An Extensive Replication Study of the ABLoTS Approach for Bug Localization

Feifei Niu[1*], Enshuo Zhang[1], Christoph Mayr-Dorn[2], Wesley K. G. Assunção[3], Liguo Huang[4], Jidong Ge[1], Bin Luo[1], Alexander Egyed[2]

[1*]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China.
[2]Institute for Software Systems Engineering, Johannes Kepler University, Linz, Austria.
[3]Department of Computer Science, North Carolina State University, North Carolina, USA.
[4]Department of Computer Science, Southern Methodist University, Dallas, Texas, USA.

*Corresponding author(s). E-mail(s): niufeifei@smail.nju.edu.cn;
Contributing authors: 2575357413@qq.com;
christoph.mayr-dorn@jku.at; wguezas@ncsu.edu; lghuang@smu.edu;
gjd@nju.edu.cn; luobin@nju.edu.cn; alexander.egyed@jku.at;

Abstract

Bug localization is the task of recommending source code locations (typically files) that contain the cause of a bug and hence need to be changed to fix the bug. Along these lines, information retrieval-based bug localization (IRBL) approaches have been adopted, which identify the most bug-prone files from the source code space. In current practice, a series of state-of-the-art IRBL techniques leverage the combination of different components (e.g., similar reports, version history, and code structure) to achieve better performance. ABLoTS is a recently proposed approach with the core component, TraceScore, that utilizes requirements and traceability information between different issue reports (i.e., feature requests and bug reports) to identify buggy source code snippets with promising results. To evaluate the accuracy of these results and obtain additional insights into the practical applicability of ABLoTS, we conducted a replication study of this approach with the original dataset and also on two extended datasets (i.e., additional Java dataset and Python dataset). The original dataset consists of 11 open source Java

projects with 8,494 bug reports. The extended Java dataset includes 16 more projects comprising 25,893 bug reports and corresponding source code commits. The extended Python dataset consists of 12 projects with 1,289 bug reports. While we find that the TraceScore component, which is the core of ABLoTS, produces comparable or even better results with the extended datasets, we also find that we cannot reproduce the ABLoTS results, as reported in its original paper, due to an overlooked side effect of incorrectly choosing a cut-off date that led to test data leaking into training data with significant effects on performance. Additionally, we conduct experiments to assess the performance of various composers that aggregate scores from different components, revealing that Logistic Regression, fixed weight, and CombSUM outperform the other composers across all three datasets, while decision tree and random forest exhibited subpar performance.

Keywords: bug localization, information retrieval, replication study, composer

1 Introduction

A software bug refers to an error, fault, or flaw that produces unexpected results or causes a system to behave unexpectedly [1]. A bug may cause the system to crash or become vulnerable to security attacks [2, 3]. Bugs are a common phenomenon. For example, a Mozilla triager complained that "every day, almost 300 bugs appear that need triage" [4]. Considering the severe consequences and frequent occurrences, bugs need to be responded to promptly and coped seriously. To this end, various techniques to assist this process have been suggested, for example, defect prediction [5, 6], bug triaging [7, 8], bug localization [9, 10], and bug fixing [11, 12].

Bug localization, which refers to identifying the parts of source code that cause the bug and need to be changed in order to fix it, is one of the main challenges when solving bugs in practice [13]. However, finding the buggy files from the source code can become a daunting task [14], especially in large projects consisting of thousands of source code files. To help to deal with this issue, researchers proposed several automatic approaches for bug localization [14–17].

Among existing approaches for bug localization, there is a series of them that leverage bug reports for better localization [14, 16, 17], since bug reports often contain rich information that allows us to infer the bug's location. Approaches that utilize the textual content of bug reports are generally described as information retrieval-based bug localization (IRBL). For a given bug report, IRBL finds and ranks code snippets that may be relevant to the bug report [18], which is usually done by calculating the similarity between the bug report and source code [18]. For example, Saha et al. [19] propose the BLUiR approach that extracts structured information (e.g., class names, method names, variable names, and comments) from source code and calculates the textual similarity between the source code and bug reports to retrieve buggy files. However, there exists a lexical gap between bug reports and source code files [20]. The terms used to describe the bug in the bug report may not match the terms used in class names, methods names, variable names, or comments. Not surprisingly, textual similarity by itself will not necessarily yield good results [14].

To improve the performance of bug localization, state-of-the-art approaches leverage multiple sources of information. Wang et al. propose the AmaLgam approach, which combines code structure, similar bug reports, and version history [14]. Another approach, namely BRTracer+, leverages bug reports similarity and stack trace from bug reports for bug localization [21]. Youm et al. integrate stack trace information with all those pieces of information used by AmaLgam [22]. Additionally, AmaLgam+ leverages five sources of information, namely version history, similar bug reports, code structure, stack trace, and reporter information [17].

Rath et al. presented a new approach, named ABLoTS, that leverages not only similar bug reports, version history, code structure, but also similar non-bug reports, like feature requests, enhancements, and tasks, as well as traceability information between bug reports and other types of issues [23]. Rath et al. reused the structure of AmaLgam, but proposed TraceScore to replace the similar bug reports component, and additionally decided to use a decision tree (DT) for dynamically combining the recommendations from the individual components. The experimental evaluation showed that ABLoTS greatly outperforms AmaLgam.

Although the original study by Rath et al. [23] showed encouraging results (with no other state-of-the-art approaches exhibiting better performance [24, 25]), there are no replications in the literature that confirm its outstanding performance. Additionally, there are no studies that investigate whether the performance also holds for other datasets, i.e., that evaluate the generalization of ABLoTS. A replication study is helpful and necessary to verify experimental results from previous studies [26]. They are a key aspect of empirical software engineering, as they bring evidence that observations made can hold (or not) under other conditions [27]. Extensive and independent evaluations are also necessary to reach industrial adoption and practice [28, 29].

In a previous work [30], we present a literal and conceptual replication [31] of the ABLoTS approach. We replicated the experiments as closely as possible to the initial procedures. We also run the experiment on a new Java dataset without changing anything else, to see how well the results hold up. Thus, we first re-implemented TraceScore, the core component of ABLoTS, and checked the replicability of the results on the original dataset. Then, we replicated the overall ABLoTS framework on the original dataset. Additionally, we investigated the TraceScore's and ABLoTS' generalizability on the new Java dataset including 16 more projects comprising 25,893 bug reports and corresponding source code commits. In general, our replication results show that TraceScore is replicable and generalizable under specific settings under a *relaxed cut-off date* (i.e., use fixed date as cut-off date). However, ABLoTS is neither replicable on the original dataset nor on a larger dataset [32]. Specifically, we observed that the implementation of ABLoTS reused a subcomponent from prior work (AmaLgam [14]) that incorrectly sets a cut-off date, which leads to test data leaking into training data.

In this paper, we report an extension of our previous work [30]. While our initial replication focused only on Java projects, in this extension, we introduce a Python dataset, and also other factors constant, to see how well the results hold up on new projects and new programming language. The new dataset and new programming language are based on two extended datasets from SEOSS 33 [32] and BuGL [33].

Additionally to the new dataset, in this work, we also investigate the performance of various fusion methods for combining ABLoTS' three main scoring components.

The contributions of this paper are:

1. An empirical investigation showing that the TraceScore component is replicable and generalizable, thus strengthening confidence that relations between bug reports and feature requests yield useful information for bug localization.
2. A failed attempt to replicate the promising results of the ABLoTS approach, thereby showing that bug localization still needs significant research efforts and is not ready for practical application. Additionally, we present the major reason why replication failed, thereby highlighting the challenge of reusing research results.
3. Valuable findings for further studies on IRBL: 1) Combining different components can yield better results, however, researchers should always be careful about choosing the proper cut-off date. 2) AmaLgam's implementation of BugCache may not be as useful as expected, since they adopted a wrong cut-off date. 3) Preliminary exploration of the effectiveness of different composers reveals that Logistic Regression (LR), fixed weight and CombSUM perform well across the three datasets. In contrast, DT and Random Forest (RF) exhibit poorer performance.
4. Replication package and experimental results[1] to replicate our experiment and evaluate the ABLoTS approach.

Our work is organized according to standard replication report guidelines for software engineering studies [27]. This is an external and independent replication study without any of the authors of the original paper taking part in the replication process.

The rest of this paper is organized as follows. Section 2 summarizes the original study, approach, evaluation, and achieved results. Section 3 elaborates our replication study design, research questions, and dataset. The experimental results are presented and discussed in Section 4. Section 5 discusses threats to validity. Related work is presented in Section 6, followed by the conclusion of this work in Section 7.

2 Replicated Study

This work is a replication of the ABLoTS approach proposed by Rath et al. [23], which consists of four components shown in Fig. 1, namely **Version History Component**, **Similar Reports Component**, **Code Structure Component** and **Composer Component**. In this section, we provide an overview of the ABLoTS approach. We firstly present the whole framework of the ABLoTS approach (Section 2.1), followed by the TraceScore component that is at the center of ABLoTS approach, encapsulated in the **Similar Reports Component** (described in Section 2.2). Then, we present the utilized evaluation metrics (Section 2.3) as well as the dataset as used in the original study (Section 2.4). Finally, we summarize the reported experimental results (Section 2.5).

[1] https://github.com/feifeiniu-se/Replication2

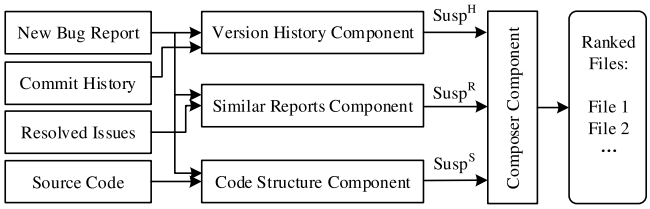


Fig. 1 Components of ABLoTS.

