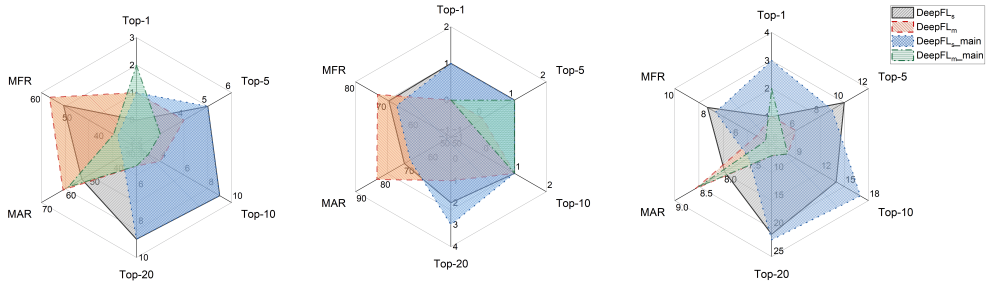
(c) Improvements at the File Level for FDS (d) Improvements at the Statement Level for FDS

(e) Improvements at the File Level for deal.II (f) Improvements at the Statement Level for deal.II

Fig. 7. Comparison of DeepFL and DeepFL$_{main}$ on MFEM, FDS, and deal.II.

Further, in order to verify whether the new features are applicable to other CAE projects, we select Kratos[12] from the remaining projects in Section 3.1. Kratos is a framework for building parallel, multi-disciplinary simulation project; it offers solutions for particle simulation and fluid-structure interaction problems. A difference of Kratos from other projects is that it uses a combination of Python and C++ to develop. We reproduce 9 bugs according to the selection principle in Section 3.1. However, there are obvious differences in the syntactic and semantic of the two programming

[12]https://github.com/KratosMultiphysics/Kratos



(a) Improvements on MFEM (b) Improvements on deal.II (c) Improvements on FDS

Fig. 8. Comparison of Single-file Bugs and Multiple-file Bugs Between DeepFL and DeepFL$_{main}$ on MFEM and FDS with Radar Chart.

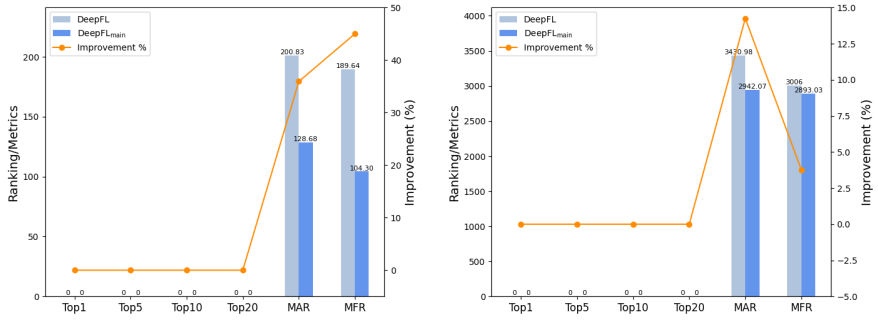
languages (i.e., Python and C++). It is certainly challenging to ensure that mutation operators can obey the grammatical rules of each language while taking into account the interaction between the two languages in a mixed-language environment. Therefore, as with the FDS setting, we abandon mutation-based features to verify whether the new features can improve the FL accuracy of new CAE projects.

Following the DeepFL setup, we perform leave-one-out cross-validation on bugs for each project.

6.3 Experiment Results

Overall Effectiveness of DeepFL$_{main}$. As shown in Figure 7, the addition of the new features into DeepFL$_{main}$ has enhanced FL in the MFEM project, with an improvement of *MAR* and *MFR* by 1.96% and 33.24% on the file level, respectively. Notably, DeepFL$_{main}$ has led to the identification of an additional 5.88% of bugs at the *Top-1* position. On the statement-level, the *MAR* and *MFR* of DeepFL$_{main}$ are improved by 27.06% and 26.72%, respectively. In the FDS project, even though the *Top-5* metric has decreased, the important *Top-1* metric has increased by 9.68%; *MAR* and *MFR* have also increased by 8.57% and 11.52% on the statement-level results. In the deal.II project, although the *Top-N* metrics at the statement level did not improve, the *MAR* and *MFR* metrics increased by 3.82% and 4.54%. This indicates that domain knowledge can generally help narrow down the potential fault locations, thus improving *MAR* and *MFR*. However, as the system scales increases (deal.II is 12 times the size of MFEM), there remains a significant amount of code that still needs to be traversed and analyzed, even with domain knowledge. The large number of code paths may reduce the effectiveness of domain knowledge, resulting in no significant increase in accuracy (*Top-N*). Therefore, efficient algorithms and stronger computational capabilities are required to address the complexity of large-scale codebases effectively.

Effectiveness on single-file and multiple-file bugs. We present the effectiveness of DeepFL$_{main}$ at the file level against single-file bugs and multiple-file bugs in Table 7. The results show that DeepFL$_{main}$ can improve the FL accuracy of both types of bugs. For single-file bugs, the *MFR* value of MFEM, FDS, and deal.II increases by 32.19%, 6.27%, and 3.60%, respectively. In addition, the improvement on the multiple-file bugs is larger. The *MFR* value of MFEM, FDS, and deal.II increases by 35.20%, 10.98%, and 1.18%, respectively. Meanwhile, DeepFL$_{main}$ also identifies more bugs at *Top-1* compared against DeepFL. This improvement is attributed to the ability to more accurately identify files that play a key role in the project, as well as their functional connections or dependencies with multiple-file bugs through main module similarity analysis. Figure 8 shows the



(a) Improvements at the File Level for Kratos (b) Improvements at the Statement Level for Kratos

Fig. 9. Comparison of DeepFL and DeepFL$_{main}$ on Kratos.

Table 8. Comparison of DeepFL and DeepFL$_{\text{main}}$ on Single-file Bugs (_s) and Multiple-file Bugs (_m) on Kratos

	DeepFL						DeepFL$_{\text{main}}$					
Projects	*Top-1*	*Top-5*	*Top-10*	*Top-20*	*MAR*	*MFR*	*Top-1*	*Top-5*	*Top-10*	*Top-20*	*MAR*	*MFR*
$Kratos_s$	0	0	0	0	246.50	246.50	0	0	0	0	132.00	132.00
$Kratos_m$	0	0	0	0	156.50	119.45	0	0	0	0	118.25	56.50

comparison of DeepFL and DeepFL$_{main}$ on single-file and multiple-file FL in three projects through a radar chart. The radar chart not only highlights the advantages of each FL method in locating different types of bugs, but also measures the overall performance of each method by the area of the enclosed area. Among these metrics, higher *Top-N* values are better, while lower *MAR* and *MFR* values are better. It is clearly visible in the figure that DeepFL is mainly displayed on the left (in gray and orange), while DeepFL$_{main}$ occupies a large area on the right (in blue and green); this indicates the new features improve the performance of DeepFL$_{main}$.

Generality of DeepFL$_{main}$. As shown in Figure 9, the evaluation on the Kratos project echoes the above findings. With the addition of new features, *MAR* and *MFR* metrics jump by 35.93% and 45.00%, respectively. These enhancements highlight the potential for notable advancements of the new features. Kratos requires the examination of an additional 2551 statements compared to the MFEM project, which highlights the ongoing challenges in multi-language bug localization [1]. This complexity aligns with the insights from the previous research [20]. For Kratos, we also present the effectiveness of DeepFL$_{main}$ against DeepFL in terms of single-file bugs and multiple-file bugs in Table 8. The *MFR* values of single-file bugs and multiple-file bugs increase by 46.45% and 52.70%, respectively. This shows that our proposed domain-based features can be well extended to other CAE projects. Figure 10 visualizes the comparison of DeepFL and DeepFL$_{main}$ on single-file and multiple-file FL in Kratos through a radar chart, which shows the same conclusion as Table 8.