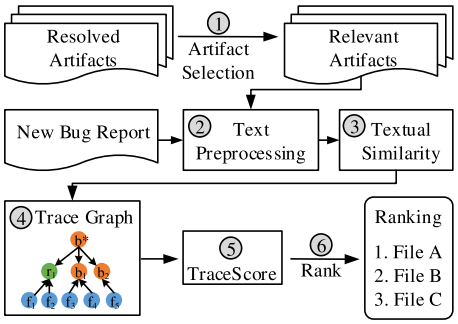


Fig. 2 TraceScore Component.

2.1 ABLoTS Approach

As shown in Fig. 1, the overall ABLoTS approach consists of four components: 1) similar reports component, 2) version history component, 3) code structure component, and 4) composer component. Rath et al. [23] implemented TraceScore as a similar reports component, and they reused the version history component, code structure component, and composer component without changes. They are briefly described below.

Version History Component uses BugCache [5, 34], to predict which files are likely to be buggy in the future. BugCache takes commit history as input and outputs a list of files with a high "suspiciousness" score. To this end, it firstly identifies bug-fixing commits (commits whose commit messages contain the word "fix" or "bug") that were committed within k days prior to the submission of the new bug report $b*$. Then the suspiciousness score of each file f is calculated by Eq. 1, where f is one of the buggy files in commit $c \in C$, t_c is the elapsed time in days between the commit c and when the bug report was filed. k was set to 15 (days) according to Wang et al. [14].

$$Susp^H(f, b*) = \sum_{c \in C \wedge f \in c} \frac{1}{1 + e^{12(1-((k-t_c)/k))}} \quad (1)$$

Similar Reports Component is based on the assumption that similar bugs will be caused by similar source code snippets. Hence, by identifying similar bugs reports and inspecting which files were changed in their bug-fixing commits, one can obtain a list of files indicating the bug location. ABLoTS approach implemented TraceScore

as the similar reports component. Specifically, it introduces a calculation scheme for the similar reports component compared with SimiScore [16].

Code Structure Component leverages BLUiR [19] to identify files from source code space according to the similarity between source code files and bug report $b*$. It outputs a ranked list of files with a suspiciousness score $Susp^S(f, b*)$.

Composer Component aggregates the three suspiciousness scores obtained by the first three components, i.e., $Susp^R$, $Susp^H$, $Susp^S$, and outputs the final results. Instead of adopting a fixed weight scheme for the three scores as done by Wang et al. [14, 17], ABLoTS applied Weka's [35] J48 DT to learn the best combination. For training, the classification algorithm takes $Susp^R(f, b*)$, $Susp^H(f, b*)$, $Susp^S(f, b*)$ as the features, and whether that file f was changed as part of the bug fix or not as the classification result. For each project separately, they trained the classifier on 80% of the bug reports that were resolved and evaluated ABLoTS on the remaining 20% that were resolved after the 80% cut-off deadline.

2.2 TraceScore Component

TraceScore is one of the main components of the ABLoTS approach. It mainly consists of six steps, as shown in Fig. 2. Given a new bug report $b*$, TraceScore takes previously resolved issue reports (including bug reports B and feature requests R) as input. **Step 1** is artifact selection, based on two criteria, namely *time domain* and *number of modified files*. For the *time domain*, bug reports $b \in B$ and feature requests $r \in R$ that are fixed within "one year before $b*$ was filed" to "the date when $b*$ was filed", would be retained. As for *number of modified files*, only bug reports $b \in B$ that modify no more than 10 Java files and feature requests $r \in R$ that modify no more than 20 Java files will be retained. The reasons for adoption of these two criteria and their validity are explained in Section 6 of the original study. **Step 2** utilizes commonly used preprocessing techniques to build a document-term-matrix [36] of the filtered artifacts from Step 1. Then, in **Step 3**, TraceScore calculates the cosine similarity between $b*$ and each artifact. In **Step 4**, a trace graph is created, with $b*$ as the root node, linked to sub-graphs of different artifacts, by the edges indicating textual similarity between $b*$ and each artifact (if there is a trace link between $b*$ and artifact, the edge is set to 1). Each artifact traces further to the files that are part of a corresponding commit in the version control system. In this way, b* is indirectly linked to a potentially large set of source code files, that need subsequent ranking, where the ranking happens on the basis of a **TraceScore** between each file and $b*$ which is calculated by Eq. 2 in **Step 5**. Finally, **Step 6** sorts all the source code files linked to $b*$ according to TraceScore and outputs the ranked list. A higher score indicates a higher likelihood of that file being relevant.

$$Susp^R(s, b*) = \sum_{a_i \in \{a | s \in fix(a)\}} \frac{sim(a_i, b*)^2}{|fix(a_i)|} \tag{2}$$

2.3 Evaluation Metrics

To evaluate the effectiveness of ABLoTS, Rath et al. adopted three metrics:

Top *k* [37] measures the percentage of bug reports in which at least one of the buggy files is in top k ranked files, where k=1, 5, 10.

Mean Average Precision (MAP) [36] is calculated as the mean of the Average Precision over all queries. Average Precision of a given bug report aggregates precision of positively recommended files as:

$$AP = \sum_{i=1}^{N} \frac{P(i) * pos(i)}{\#\,of\,positive\,instances} \tag{3}$$

where i is a rank of the ranked files, N is the number of ranked files and $pos(i) \in \{0,1\}$ indicates whether the ith file is a buggy file or not. $P(i)$ is the precision at a given top i files:

$$P(i) = \frac{\#\,of\,buggy\,files\,in\,top\,i}{i} \tag{4}$$

Mean Reciprocal Rank (MRR) [38] computes the average of the reciprocal of the positions of the first correctly located buggy file in the ranked files, following this equation:

$$MRR = \frac{1}{Q} \sum_{|Q|}^{q=1} \frac{1}{rank_i} \tag{5}$$

2.4 Original Dataset

In the original study, Rath et al. contributed a dataset [39] consisting of 15 open source Java projects with 13,581 bug reports and 9,219 feature requests. Firstly, they collected issues reports (i.e., bug reports and feature requests), as well as the dependency trace links from Jira [40], and downloaded source code of these projects from GitHub [41]. Then the heuristic proposed in [42] was applied to create links between issues and commits. The ABLoTS approach was evaluated based on this dataset.

2.5 Achieved Performance Originally Reported

The reported performance by ABLoTS is shown in Table 1, which is the average on 15 projects. The average MAP and MRR of TraceScore is 20.2% and 26%, while that of ABLoTS is 48.8% and 54.5%, respectively. Specifically, ABLoTS exhibits the ability to accurately identify at least one buggy file for 48.7% of the bug reports when considering only the top-ranked files. Moreover, in the top 5 ranked files, ABLoTS successfully identifies at least one buggy file for 64.9% of the bug reports. Rath et al. revealed that TraceScore benefits from leveraging non-bug issues as well as traceability information. It can outperform two state-of-the-art similar reports based approaches: SimiScore [16] and CollabScore [43]. The overall ABLoTS framework leverages DT as composer and outperforms the AmaLgam framework [14] which leverages fixed weight composer.

Table 1 Original, reported results [23]

Algorithm	MAP	MRR	Top 1	Top 5	Top 10
TraceScore	0.202	0.260	0.174	0.350	0.436
ABLoTS	0.488	0.545	0.487	0.610	0.649

3 Replication Methodology

The goal of this replication study is to investigate whether the results based on the TraceScore component and ABLoTS approach are replicable and generalizable to different projects and languages. Furthermore, we explore the performance of different composers on aggregating the three suspiciousness scores. To this end, we 1) replicate the component and the approach on a subset of the original dataset; 2) apply the component and the approach on 16 more Java projects and 12 more Python projects; and 3) evaluate the performance of different composers by fusing the three components in diverse ways. We provide a description of the extended datasets in Section 3.1. Subsequently, we outline our research questions in Section 3.2, followed by detailed explanations of the replication methodology (Section 3.3 through Section 3.5). We reuse the same evaluation metrics as Rath et al., including MAP, MRR, and Top 1, 3, and 5 (Section 2.3).

This study is considered to be an external [27] replication study of the original study, since none of the authors took part in the replication process. However, we reused 11 projects of the original dataset to verify the results.[2]

3.1 Dataset

For the replicability validation, we reuse the dataset provided for replication by Rath et al. [39], which we refer to as "Original-Java". However, by the time we carried out the replication study, many commits from four projects (i.e., Axis2, Hadoop, Infinispan, and Pig) were no longer available on GitHub, and neither were part of the original replication package. Hence, as we could not obtain the complete commit history for BugCache, we excluded these four projects from the analysis in this paper.

As stated in Section 3.2, the generalizability evaluation comes from two aspects: 1) additional dataset of the same language and 2) additional dataset of different programming language. To this end, we picked two additional datasets: 1) SEOSS 33 dataset [32] and 2) BuGL dataset [33]. The SEOSS 33 dataset includes 18 additional projects and 36,482 bug reports out of which we could not use 2 projects due to the same issue of non-accessible commits. We refer to this dataset as "Extended-Java". This extended dataset also includes the 15 original projects from the replication package [39]. We choose this dataset because it not only links bug reports to commit code change, but also includes traceability information between bug reports and non-bug issues, which caters to our needs perfectly. The BuGL dataset is a large-scale cross-language dataset for bug localization. BuGL consists of more than 10,000 bug reports drawn from open source projects written in four programming languages, namely C, C++, Java, and Python. In this research, we only select Python projects in BuGL

[2]The other four projects from the original dataset were excluded due to some missing commits on GitHub.

dataset, which includes 12 open source projects with 1,289 bug reports. We refer to this dataset as "Extended-Python".

The reason for choosing Python for our study is that it is one of the most popular programming languages [44, 45], even outperformed Java according to the 2023 Stack Overflow Developer Survey. However, there is no feature requests or traceability information between issues contained in the Python dataset. Thus, we can only leverage the similar bug reports for TraceScore, which may bring bias to our results. Details about the extended dataset are shown in Table 2, which shows that the Java projects tend to be larger compared to Python projects when considering the number of bug reports and source code files.