Conclusion. The use of main module similarity-based features improves the accuracy of FL, with the improvement of *MAR* and *MFR* metrics by up to 35.93% and 45.00%, respectively.

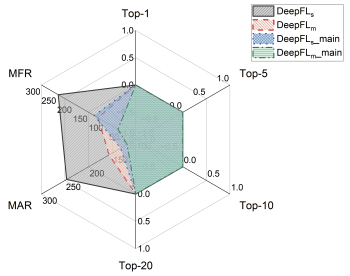


Fig. 10. Comparison of Single-file Bugs and Multiple-file Bugs Between DeepFL and DeepFL$_{main}$ on Kratos with Radar Chart.

7 THREATS TO VALIDITY

Threats to internal validity. Internal threats arise primarily from the way we implement our FL methods. To reduce this threat, we select FL methods that can reproduce, and conduct experiments strictly according to the default settings in the original paper.

Threats to external validity. External threats mainly stem from the commonality or generality of different CAE projects. To reduce this threat, we select three projects from different CAE fields, and ensure the diversity in programming languages and sizes of the projects. In addition, to further verify the generality of the new features, we introduce the Kratos project, which has not been used in the previous experiments, as a new test dataset. As part of future work, we plan to expand to a wider range of benchmarks.

8 RELATED WORK

8.1 Research on CAE projects

This study is related to the field of CAE. Lou et al. [36] propose a mechanical fault diagnosis method that integrates numerical simulation with transfer learning. They generate simulation samples using the finite element method and enhance them using generative adversarial networks, ultimately training a transfer learning network to extract bug features. Di Franco et al. [15] conduct a comprehensive study of digital bug characteristics in the real world. They analyze bugs in numerical software libraries and discuss their classification, frequency, and methods of correction. Cuesta et al. [9] introduce a CAE model simulation state estimation method. This method is used to simulate actual system states that cannot be directly measured.

Unlike these studies on theoretical models and technological development, our work focuses on locating faults for CAE projects in practical engineering applications, which are key infrastructures in modern scientific and engineering research.

8.2 Empirical Study on Fault Localization

This study relates to empirical research on FL. Wong et al. [56] classify and comprehensively summarize software FL methods. They highlight the potential shortcomings and limitations of existing methods, as well as a discussion of key issues and concerns in software FL. Pearson et al. evaluate seven spectrum-based and mutation-based FL methods through 2995 artificial faults and 310 real faults. They find that these methods are limited in locating real faults [42]. The study also identifies the key issues that affect FL methods in practical applications. Jiang et al [24] conduct a systematic empirical study on the combination of spectrum-based FL with statistical debugging, and establish a unified model for both. Zou et al. [67] examine the effectiveness of 11 FL methods

Table 9. Comparison of FL Research Between Embedded Control and Simulation Software and CAE Projects

	Embedded Control and Simulation Software	CAE Projects
Target and Application Areas	Primarily used for embedded system development, control system design, or simulations in specific industries such as manufacturing, automotive, and consumer electronics.	Focused on physical simulation and engineering analysis, with applications in fields like aerospace, mechanical engineering, and civil engineering.
Development and Simulation Methods	Typically developed using graphical models. Embedded system development often involves real-time simulation or hardware-in-the-loop (HIL) simulation, emphasizing real-time performance and control accuracy.	Primarily focused on numerical simulations and solver computations, relying on complex physical and mathematical models to solve engineering problems, often necessitating high-performance computing for large-scale numerical challenges.
Complexity and Scale	Complexity often arises from hardware integration; for instance, NXP TV520 platform control software must manage multiple CPUs and signal processing modules to handle tasks like audio/video processing and user interface interactions.	Complexity stems from the coupling of multiple physical fields and large-scale numerical computations.
Input and Testing Difficulty	Inputs are usually structured and standardized. For instance, Simulink models handle limited signal inputs, and tests in HIL simulations are typically controllable. While NXP TV520 control software processes complex inputs like remote signals and audio/video streams, input types remain limited.	Inputs are more diverse and complex, often involving hundreds or thousands of physical parameters. Testing these systems is challenging due to the difficulty in automatically generating inputs, with large-scale test cases and high time costs for generating and executing tests.
Execution Time and Resource Requirements	Due to real-time and resource constraints, industrial software typically demands faster execution times. Control software in embedded systems must operate in real-time environments, with rapid responses to user interactions or hardware events.	CAE projects often require significant computational resources for large-scale numerical simulations, with execution times ranging from hours to days.
Challenge	In industrial software, the research team faced significant technical challenges in code instrumentation and data collection due to specific language features, architectural choices, and the closed nature of proprietary projects. Furthermore, constraints like memory, processing power, and real-time requirements complicate the practical application of these tools. Many companies also lack systematic processes for generating test cases, hindering the effectiveness of automated debugging tools.	

from 7 different families on 357 real-world faults. They apply a learning-to-rank model to combine these FL methods. Pearson et al. [42] compare 7 FL methods. They study whether artificial faults can be a good substitute for real faults. They also analyze which characteristics make FL methods perform well on real faults.

However, considering the nature of CAE projects, we conduct the first empirical study to analyze the feasibility of applying FL methods on real-world CAE projects.

8.3 Differences Between Fault Localization in CAE and Other Software

This study relates to the differences between fault localization in CAE and existing industrial software. Abreu et al. [3] investigate the application of spectrum-based fault localization in embedded software, specifically focusing on the LCD TV control software for the NXP TV520 platform. Their findings demonstrate that SBFL exhibits strong scalability and effectiveness when applied to large-scale embedded code. Additionally, they identify the challenges of adopting automated debugging technologies in the industry [2]. To address these challenges, they propose recommendations for the

software engineering community, suggesting open collaboration and the integration of automated tools into commonly used IDEs. Liu et al. propose a Simulink model fault localization method that combines statistical debugging with dynamic model slicing and further enhance localization accuracy using the iterative algorithm iSimFL [35]. Their application in automotive embedded systems demonstrates that this approach achieves strong accuracy and effectiveness. Table 9 compares fault localization challenges between embedded control software and CAE projects. Embedded software focuses on control system design with real-time simulation, hardware integration, and low execution demands. CAE projects, however, center on physical simulation and engineering analysis, requiring high-performance computing, complex inputs, and multi-physics coupling. Both face challenges in code instrumentation, data collection, real-time constraints, and a lack of automated test generation, limiting fault localization effectiveness.

Our research also reveals significant differences in the performance of existing FL methods when applied to CAE projects compared to general software projects. In RQ1, we found that when using the same FL methods on the Defects4J dataset, previous studies indicate that developers only need to examine an average of 20.32 statements (i.e., MFR=20.32) in the ranking list before locating the bug. However, in CAE projects, the number of statements that need to be checked is 23 times higher. In RQ2, we observed a substantial difference from previous research on Defects4J, Siemens, and SIR datasets regarding the execution time of MBFL and SBFL methods on CAE projects. Even though we employed more powerful computational resources in this study, the execution time for the same MBFL and SBFL methods on CAE projects was still 10 to 71 times longer than the execution times reported in previous studies on other datasets. In RQ3, we found that in Java projects, single-file bugs and multiple-file bugs account for 82.5% and 17.5%, respectively. In contrast, the proportion of multi-file bugs in CAE projects is significantly higher (36.8%)[13], far exceeding the percentage found in Java projects. Therefore, the overall fault location effect was worse than that of general software projects.