Apart from the information in the dataset, we additionally collected version information for each project. In each commit, developers may modify a file, add a new file, or remove an old file. Removed files are obsolete and should not appear in the recommendation of a new bug report. However, the similar reports component leverages historical issues, which may be pointing to no longer existing files. In this way, they may bias the prediction results. To address this issue, we determine for each commit which files exist just prior to this commit. Files in this set can only be used as the candidates to recommend the bug's location.

Table 2 Characteristics of the Extended-Java and Extended-Python Dataset.

Java Projects	# Bug Reports	# Non-bug Reports	# Files	Python Projects	# Bug Reports	# Files
ARCHIVA	371	411	1024	ERTBOT	170	345
CASSANDRA	3571	2813	3595	COMPOSE	155	82
ERRAI	267	194	3703	DJANGO_R._F.	153	155
FLINK	1350	2351	13192	FLASK	53	73
GROOVY	1933	1017	1375	KERAS	51	198
HBASE	4581	5171	4657	MITMPROXY	107	371
HIBERNATE	1947	1706	11720	PIPENV	136	857
HIVE	4776	4326	4657	REQUESTS	75	35
JBOSS-T.-M.	331	489	4204	SCIKIT-LEARN	1082	767
KAFKA	639	1149	3423	SCRAPY	112	304
LUCENE	3773	5324	5482	SPACY	55	643
MAVEN	760	574	1381	TORNADO	40	114
RESTEASY	345	228	3866			
SPARK	328	7022	1055			
SWITCHYARD	451	759	2953			
ZOOKEEPER	470	471	900			

3.2 Research Questions

The replication study aims mainly at answering the following research questions (RQ):

- RQ1. How effective is TraceScore in identifying bug-relevant source code files?
  - RQ1.1 Are we able to replicate the original performance of the TraceScore component?

  - RQ1.2 Does the TraceScore component yield similar performance when applied to additional Java projects?
  - RQ1.3 How does TraceScore perform on Python projects?
- RQ2. How effective is ABLoTS for bug localization?
  - RQ2.1 Are we able to replicate the results of the ABLoTS approach?
  - RQ2.2 Does the ABLoTS approach yield similar performance when applied to additional Java projects?
  - RQ2.3 How does ABLoTS perform on Python projects?
- RQ3. How do different composers perform when combining three components?

In our study, RQ1 and RQ2 are adapted from those addressed in the original study, which involve the main contribution of the Rath et al. study [23]. Specifically, RQ1 is adapted from the first question of the original study, which evaluates the TraceScore component. RQ2 is adapted from fourth question of the original study, which evaluates the ABLoTS approach. For each research question, we analyze two dimensions, namely 1) replicability and 2) generalizability. The core difference of the dimensions is the dataset being used for evaluation. For the replicability validation, we evaluate on the same projects with the original study, to see if our replication results are consistent with the original results. As for the generalizability validation, it can be divided into two levels. The first level focuses on the same programming language, namely Java, but on 16 additional projects, to see if the approach is applicable to other projects as well. The second level extends to different programming language, particularly focusing on 12 additional Python projects, to see how the approach performs on different programming languages. The other two RQs in the original study (i.e., second and third questions) mainly investigate the effectiveness of artifacts selection, which is irrelevant of our goal, so we do not include them in this study.

Our RQ3 aims at investigating the performance of different composers at combining the three suspiciousness scores, including fixed weight, Multi-layer Perception (MLP), DT, RF, LR, CombSUM, CombMNZ, CombANZ, CorrB, and Borda Count. Aside from merely aiming to improve the composition mechanism, the reason for studying the composer performance emerges from our finding in RQ2 that the original ABLoTS composer was trained on leaking data (i.e., the wrong cut-off date for BugCache) and hence is expected to no longer work effectively when data leakage does not occur. We are thus interested whether an improved composer may achieve the original ABLoTS performance or whether the fixed weights from the AmaLgam approach yield the best possible scoring result.

3.3 Procedure to answer RQ1

This research question mainly focuses on the main contribution of Rath et al., i.e., TraceScore for the similar reports component. Since the original source code is not available, we followed the procedures proposed in the original paper (illustrated in Section 2.2) as close as possible to duplicate all facets of TraceScore. Specifically, for each project, we sort all the issues according to the resolved date. Then we split all the bug reports 80:20, with the latter 20% used as the test set to recommend buggy files.

Similar to the original study, we filter the number of related bug reports and features as well as commits based on age and size from which to obtain a recommendation. For each bug report $b*$ in the test set, we consider only bug reports (and features) b that occurred before $b*$ as determined by the following condition: $b.fixed\_date > b*.created\_date - 365\,days$. However, we are of the opinion that there is another constraint that also should be satisfied: $b.fixed\_date < b*.created\_date$, which means that only bug reports fixed before $b*$ were filed should be retained. These two settings describe the following two recommendation situations: the former describes the bug localization mechanism called shortly before fixing the bug, close to the bug report's closing date, while the latter describes a recommendation immediately made upon bug creation. For our replication, we were unable to determine whether the authors only adopted the first constraint (denoted as *relaxed cut-off date*) or adopted both constraints (denoted as *strict cut-off date*). We conducted the replication with both relaxed and strict cut-off date to understand the impact the additional constraint has on the results. Then, we select bug reports/features requests according to the *number of modified files* identified in their commits. We exclude issues that modify more than 10 files for bug reports and more than 20 files for non-bug reports. We then build up the trace graph from these issue subset as shown in Fig. 2 in Step 4. The edges between the root node $b*$ and other artifacts are calculated using cosine similarity [46]. When an issue explicitly links to another issue, then the link weight overrides the cosine similarity and becomes 1. With the trace graph, the TraceScore between each file node s and $b*$, $TraceScore(s, b*)$ is calculated. Finally, all the files according to their tracescore, we will get the ranked list for $b*$.

Then we evaluate our replication on the two extended datasets. For the extended datasets, we apply preprocessing (Step 2 in Section 2.2) to be consistent with the original dataset and to fit the replication. Specifically, for each issue, we preprocess the text including both summary and description (Step 2 in Section 2.2). We utilize NLTK library [47] in Python for preprocessing, including stop words removal, camel case splitting, lower casing, and stemming. Then the preprocessed texts are converted into TF-IDF [48] vector with the sklearn library [49]. For the source code, we exclude non-source code files based on the file name extension and only retain source code files ("*.java" for Extended-Java and "*.py" for Extended-Python). For each file changed in each commit, the extended dataset contains the old name and the new name for this file. Following the original study, we only utilize the new name for each file, which means removed files will be excluded for each commit.

3.4 Procedure to answer RQ2

The ABLoTS approach is essentially an ensemble of three components, namely similar reports, version history, and code structure, as shown in Fig. 1. As described in the original study, the ABLoTS approach is an evolved version of AmaLgam [14] with two main differences: 1) it replaces the similar reports component with TraceScore; and 2) it applies a dynamic suspiciousness score combination (instead of the former static one). At the time of conducting the replication, there is no available implementation for the whole framework. We, therefore, replicated the framework along the following lines.

Version History. As mentioned in Section 2.1, the version history component is implemented by BugCache, which is proposed by Kim et al. [5]. BugCache maintains the modification history of files to predict buggy-prone files in the future. It proved that more recently and frequently modified files are more likely to be buggy in the future. Rahman et al. proposed a simpler version of BugCache [34], which only maintains a short history of file modification. Google's developers adapted Rahman et al.'s algorithm on their large systems [50, 51]. AmaLgam adapted Google's well-tested algorithm with a version history component. We reused AmaLgam's implementation of BugCache,[3] but made the following modifications:

1. The BugCache version used in AmaLgam was written in Java, while we manually translated it to Python to be compatible with our implementation.
2. In their paper, Wang et al. [14] explain that the approach identifies commits that are committed 15 days before *the new bug report is created*. However, after checking the source code, we found that the implementation utilized the bug report's *resolved date* as the cut-off date to obtain previously committed commits within 15 days. We contacted the authors, and they agreed that the bug report's *creation dates* should have been adopted. Therefore, in our implementation, we used *the creation date* for all our experiments.
3. To identify bug-fixing commits, Wang et al. proposed that commit logs should match regular expression regex: *(.*fix.*)|(.*bug.*)*. Considering that some programming languages (e.g., Java and Python) are case-sensitive, we firstly convert commit logs into lowercase, which is missing in the original implementation. What is more, according to our observation of the dataset, some bug-fixing commit logs may not contain keywords like "fix" or "bug". However, they might start with the bug report's ID. To this end, we also include commits that start with any bug ID in their logs, to identify bug-fixing commits more accurately. AmaLgam's authors also agree with us on this. This adapted selection of commits only affects the commits used for BugCache, but not any other component in ABLoTS.

Code Structure. Code Structure metrics are obtained with BLUiR [19], which calculates the similarity between a new bug report and the code structure of a source code file. It takes the summary and description of a bug report as two separate parts and extracts class names, method names, variable names, and comments of a source code file represented as an Abstract Syntax Tree (AST). Then it indexes and searches buggy files based on the Indri toolkit [52]. In this paper, instead of replicating our own BLUiR tool, we used the implementation[4] from an empirical study by Lee et al. [24] to obtain the $Susp^S(s, b*)$ score. Lee et al. implementation were originally designed for Java language. Since our experiments also involve Python files, we parsed Python files with the AST module to extract the code structure. Then we used the Indri[5] tool to calculate the similarity between bug reports and code structure of Python files.

Composer. ABLoTS applied the J48 DT with default pruning settings to classify source code files for bug reports. Specifically, for each $b*$, there are multiple candidate source code files s for recommendation. For each (s,b*), there will be a label $C \in$

[3] https://sites.google.com/view/mambalab/projects/amalgam
[4] https://github.com/exatoa/bench4bl
[5] https://sourceforge.net/projects/lemur/files/lemur/

$\{true, false\}$ indicating whether the file s is modified to fix b or not. For training, the classifier takes the $Susp^R(s, b*)$, $Susp^H(s, b*)$, and $Susp^S(s, b*)$ scores for each (s, b*) as feature and C as the label. For test data, instead of outputting a label indicating true or false, the probability of s being *true* (i.e., s is modified by $b*$) is utilized. Then for each $b*$, all the files are ranked according to the probability score.