9 CONCLUSION

This paper investigates the effectiveness and feasibility of various FL methods (including 6 methods from 5 categories) in the field of CAE. Through a systematic analysis of 76 real-world bugs from three projects: FDS, deal.II, and MFEM, we demonstrate that even state-of-the-art FL methods have limitations, requiring long time to determine bug locations. Based on the findings of this paper, we propose a set of features based on the CAE main module to improve FL for CAE projects. The addition of new features improves the best performing FL method in this study by 35.93% and 45.00% in terms of *MAR* and *MFR*, respectively. Although CAE projects play an irreplaceable role in modern engineering design and analysis, FL methods still require continuous innovation and cooperation, including improving existing FL methods to adapt to the multi-language environment of CAE, creating benchmark datasets, and developing high-quality test cases. These directions can not only improve the accuracy and practicality of FL methods, but also help software developers locate bugs in complex CAE projects more effectively.

10 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments, which help us a lot to improve our paper. This work is supported in part by the National Natural Science Foundation of China under Grant Nos. 62202079, 62132020, 62032004, and 62472062; the Fundamental Research Funds for the Central Universities, Grant No. DUT24LAB126; and the National Key Research and Development Program of China, Grant No. 2018YF-B1003900.

[13]The 36.8% figure is calculated based on a subset of sampled projects, including FDS, deal.II, and MFEM.

REFERENCES

[1] Mouna Abidi, Md Saidur Rahman, Moses Openja, and Foutse Khomh. 2021. Are Multi-Language Design Smells Fault-Prone? An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 29 (feb 2021), 56 pages. https://doi.org/10.1145/3432690

[2] Rui Abreu. 2022. The Bumpy Road of Taking Automated Debugging to Industry. arXiv:2212.01237 [cs.SE] https://arxiv.org/abs/2212.01237

[3] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035 SI: TAIC PART 2007 and MUTATION 2007.

[4] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 88–99. https://doi.org/10.1109/ASE.2009.25

[5] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. 2018. We don't need another hero? the impact of "heroes" on software development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 245–253. https://doi.org/10.1145/3183519.3183549

[6] H. Agrawal, J.R. Horgan, S. London, and W.E. Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95.* 143–151. https://doi.org/10.1109/ISSRE.1995.497652

[7] Mustapha Ait Hssain, Rachid Mir, and Youness El Hammami. 2020. Numerical Simulation of the Cooling of Heated Electronic Blocks in Horizontal Channel by Mixed Convection of Nanofluids. *Journal of Nanomaterials* 2020 (02 2020). https://doi.org/10.1155/2020/4187074

[8] Yihua Cao, Guozhi Li, and Dan Song. 2021. Numerical simulation of melting of ice accreted on an airfoil. *Aerospace Science and Technology* 119 (2021), 107223. https://doi.org/10.1016/j.ast.2021.107223

[9] Pablo Luque Carlos Cuesta and Daniel A. Mantaras. 2022. A methodology to improve simulation of multibody systems using estimation techniques. *Automatika* 63, 1 (2022), 16–25. https://doi.org/10.1080/00051144.2021.1999129 arXiv:https://doi.org/10.1080/00051144.2021.1999129

[10] An Ran Chen, Md Nakhla Rafi, Tse-Hsun Chen, and Shaohua Wang. 2023. Back to the Future! Studying Data Cleanness in Defects4J and its Impact on Fault Localization. arXiv:2310.19139 [cs.SE]

[11] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 223–234. https://doi.org/10.1145/3338906.3338957

[12] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2021. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 78–89. https://doi.org/10.1145/3324884.3416570

[13] Zheng-Wei Chen, Tang-Hong Liu, Chun-Guang Yan, Miao Yu, Zi-Jian Guo, and Tian-Tian Wang. 2019. Numerical simulation and comparison of the slipstreams of trains with different nose lengths under crosswind. *Journal of Wind Engineering and Industrial Aerodynamics* 190 (2019), 256–272. https://doi.org/10.1016/j.jweia.2019.05.005

[14] Tung Dao, Na Meng, and ThanhVu Nguyen. 2023. Triggering Modes in Spectrum-Based Multi-location Fault Localization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1774–1785. https://doi.org/10.1145/3611643.3613884

[15] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 509–519.

[16] Ali Ghanbari and Lingming Zhang. 2019. PraPR: Practical Program Repair via Bytecode Mutation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1118–1121. https://doi.org/10.1109/ASE.2019.00116

[17] Foyzul Hassan, Na Meng, and Xiaoyin Wang. 2023. UniLoc: Unified Fault Localization of Continuous Integration Failures. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 136 (sep 2023), 31 pages. https://doi.org/10.1145/3593799

[18] Xiao He, Xingwei Wang, Jia Shi, and Yi Liu. 2020. Testing high performance numerical simulation programs: experience, lessons learned, and open issues. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 502–515. https://doi.org/10.1145/3395363.3397382

[19] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, André van Hoorn, Antonio Filieri, and David Lo. 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience* 49 (2019), 1197 – 1224. https://api.semanticscholar.org/CorpusID:182489114

[20] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-Based Fault Localization for Real-World Multilingual Programs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 464–475. https://doi.org/10.1109/ASE.2015.14

[21] Simo Hostikka, Kevin McGrattan, and Anthony Hamins. 2003. Numerical Modeling of Pool Fires Using LES and Finite Volume Method for Radiation. Fire Safety Science. Proceedings. Seventh (7th) International Symposium. International Association for Fire Safety Science (IAFSS). June 16-21, 2003 , Worcester, MA, US. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=909855

[22] Tom Janssen, Rui Abreu, and Arjan J.C. van Gemund. 2009. Zoltar: A Toolset for Automatic Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 662–664. https://doi.org/10.1109/ASE.2009.27

[23] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[24] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 502–514. https://doi.org/10.1109/ASE.2019.00054

[25] Jiajun Jiang, Yumeng Wang, Junjie Chen, Delin Lv, and Mengjiao Liu. 2023. Variable-based Fault Localization via Enhanced Decision Tree. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 41 (dec 2023), 32 pages. https://doi.org/10.1145/3624741

[26] 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. https://doi.org/10.1145/581339.581397

[27] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 92–101. https://doi.org/10.1145/2597073.2597074

[28] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A Critical Evaluation of Spectrum-Based Fault Localization Techniques on a Large-Scale Software System. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 114–125. https://doi.org/10.1109/QRS.2017.22

[29] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *29th International Conference on Software Engineering (ICSE'07)*. 489–498. https://doi.org/10.1109/ICSE.2007.66

[30] Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. 2019. Precise Learn-to-Rank Fault Localization Using Dynamic and Static Features of Target Programs. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 23 (oct 2019), 34 pages. https://doi.org/10.1145/3345628

[31] 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. https://doi.org/10.1145/3293882.3330574

[32] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 92 (oct 2017), 30 pages. https://doi.org/10.1145/3133916

[33] 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. https://doi.org/10.1109/ICSE43902.2021.00067

[34] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 162 (oct 2019), 30 pages. https://doi.org/10.1145/3360588

[35] Bing Liu, Shiva Nejati, Lionel Briand, and Thomas Bruckmann. 2016. Simulink fault localization: an iterative statistical debugging approach: SIMULINK FAULT LOCALIZATION. *Software Testing, Verification and Reliability* 26 (09 2016). https://doi.org/10.1002/stvr.1605

[36] YunXia Lou, Anil Kumar, and Jiawei Xiang. 2023. Machinery fault diagnostic method based on numerical simulation driving partial transfer learning. *Science China Technological Sciences* 66 (11 2023). https://doi.org/10.1007/s11431-023-2496-6

[37] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 664–676. https://doi.org/10.1145/3468264.3468580

[38] Nora McDonald and Sean Goggins. 2013. Performance and participation in open source software on GitHub. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems* (Paris, France) *(CHI EA '13)*. Association for Computing Machinery, New York, NY, USA, 139–144. https://doi.org/10.1145/2468356.2468382

[39] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1169–1180. https://doi.org/10.1145/3510003.3510147

[40] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society, USA, 153–162. https://doi.org/10.1109/ICST.2014.28

[41] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test. Verif. Reliab.* 25, 5–7 (aug 2015), 605–628. https://doi.org/10.1002/stvr.1509

[42] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 609–620. https://doi.org/10.1109/ICSE.2017.62

[43] A Pescaru, E Oanta, T Axinte, and AD Dascalescu. 2015. Extended precision data types for the development of the original computer aided engineering applications. In *IOP Conference Series: Materials Science and Engineering*, Vol. 95. IOP Publishing, 012125.