For each project, Rath et al. sorted all the bug reports by resolved date and took the first 80% bug reports as training data, and the remaining 20% as test data. To mitigate the influence of imbalanced training data, ABLoTS used Weka's sub-sampling to under-sampling the training data.

Since our replication is based on Python, we chose the popular open source Python library sklearn [49] for the DT classifier, and *RandomUnderSampler* in the Imblearn library [53] for under-sampling. Essentially they are the same algorithm with the original study, but just implemented by different libraries. We assume that this will not cause significant difference to the result as we used exactly the same training data as in the original paper (i.e., rather than sampling our own set of training data we utilized the precalculated suspiciousness scores and classification result from the replication package to obtain a trained DT).

We applied the same procedure on the original dataset and the extended datasets. After completing the replication of the entire framework on the original dataset, we assess the performance of ABLoTS on the extended Java and Python datasets to evaluate how it performs.

3.5 Procedure to answer RQ3

Rath et al. [23] chose DT as the composer and claimed that DT outperformed fixed weight utilized by AmaLgam [14, 17]. The essence of ABLoTS lies in adopting an aggregation strategy, where the three different scores: $Susp^R$, $Susp^H$, $Susp^S$ for each $<b*, s>$ pair are aggregated to obtain the final relevance score. Since there also exist other supervised and unsupervised aggregation strategies, in this study, we would like to explore the performance of different strategies, including unsupervised: fixed weight, CombSUM [54, 55], CombMNZ [54, 55], CombANZ [54, 55], CorrB [56] and Borda count [57], and supervised: MLP, DT, RF and LR. CombSUM, CombMNZ, CombANZ, and Borda count are also rank fusion methods that have been investigated for fusing fault localizers [58]. Different methods may be more suitable for different scenarios. Thus, it is important to choose the most appropriate aggregation approach.

Fixed Weight has been proved to be effective for IRBL [14, 17, 22]. The suspiciousness score for the source code file s is calculated according to Eq. 6 and Eq. 7, where the value of a and b are set to 0.2 and 0.3 as per prior work.

$$Susp^{R,S}(s) = a * Susp^R(s) + (1 - a) * Susp^S(s) \tag{6}$$

$$\begin{aligned} &Susp^{R,S,H}(s) = b * Susp^H(s) + \\ &(1 - b) * Susp^{R,S}(s) \end{aligned} \tag{7}$$

CombSUM [54, 55] is a simple rank fusion method where the scores of documents from different lists are summed, and the documents are ranked based on the total sum. While straightforward, CombSUM assumes that all methods contribute equally, which may not always be the case.

CombANZ is another rank fusion method that combines different scores by computing the average of the non-zero scores.

CombMNZ [54, 55] is a variant of CombANZ. It involves multiplying the summation of scores for a given element by the number of techniques that assign a non-zero score to that element.

CorrB [56] is a correlation-based method that calculates the weight of a technique by assessing the overlap of its list of the top-N most suspicious program elements with lists generated by other techniques.

Borda Count [57] is a popular rank fusion method. In Borda count, each document in a ranked list receives a score equal to the sum of the positions it holds in the individual lists. The document with the highest Borda score is ranked first in the integrated list.

Apart from the above explained unsupervised rank fusion methods, there are also supervised methods that have been used to learn feature importance for classification problems. In this study, we use the commonly used classification algorithms: MLP, RF and LR. For each bug report and source code file pair $<b*,s>$, if s is related to $b*$, then the ground truth is set to be 1, otherwise 0. The optimization function of classification algorithms is to learn if s is related to $b*$.

4 Results and Discussion

We present below the results of our replication study, organized for answering each RQ. Then, we discuss the overall findings and implication or our work.

4.1 RQ1. How effective is TraceScore in identifying bug-relevant source code files?

RQ1.1: Replicability. We carried out the replication according to Section 3.3. Results on the original dataset are as shown in Table 3. The performance impact of using the *strict cut-off date* is on average around 17% lower than using the *relaxed cut-off date*.

To find out which implementation most likely was adopted by the original implementation, we performed a pairwise t-test on the 11 projects, comparing both replication results against the reported results in [23] to establish statistically whether these results can be considered to be the same. According to the pairwise t-test, the *relaxed cut-off date* is closer to the original implementation. The pairwise t-test results presented in Table 4 show that for MRR, Top1, Top5, and Top10, there is no significant difference while for MAP. Thus, we have to reject the null hypothesis for the *relaxed cut-off date*: the average MAP reported by Rath et al. is 32% higher than our replication result. For the remaining four evaluation metrics, there is no significant difference, with the mean values are statistically the same. So we conclude that with the *relaxed cut-off date* TraceScore can be considered replicable, while with the *strict*

Table 3 TraceScore performance on the original dataset.

Relaxed Cut-off Date					
PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
DERBY	0.124	0.240	0.149	0.340	0.404
DROOLS	0.183	0.383	0.276	0.502	0.615
HORNETQ	0.134	0.241	0.130	0.352	0.481
IZPACK	0.170	0.229	0.156	0.328	0.422
KEYCLOAK	0.125	0.234	0.152	0.323	0.418
LOG4J2	0.182	0.271	0.191	0.360	0.416
RAILO	0.138	0.202	0.117	0.267	0.350
SEAM2	0.134	0.195	0.141	0.244	0.288
TEIID	0.194	0.278	0.188	0.385	0.465
WELD	0.102	0.208	0.098	0.312	0.420
WILDFLY	0.108	0.185	0.116	0.268	0.326
Average	0.145	0.242	0.156	0.335	0.419
Strict Cut-off Date					
PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
DERBY	0.084	0.158	0.096	0.219	0.272
DROOLS	0.171	0.37	0.265	0.467	0.603
HORNETQ	0.105	0.207	0.093	0.315	0.444
IZPACK	0.101	0.152	0.094	0.219	0.297
KEYCLOAK	0.081	0.16	0.082	0.241	0.323
LOG4J2	0.165	0.256	0.18	0.315	0.382
RAILO	0.131	0.194	0.117	0.25	0.35
SEAM2	0.099	0.159	0.103	0.212	0.263
TEIID	0.140	0.222	0.135	0.331	0.412
WELD	0.103	0.201	0.098	0.304	0.411
WILDFLY	0.085	0.146	0.087	0.217	0.268
Average	0.115	0.202	0.123	0.281	0.366

cut-off date it cannot be considered replicable, since we cannot achieve statistically comparable or better results.

To give benefit to doubt, we adopted the *relaxed cut-off date* for the remainder of the replication and generalization investigations. However, in practice, the choice between *relaxed cut-off date* and *strict cut-off date* is artificial as only commits available at the time the bug localization mechanism is applied are considered for producing the recommendation.

Table 4 Pairwise t-test between relax constraint result and original result.

Metrics	Pairs		Deviation	P value
	Original	Replication		
MAP	0.191	0.145	0.05	0.000**
MRR	0.248	0.242	0.01	0.522
Top 1	0.163	0.156	0.01	0.407
Top 5	0.336	0.335	0.00	0.924
Top 10	0.419	0.419	0.00	0.99

*p <0.05 **p <0.01

RQ1.2 & RQ1.3: Generalizability. The evaluation results based on the extended Java and Python dataset are shown in Table 5. The average MAP, MRR, Top 1, Top 5 and Top 10 are 18.3%, 28.4%, 19.6%, 38.4%, 47.3% on Java dataset, and 25.6%, 34.6%, 23.5%, 48.4%, 59.2% on Python dataset, respectively. The average results are around 20% (Top 1) $\sim$ 40% (MAP) higher on Python projects than on Java projects. The MAP values ranges from 4.4% to 32.5% in the Java dataset, and from 5.2% to 50.5% in the Python dataset. The MRR values stretches from 11.4% to 47.3% for the Java dataset, and lies between 11.4% and 56.9% for the Python dataset.

Table 5 TraceScore performance on the extended datasets.

Extended-Java dataset					
PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
ARCHIVA	0.134	0.22	0.147	0.28	0.413
CASSANDRA	0.218	0.333	0.222	0.453	0.551
ERRAI	0.059	0.15	0.093	0.204	0.296
FLINK	0.18	0.305	0.207	0.415	0.522
GROOVY	0.325	0.393	0.271	0.522	0.625
HBASE	0.236	0.352	0.25	0.455	0.561
HIBERNATE	0.118	0.231	0.172	0.3	0.359
HIVE	0.264	0.38	0.267	0.506	0.599
JBOSS-T.-M.	0.136	0.247	0.164	0.373	0.433
KAFKA	0.296	0.473	0.367	0.578	0.688
LUCENE	0.201	0.32	0.228	0.419	0.494
MAVEN	0.162	0.222	0.132	0.316	0.382
RESTEASY	0.101	0.202	0.101	0.348	0.435
SPARK	0.31	0.383	0.273	0.545	0.576
SWITCHYARD	0.044	0.114	0.088	0.121	0.198
ZOOKEEPER	0.149	0.226	0.149	0.309	0.436
Average	**0.183**	**0.284**	**0.196**	**0.384**	**0.473**
Extended-Python dataset					
PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
CERTBOT	0.264	0.369	0.235	0.500	0.706
COMPOSE	0.325	0.423	0.290	0.548	0.677
DJANGO_R._F.	0.505	0.555	0.484	0.613	0.710
FLASK	0.279	0.365	0.182	0.636	0.727
KERAS	0.149	0.171	0.091	0.273	0.455
MITMPROXY	0.133	0.247	0.182	0.364	0.409
PIPENV	0.310	0.569	0.464	0.714	0.786
REQUESTS	0.328	0.343	0.200	0.600	0.600
SCIKIT-LEARN	0.305	0.373	0.290	0.465	0.530
SCRAPY	0.190	0.297	0.217	0.304	0.522
SPACY	0.235	0.321	0.182	0.545	0.727
TORNADO	0.052	0.114	0.000	0.250	0.250
Average	**0.256**	**0.346**	**0.235**	**0.484**	**0.592**

In order to confirm if there is a difference between the distribution of the original results and extended results, we leverage the two-sample Kolmogorov-Smirnov test (K-S test) [59], which is used to test whether two samples come from the same underlying one-dimensional probability distribution. For each evaluation metric, we perform a two-sample K-S test, with one sample being the results from the original dataset and

the other sample being the results from one of the two extended datasets. Results are shown in the "TraceScore-J" and "TraceScore-P" columns of Table 6 for the Extended-Java dataset and the Extended-Python dataset, individually. On the Extended-Java dataset, all the p-values are greater than 5%, indicating that the two samples come from the same distribution. For the Extended-Python dataset, however, all the p-values are less than 5%, thus we cannot assume that the two samples come from the same distribution. The box plot in Fig. 3 shows the data value on all five metrics, from which we can see that on the Extended-Java dataset, TraceScore yields slightly higher median and wider variations, while on the Extended-Python dataset, TraceScore yields much higher median, maximum, and minimum, which indicates TraceScore yields higher performance on the Extended-Python dataset than on the original Java dataset. The average over the Extended-Java dataset is about 12% (Top 10) $\sim$ 27% (MAP) higher than on the original dataset, while that of the Extended-Python dataset is about 41% (Top 10) $\sim$ 77% (MAP) higher than on the original dataset.

We also investigate the improvement of TraceScore over the same baseline as in the original paper.[6] With SimiScore [16] as baseline, we obtain the improvement of TraceScore over SimiScore on both original and Extended-Java dataset. The "Improvement" column of Table 6 shows the results of the K-S test. Given the p-values, the improvement of MRR, Top 1, and Top 10 on the original dataset and the extended dataset are very likely to come from different distributions. To this end, from the box plot in Fig. 4 for improvement, we can observe a much higher median, maximum, and minimum, which indicates TraceScore yields higher performance improvement on the Extended-Java dataset.

We can therefore conclude that the performance of TraceScore also holds for a larger Java dataset or for another Python dataset, and we gain confidence that TraceScore's performance is generally achievable.

Table 6 K-S test result.

	TraceScore-Java		TraceScore-Python		Improvement	
Metrics	K-S test	P value	K-S test	P value	K-S test	P value
MAP	0.438	0.124	0.667	0.004	0.500	0.054
MRR	0.409	0.175	0.659	0.006	0.693	0.002
Top 1	0.409	0.175	0.561	0.039	0.625	0.007
Top 5	0.409	0.175	0.576	0.023	0.443	0.115
Top 10	0.415	0.159	0.659	0.006	0.540	0.028

Answering RQ1: *Under the relax cut-off date constraint, TraceScore is replicable and also can be generalized to both Java and Python extended datasets. However, under the strict cut-off date constraint, we cannot claim replicability as the performance is significantly lower than reported.*

[6]Since the main improvement of TraceScore over SimiScore lies in the leveraging of requirements and traceability information, but there is no such information in Extended-Python dataset, we do not compare the improvement over the Extended-Python dataset

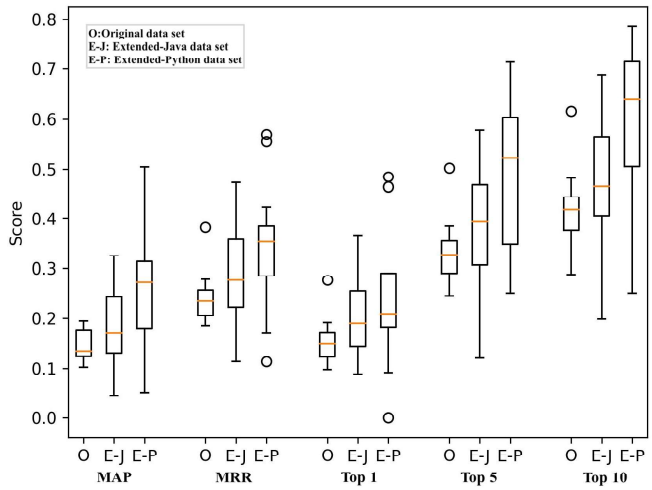


Fig. 3 Box plots of TraceScore on original and extended Java and Python datasets.

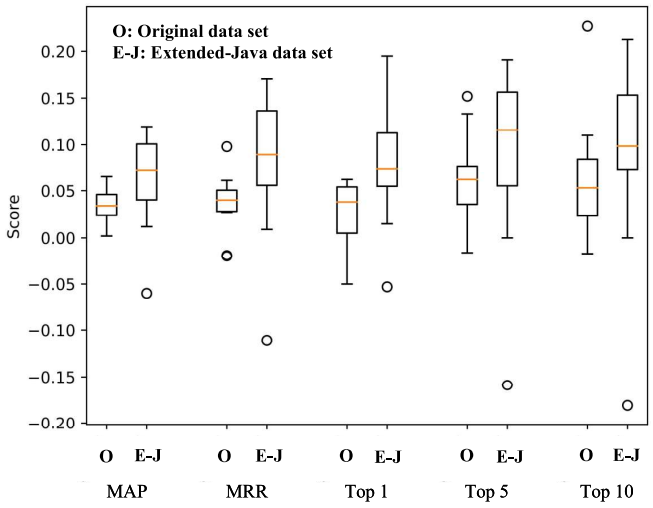


Fig. 4 Box plots illustrating the improvement of TraceScore over SimiScore on both the original and extended Java datasets.

4.2 RQ2. How effective is ABLoTS for bug localization?

RQ2.1: Replicability. ABLoTS's performance results on the original dataset are shown in Table 7. Compared to the results reported in the original paper (cf. Table 1) we observe that our replication produces far worse results. MAP and MRR are below 10% for most projects. ABLoTS, which combines three scores, namely $Susp^R$, $Susp^H$, and $Susp^S$, does not even achieve the same results as the single $Susp^R$ score. This

counterintuitive result motivated us to investigate in more detail how this outcome can be explained.

For the strict replication, we trained the DT on the intermediate three scores (i.e., $Susp^R$, $Susp^H$, $Susp^S$) made available by Rath et al. in their replication package, For comparison, we also trained a separate DT from our own sample of files, their suspiciousness scores, and bug reports. Note that the original replication package just provided tuples of suspiciousness scores and classification results, but not which bug report and which files were used to obtain those suspiciousness scores. We, however, applied the same sampling criteria.

We inspected the original DT (i.e., the one obtained from the replication data) to obtain the average feature importance (non-normalized) of each component: 0.037 for BLUiR, 0.377 for BugCache, and 0.018 for TraceScore. This indicates that BugCache almost exclusively determines the final classification result. In contrast, in the AmaLgam approach, which was used as a baseline for ABLoTS, the authors empirically set fixed weights for the three suspiciousness scores, which are 0.56 for BLUiR, 0.3 for BugCache and 0.14 for TraceScore. Our DT trained from scratch exhibited the following (non-normalized) feature importance: 0.243 for BLUiR, 0.007 for BugCache, 0.037 for TraceScore, which still does not yield as good results (see Table 7) as the fixed weights determined for AmaLgam.

This discrepancy in feature importance values helped us identify the root cause for the difference in performance results. Rath et al. adopted the implementation of BugCache by Wang et al. [14], where the bug report's fixed date was utilized for the cut-off date, as shown in Fig. 5. If one or more bug-fixing commits occurred within 15 days prior to the fixed date, BugCache would recommend the files within these commits (i.e., potentially exactly those files that were changed to fix the bug). However, in a realistic bug localization situation, any file recommendation would only be useful before any of those commits. Thus, for correct evaluation, these commits must not be used.

Fig. 5 illustrates such a situation. There is a bug report "HORNETQ-1301" created on 2014-01-09, and fixed on 2014-01-14. Two commits c_6 and c_7 were committed to fix this bug between the created date and fixed date, on 2014-01-09. When BugCache adopts the fixed date as the cut-off date and identifies bug-fixing commits within 15 days, then c_4, c_5, c_6, and c_7 would be taken into consideration and result in a high $Susp^H$ score, according to Eq. 1. Doing so, the DT would learn that the scores by BugCache are very indicative of the actual classification result and hence assign it a high feature importance. However, in practice, c_6 and c_7 are unknown for predicting bug report "HORNETQ-1301", they are foreknowledge about the bug. The right way of implementing BugCache is using the creation date, or any date before the bug's first partial fix implementation. After contacting the authors of both ABLoTS and AmaLgam, AmaLgam's authors stated that they agreed with our finding and that they adopted the wrong date, while authors of ABLoTS stated that they directly reused AmaLgam's implementation.

The incorrectly derived $Susp^H$ scores thus greatly boost the result of the DT. When we utilized BugCache in the correct manner (i.e., use the created date as the cut-off date), DT did not yield results even close to the original performance (even when

applying hyperparameter tuning). For comparison, we adopted AmaLgam's composer with a fixed weight for each component: 0.56 for BLUiR, 0.3 for BugCache, and 0.14 for TraceScore. The results of the fixed weight composer are shown in Table 8, the average MAP, MRR, Top 1. Top 5, and Top 10 are 29.8%, 43.3%, 32%, 56.3% and 64%, respectively. Compared to TraceScore, the results have been improved by 105.8%, 78.7%, 105.2%, 68.4%, and 52.9%, respectively.

Aside from the DT feature importance values, a second discrepancy emerged when we investigated the evaluation dataset. In the replication package, the intermediary suspiciousness scores were provided not only as a training set for the DT but also as an evaluation set (i.e., the remaining 20%). When we trained and evaluated with these two datasets, we could replicate the results. However, as outlined above, when obtaining the suspiciousness scores ourselves, we could not. The discrepancy we found was that the evaluation dataset contained far fewer evaluation data points (i.e., suspiciousness scores with their classification ground truth) than these projects contained source code files. In other words, for a particular bug, not all source code files were utilized for evaluation but just a subset. Across all projects, the number of candidates ranges from 60 to 70, regardless of actual number of files in the respective project. For the project HORNETQ, for example, even when we select only files for which a TraceScore suspiciousness score and a BLUiR suspiciousness score exist, we obtain around 4500 file candidates. In addition, for some of these files the evaluation dataset does not provide any of the three suspiciousness scores at all, just the classification result. Hence, we could not establish how these file candidates have been filtered and why only a subset has been chosen. The paper does not describe this aspect, but rather refers to the evaluation design of Amalgam.