[44] Jie Qian, Xiaolin Ju, and Xiang Chen. 2022. GNet4FL: Effective Fault Localization via Graph Convolutional Neural Network. https://doi.org/10.21203/rs.3.rs-2000722/v1

[45] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. 2011. BugCache for inspections: hit or miss?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 322–331. https://doi.org/10.1145/2025113.2025157

[46] Benny Raphael and Ian FC Smith. 2003. *Fundamentals of computer-aided engineering*. John wiley & sons.

[47] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

[48] M. Renieres and S.P. Reiss. 2003. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* 30–39. https://doi.org/10.1109/ASE.2003.1240292

[49] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 273–283. https://doi.org/10.1145/3092703.3092717

[50] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2023. Learning to Construct Better Mutation Faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 64, 13 pages. https://doi.org/10.1145/3551349.3556949

[51] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning How to Mutate Source Code from Bug-Fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 301–312. https://doi.org/10.1109/ICSME.2019.00046

[52] Donghui Wang, Fan Hu, Zhenyu Ma, Zeping Wu, and Weihua Zhang. 2014. A CAD/CAE integrated framework for structural design optimization using sequential approximation optimization. *Advances in Engineering Software* 76 (2014), 56–68.

[53] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2348–2368. https://doi.org/10.1109/TSE.2019.2948158

[54] Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus Agus Haryono, Shaowei Wang, and David Lo. 2022. Real world projects, real faults: evaluating spectrum based fault localization techniques on Python projects. *Empirical Software Engineering* 27, 6 (2022), 147.

[55] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 181–190. https://doi.org/10.1109/ICSME.2014.40

[56] 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. https://doi.org/10.1109/TSE.2016.2521368

[57] 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* (San Jose,

CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 204–214. https://doi.org/10.1145/2610384.2610386

[58] Yonghao Wu, Zheng Li, Jie Zhang, Mike Papadakis, Mark Harman, and Yong Liu. 2023. Large Language Models in Fault Localisation. https://doi.org/10.48550/arXiv.2308.15276

[59] Yue Yan, Shujuan Jiang, Yanmei Zhang, and Cheng Zhang. 2023. An effective fault localization approach based on PageRank and mutation analysis. *Journal of Systems and Software* 204 (2023), 111799. https://doi.org/10.1016/j.jss.2023.111799

[60] Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2022. Fault Localization via Efficient Probabilistic Modeling of Program Semantics. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 958–969. https://doi.org/10.1145/3510003.3510073

[61] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/3092703.3092731

[62] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. 2021. An Empirical Study of Boosting Spectrum-Based Fault Localization via PageRank. *IEEE Transactions on Software Engineering* 47, 6 (2021), 1089–1113. https://doi.org/10.1109/TSE.2019.2911283

[63] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 272–281. https://doi.org/10.1145/1134285.1134324

[64] Zhuo Zhang, Ya Li, Jianxin Xue, and Xiaoguang Mao. 2024. Improving fault localization with pre-training. *Frontiers of Computer Science* 18, 1, Article 181205 (2024), 0 pages. https://doi.org/10.1007/s11704-023-2597-8

[65] Zhuo Zhang, Jianxin Xue, Deheng Yang, and Xiaoguang Mao. 2024. ContextAug: model-domain failing test augmentation with contextual information. *Frontiers of Computer Science* 18, 2 (2024), 182202.

[66] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*. 14–24. https://doi.org/10.1109/ICSE.2012.6227210

[67] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* 47, 2 (2021), 332–347. https://doi.org/10.1109/TSE.2019.2892102

Received XX XX XXX; revised XX XX XXX; accepted XX XX XXX