In summary, we found that ABLoTS adopted the wrong cut-off date for BugCache due to having reused the component and configuration from AmaLgam without further investigation, resulting in the incorrect $Susp^H$ scores. Hence, we conclude that ABLoTS performance cannot be replicated.

Table 7 ABLoTS performance on original dataset.

PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
DERBY	0.076	0.111	0.02	0.171	0.326
DROOLS	0.049	0.06	0.023	0.054	0.097
HORNETQ	0.057	0.067	0	0.056	0.185
IZPACK	0.086	0.11	0.016	0.172	0.375
KEYCLOAK	0.029	0.05	0.006	0.044	0.101
LOG4J2	0.065	0.072	0.011	0.067	0.146
RAILO	0.06	0.077	0	0.1	0.283
SEAM2	0.08	0.105	0.019	0.179	0.333
TEIID	0.056	0.079	0.015	0.104	0.231
WELD	0.02	0.024	0	0.018	0.027
WILDFLY	0.03	0.04	0.007	0.036	0.087
Average	**0.055**	**0.072**	**0.011**	**0.091**	**0.199**

RQ2.2 & RQ2.3: Generalizability. Since the evaluation results of the original paper presenting ABLoTS are not replicable, exploring its performance on the extended dataset for generalizability evaluation would yield little insight. However, in

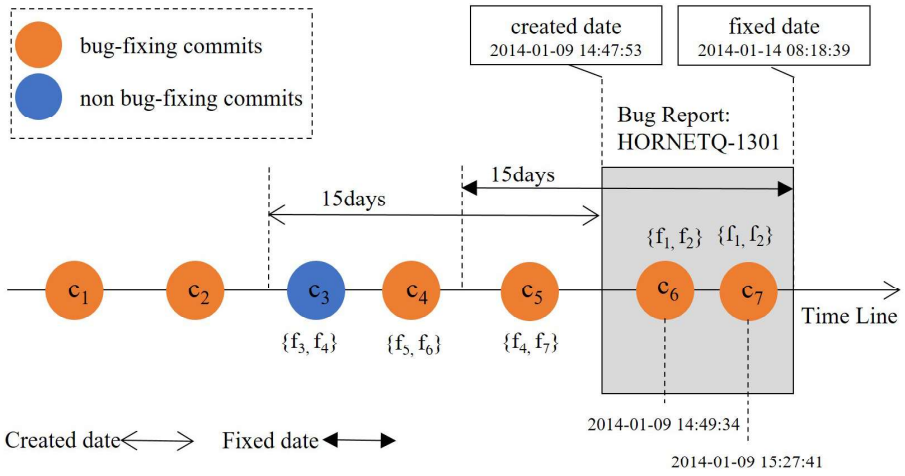


Fig. 5 BugCache using created date VS using fixed date.

Table 8 Fixed weight composer on original dataset.

PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
DERBY	0.312	0.478	0.36	0.615	0.725
DROOLS	0.272	0.464	0.339	0.607	0.712
HORNETQ	0.37	0.555	0.426	0.704	0.778
IZPACK	0.37	0.493	0.391	0.594	0.672
KEYCLOAK	0.234	0.377	0.247	0.525	0.595
LOG4J2	0.391	0.541	0.416	0.719	0.753
RAILO	0.286	0.398	0.267	0.567	0.65
SEAM2	0.339	0.402	0.308	0.532	0.583
TEIID	0.12	0.169	0.1	0.208	0.296
WELD	0.252	0.445	0.33	0.562	0.634
WILDFLY	0.334	0.441	0.333	0.565	0.645
Average	**0.298**	**0.433**	**0.320**	**0.563**	**0.640**

order to explore how TraceScore would perform when jointly used with the other two components, like in AmaLgam [14, 17], we applied a fixed weight to aggregate the three scores. That is, the suspiciousness score for the source code file s is calculated according to Eq. 7, where the value of a and b are set to 0.2 and 0.3 as per prior work.

The results of fixed weight are shown in Table 9. On the additional 16 Java projects, the fixed weight composer can achieve an average MAP, MRR, Top 1, Top 5, Top 10 as 34.4%, 47.7%, 35.6%, 62.1% and 71.4%, which improves over the single TraceScore by 87.8%, 67.8%, 81.8%, 61.6% and 50.8%, respectively. Compared to the results on the original dataset, the average evaluation results over the extended dataset are 10% $\sim$ 16% higher (e.g., the average MAP is 34.4 vs 20.2). On the additional Python projects, the average MAP, MRR, Top 1, Top 5, and Top 10 are 43.5%, 55.1%, 43.9%, 68.8% and 78.5%, respectively, which are around 10% (Top 10) $\sim$ 26% (MAP) higher than the average of additional Java projects. K-S test (Table 10) shows that on the

extended Java dataset, all the p-values are greater than 5%, so we should reject the hypothesis that the two samples come from different distributions. On the extended Python dataset, all the p-values are less than 5%, thus the hypothesis that the two samples come from different distributions should be assumed to be true. According to the box plot in Fig. 6, we can see that on the extended Java dataset, the distribution of each metric is more concentrated, more similar, and the mean values are closer. On the extended Python dataset, the distribution of each metric is more dispersed, with wider ranges, but the mean is significantly higher than that of the other two Java datasets.

Based on the results and analysis, we can conclude that the performance of the fixed weight composer also holds for a larger dataset, and we gain confidence in its generalizability.

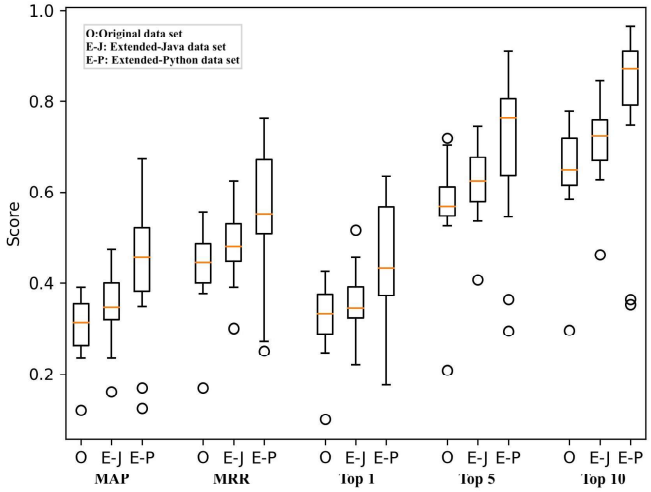


Fig. 6 Box plots of fixed weight on original and extended Java and Python datasets.

Answering RQ2: *The reported results of ABLoTS are not replicable, because of the incorrect use of the cut-off date in the BugCache component and the sub-optimal configuration of the composer. Consequently, we did not check the generalizability of ABLoTS, since applying an incorrect technique would provide little useful insight. However, with a fixed weight scheme, the results are generalizable on the extended dataset.*

4.3 RQ3. How do different composers perform when combining three components?

To assess the performance of different composers in combining three components, we employed both supervised and unsupervised methods, including DT, MLP, RF, LR, fixed weight, CombSUM, CombMNZ, CombANZ, CorrB, and Borda Count, as the

Table 9 Fixed weight composer performance on the extended datasets.

Extended-Java dataset					
PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
ARCHIVA	0.322	0.477	0.347	0.587	0.667
CASSANDRA	0.335	0.462	0.330	0.622	0.741
ERRAI	0.310	0.505	0.389	0.630	0.722
FLINK	0.416	0.560	0.456	0.670	0.752
GROOVY	0.388	0.458	0.331	0.618	0.726
HBASE	0.398	0.528	0.398	0.697	0.778
HIBERNATE	0.234	0.400	0.290	0.551	0.626
HIVE	0.357	0.483	0.343	0.647	0.746
JBOSS-T.-M.	0.370	0.536	0.403	0.701	0.791
KAFKA	0.474	0.623	0.516	0.742	0.844
LUCENE	0.321	0.466	0.336	0.624	0.710
MAVEN	0.337	0.416	0.296	0.546	0.671
RESTEASY	0.257	0.391	0.275	0.536	0.638
SPARK	0.406	0.496	0.379	0.606	0.712
SWITCHYARD	0.160	0.300	0.220	0.407	0.462
ZOOKEEPER	0.422	0.537	0.383	0.745	0.830
AVERAGE	**0.344**	**0.477**	**0.356**	**0.621**	**0.714**
Extended-Python dataset					
PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
CERTBOT	0.124	0.25	0.176	0.294	0.353
COMPOSE	0.349	0.465	0.29	0.677	0.806
DJANGO_R._F.	0.578	0.637	0.548	0.742	0.871
FLASK	0.674	0.762	0.636	0.909	0.909
KERAS	0.488	0.523	0.455	0.545	0.818
MITMPROXY	0.43	0.551	0.409	0.818	0.909
PIPENV	0.393	0.687	0.571	0.786	0.964
REQUESTS	0.502	0.551	0.4	0.8	0.933
SCIKIT-LEARN	0.422	0.533	0.41	0.668	0.747
SCRAPY	0.481	0.667	0.565	0.783	0.87
SPACY	0.169	0.271	0.182	0.364	0.364
TORNADO	0.606	0.714	0.625	0.875	0.875
Average	**0.435**	**0.551**	**0.439**	**0.688**	**0.785**

composer component. The average results of each composer across the three datasets are presented in Table 11. The box plots for different composers on the three datasets are depicted in Fig. 7, Fig. 8, and Fig. 9, respectively. From the table, we observe that LR, CombSUM, and fixed weight consistently achieve superior results compared to other composers. LR attains the best average outcomes on the original Java dataset and the extended Python dataset. Conversely, DT exhibits the poorest results across all three datasets, with RF following closely behind. Results are even worse than pure TraceScore. This indicates that the contributions of different component scores are linear, which LR, fixed weight, and CombSUM can model effectively due to their ability to handle linear relationships and lower risk of overfitting. In contrast, RF and DT, being tree-based structures, may struggle to fit the component scores effectively with their decision tree-based splits, which could compromise their performance.

To visualize the performance of various composers, we generated the heatmap shown in Fig. 10. This heatmap details the top-performing composer for each project

Table 10 K-S test result of Fixed Weight.

	Fixed Weight-Java		Fixed Weight-Python	
Metrics	K-S test	P value	K-S test	P value
MAP	0.313	0.452	0.750	0.001
MRR	0.295	0.512	0.568	0.031
Top 1	0.210	0.856	0.568	0.031
Top 5	0.443	0.115	0.583	0.017
Top 10	0.358	0.289	0.750	0.001

across different evaluation metrics. Analysis of the heatmap reveals that, across all projects and the five evaluation metrics, LR demonstrated the highest performance on 68 cases, closely followed by CombSUM with 64 instances. Interestingly, DT never attained the top-performing result. In summary, however, there is no composer that consistently performs best across all projects.

Based on the results above, the question that arises is: for real-world scenarios, which composer to select? The choice of a composer depends on the characteristics of the data, the nature of the retrieval methods, the relationship between the contributions of different components, and other desired aspects. Therefore, it is recommended to experiment with different composer methods (especially those good at handling linear relationships) and evaluate their performance to determine which one works best in a specific application context.

Across all composers, the average results on the extended Python dataset surpass those on the extended Java dataset, which, in turn, outperform the results on the original dataset. The MAP and MRR on the Python dataset reach as high as 44% and 56.2%, respectively. This result suggests that bug localization might be easier on the Python dataset than on the Java dataset. Moreover, we observed that Python projects have a smaller size, as measured by the number of source code files, in comparison to Java projects. This observation instigated an exploration into whether a correlation exists between bug localization accuracy and project size. Table 12 presents the Pearson correlation between project size and localization accuracy across all 39 projects. Given that all the p-values are greater than 0.05 (except for MAP of LR), we are unable to reject the null hypothesis. There is insufficient evidence to assert the presence of a significant effect or relationship between localization accuracy and project size.

Answering RQ3: *Among all the composers, LR, CombSUM, and fixed weight demonstrate favorable performance across the three datasets, while DT and RF significantly lag behind other composers. Hence, we highlight that the original fixed weights introduced in AmaLgam provide a simple, immediately applicable composition configuration for most settings. Nevertheless, it is advisable to explore various methods in order to select a composer based on specific situations.*

4.4 Discussion

Overall, as shown in Table 13, our experimental results suggest that TraceScore is **replicable under *relaxed cut-off date* constraint**, but, **non-replicable under**

Table 11 Average Results of Different Composers.

Original Java dataset					
	MAP	MRR	Top 1	Top 5	Top 10
LR	**0.302**	**0.435**	**0.321**	0.561	**0.650**
CombSUM	0.300	0.433	0.321	**0.566**	0.643
Fixed Weight	0.298	0.433	0.320	0.563	0.640
MLP	0.298	0.430	0.316	0.560	0.642
CorrB	0.289	0.418	0.303	0.550	0.634
CombMNZ	0.277	0.416	0.314	0.527	0.627
Borda Count	0.219	0.356	0.262	0.464	0.543
CombANZ	0.195	0.292	0.191	0.387	0.479
RF	0.179	0.279	0.161	0.394	0.526
DT	0.056	0.077	0.023	0.090	0.171
Extended-Java dataset					
	MAP	MRR	Top 1	Top 5	Top 10
CombSUM	**0.359**	**0.498**	**0.379**	**0.641**	**0.729**
Fixed Weight	0.344	0.477	0.356	0.621	0.714
LR	0.342	0.473	0.352	0.615	0.711
CombMNZ	0.330	0.473	0.353	0.609	0.700
CorrB	0.337	0.466	0.344	0.609	0.710
MLP	0.300	0.417	0.304	0.541	0.641
Borda Count	0.257	0.396	0.293	0.515	0.590
CombANZ	0.247	0.353	0.240	0.474	0.589
RF	0.203	0.295	0.170	0.428	0.564
DT	0.064	0.083	0.015	0.121	0.225
Extended-Python dataset					
	MAP	MRR	Top 1	Top 5	Top 10
LR	**0.440**	**0.562**	**0.449**	**0.693**	**0.794**
Fixed Weight	0.435	0.551	0.439	0.688	0.785
CorrB	0.430	0.538	0.414	0.682	0.787
CombSUM	0.426	0.535	0.429	0.646	0.756
MLP	0.414	0.519	0.394	0.672	0.776
CombMNZ	0.397	0.516	0.415	0.651	0.744
CombANZ	0.334	0.427	0.298	0.599	0.691
Borda Count	0.321	0.417	0.298	0.535	0.672
RF	0.233	0.332	0.196	0.485	0.635
DT	0.177	0.222	0.079	0.394	0.565

***strict cut-off date* constraint**, where the former can achieve better results. However, in actual applications, the choice between relaxed cut-off date and strict cut-off date is flexible, as commits available at the time when developers perform bug-fixing tasks will be considered for recommendation.

On the extended dataset, TraceScore also yields similar results compared with on the original dataset, which demonstrates that **TraceScore possesses good generalizability**. However, the results vary more (i.e., some projects exhibit much higher performance, other projects exhibit even lower performance), which means it is not possible to accurately predict the performance of TraceScore on a new project. Additional investigations are necessary to understand when TraceScore is expected to perform well and under which conditions TraceScore will not yield a lot of benefits.

ABLoTS, in contrast, **is not reproducible** for two main reasons: 1) the authors reused the wrong BugCache implementation from Wang et al. [14] (we confirmed the

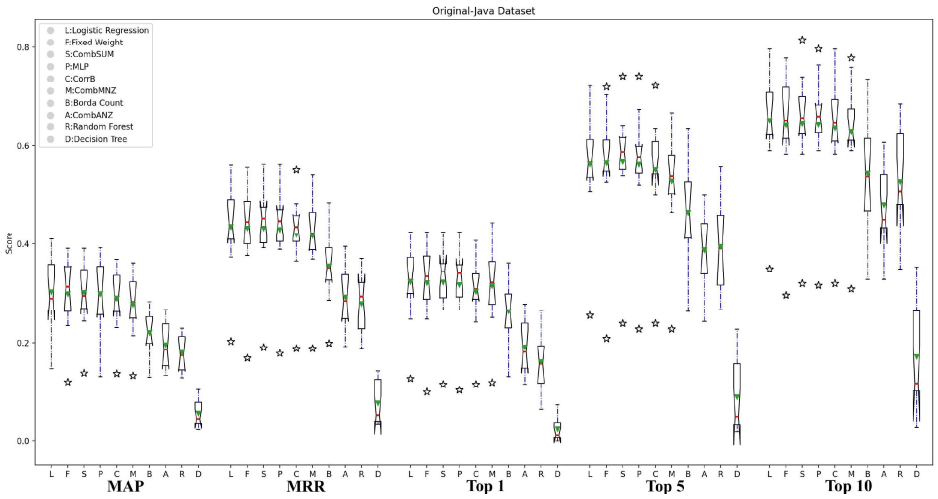


Fig. 7 Performance of Different Aggregation Methods on Original Java Dataset.

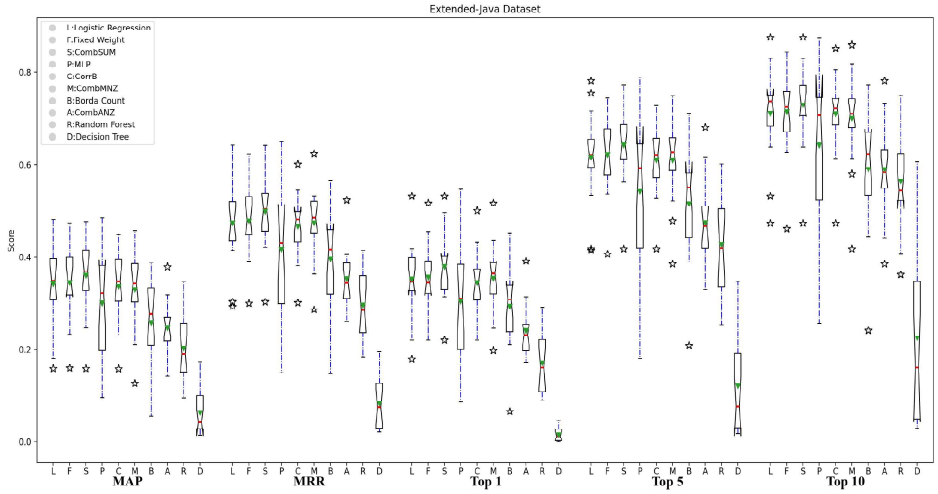


Fig. 8 Performance of Different Aggregation Methods on Extended Java Dataset.

incorrect use with Wang et al.), which results in the BugCache score greatly boosting the final result; 2) when we adopt the correct BugCache score, we could not duplicate the DT composer because of a lack of details in the original study. We are skeptical whether DT is the right choice for the composer, as also different sampling strategies and hyperparameter tuning yielded a performance worse than the static composer

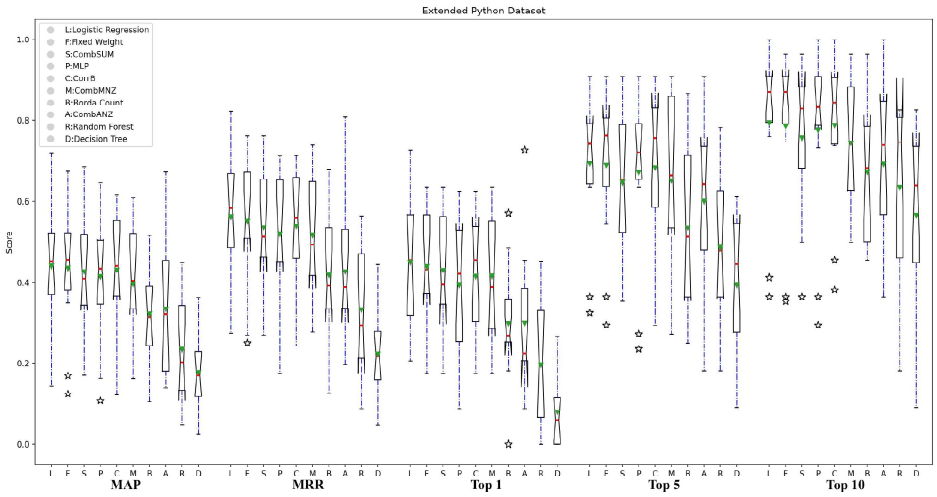


Fig. 9 Performance of Different Aggregation Methods on Extended Python Dataset.

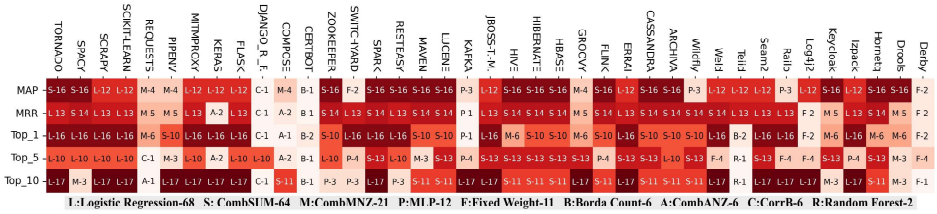


Fig. 10 Heatmap of best composer on each project. Each row in the heatmap corresponds to a specific evaluation metric, while each column represents an individual project. For instance, in the first row and first column, "S-16" signifies that the "CombSUM" composer achieved the best result in the MAP metric for the "TORNADO" project. "16" means that, across all 39 projects, CombSUM yielded the optimal results in MAP 16 times.

configuration. When we utilized this fixed weight composer proposed by AmaLgam we observed its performance to hold also for the extended dataset.

Both TraceScore and the fixed weight-based ABLoTS demonstrate superior results on the extended Python dataset compared to the other two Java datasets. What is particularly noteworthy is that, despite the absence of feature requests and traceability information in the Python dataset, TraceScore still achieves better localization results than on the other two Java datasets. A test examining the correlation between project size and bug localization performance indicated that such a correlation does not exist (refer to Table. 12 for the Pearson correlation coefficient of project size and bug localization performance). Hence we believe that further investigation into bug reports and bug locations is required to understand which factors, such as textual

Table 12 Pearson Correlation Coefficient of Project Size and Localization Accuracy.

Pearson	Fixed weight		LR		CombSUM	
Correlation	**Correlation**	**p-value**	**Correlation**	**p-value**	**Correlation**	**p-value**
MAP	-0.306	0.058	-0.364	0.023	-0.274	0.091
MRR	-0.182	0.268	-0.290	0.074	-0.100	0.546
Top 1	-0.190	0.246	-0.309	0.056	-0.104	0.530
Top 5	-0.158	0.337	-0.240	0.141	-0.049	0.767
Top 10	-0.174	0.289	-0.248	0.128	-0.093	0.572

alignment of terms used in issues and source code, or cohesion of terms, or diversity of terms help to explain bug localization performance.

We observed that combining all three scores can improve the TraceScore result by 50% $\sim$ 105% on both datasets, which suggests that a combination of different components is likely to outperform any single mechanism. To this end, the choice of composer is a crucial aspect. Our empirical evaluation of the performance of various composers unveiled that LR, CombSUM, and fixed weight consistently outperformed, while DT and RF exhibited subpar performance.

One additional take-away of our replication study is to pay attention to the challenge of properly evaluating a technique in the presence of temporal aspects, especially when third-party research outcomes (i.e., BugCache) behave differently than expected. The case of the 15-day interval of BugCache is especially tricky, as for other datasets where commits of a bug predominately happen more than 15 days before the bug's closing date, no such negative side effect would have been noticeable. In the case of ABLoTS sanity checks on the DT's feature importance values would have identified unexpected results (i.e., with BugCache rather than TraceScore dominating the classification result), subsequently triggering confirmation or revision of the composer mechanism.

Overall, the results of this replication study suggest that the state of the art in bug localization is not as useful as prior results have suggested and that further research is still needed to obtain results that are good enough to be useful in practice.

Table 13 Summary of Results.

		Replicable	Generalizable
TraceScore	Relaxed cut-off date	Yes	Yes
	Strict cut-off date	No	-
ABLoTS		No	-
Fixed Weight Composer		-	Yes

4.5 Implications to Future Replication

In this section, we summarize lessons learned through our replication specific to bug localization. The goal is to support researchers in more efficiently and rapidly replicating the approach for a comparative study or as a baseline for novel approaches.

- **Data Collection:** During the data collection, Rath et al. collected both bug reports and non-bug reports, traceability information between reports, commit logs, commit code change, and constructed links from issues to code change. For the ground truth construction, Rath et al. utilized the modified files and newly added files as the ground truth. However, in our opinion, which files are newly added cannot be predicted by definition. In contrast, removed files are predictable, and should be included in the ground truth. This slight difference in the construction of the ground truth would not change the approach, but could potentially affect the evaluation results.
- **Trace Graph Construction:** The construction of the trace graph requires previously fixed issues. When replicating, researchers need to keep in mind that bug fixing (or development efforts in the scope of an issue in general) are not atomic events but potentially long-lasting activities with days, sometimes even months, between issue creation date, commit dates, and issue closing dates. Hence, they need to be careful about the date for artifacts selection. When predicting, only information available at that time should be used.
- **BugCache Calculation:** For selecting the historical bug-fixing commits, apart from keywords-based selection, also a bug ID can identify bug-fixing commits. More important, as with the trace graph construction, any commit information taken for file candidate scoring must have been already available by the fictive recommendation time (e.g., bug creation date) or at the latest the bug's first partial fix implementation. As we have seen in the replication study, using the bugs' fixed date may lead to data leakage.
- **Choice of Composer:** Given a limited set of features (i.e., the three suspiciousness scores), a DT may not be the most suitable option. Researchers could initiate the exploration with LR, CombSUM, and the fixed weight composer. However, it is crucial to experiment with various methods to select a composer tailored to specific situations.

5 Threats to Validity

In this section, we discuss the main threats to the validity of our results, and how we mitigate them. We use the taxonomy of Wohlin et al. [60].

Construct Validity. One possible threat to construct validity is that there is no available open source implementation of ABLoTS approach, which means we had to re-implement it by ourselves. To alleviate this, we carefully read the original study, trying to reproduce it as close as possible. For the BLUiR component, we reused existing open source code from a published paper to reduce possible errors. As for BugCache component, we translate the original implementation from Java into Python with great care. We carefully examined the code and the output to avoid errors. Furthermore, we reached out to the authors of both ABLoTS and AmaLgam approaches to clarify experiment details, such as the selection of the cut-off date, aiming to ensure precise replication. Another potential construct validity concern arises from the dataset. The Python projects are sourced from the BuGL [33] dataset, a large-scale cross-language

dataset for bug localization. The Python dataset lacks feature requests and traceability information, potentially introducing bias to the performance evaluation of TraceScore.

Internal Validity. From a perspective of internal validity, potential errors can happen in the reproduction (e.g., settings and library usage), which is a common threat to replication studies. We tried out possible settings and compared the results with the original study. Another potential threat is that the open source projects in our dataset might have been changed by the day we collected from GitHub. To address this threat, we filter out projects that do not have complete commit information anymore.

External Validity. Regarding external validity, we experimented only on open source Java projects and Python projects. We encourage future studies to replicate this study with commercial projects to be able to obtain insights into ABLoTS' generalizability in industrial settings.

Conclusion Validity. Conclusion Validity could come from the interpretation of the results, which includes the evaluation metrics for evaluation and K-S test for comparison. To mitigate the threat, we adopted the same evaluation metrics adopted in the original paper. Then the two sample K-S test was utilized to compare the difference of experiment results, as it is sensitive to differences in both location and shape of the empirical cumulative distribution functions of the two samples.

6 Related Work

Several IRBF approaches have been proposed in the last years, which leverage information retrieval techniques to find buggy-prone snippets from all source code candidates. We describe below the approaches related to our work.

BugLocator calculates similarity between bug reports to recommend similar files to similar bug reports [16]. Sisman and Kak propose a source code version history-based fault localization approach, which utilizes the frequency of a file being buggy and its modifications to prioritize candidate source code files [61]. Wang et al. combine similar bug reports, code version history, and code structure to find the buggy files [14, 17]. Lucia et al. investigate five data fusion methods to improve spectrum-based fault localization techniques [58].

Niu et al. propose a refactoring-aware traceability model for constructing more accurate code history, which can boost the results of similar bug reports and code version history component [62]. Wen et al. use change logs and change hunks from commit message as alternative of segments of source code files to enable more accurate bug localization [63].

Recently, with the emergency of deep learning (DL), many IRBL approaches utilizing DL techniques have been proposed [64–66]. They compute the semantic similarity between source code and bug reports by converting them into deep representations. They achieve better results than methods relying solely on code structure. However, similar reports component and version history component continue to play indispensable roles. Many approaches still combine results from the similar reports component [67–74] and version history component [67–79] with DL-based semantic similarity to achieve improved bug localization effectiveness. For example, the MD-CNN approach [76] combines similar bug history and bug-fixing history with other

code structure similarities, employing convolutional neural networks to extract features for IRBL.

For comparison of state-of-the-art approaches, Garnier and Garcia evaluate the effectiveness of BLUiR [19], BLUiR+ [19], and AmaLgam [14] on 20 C# projects [80]. Lee et al. conducted a generalized and large-scale investigation into six IRBL techniques [24]. Akbar et al. divided IRBL tools into three generations and presented a comprehensive large-scale evaluation of all three generations of bug-localization tools with code libraries in multiple languages (including Java, C, C++ and Python) [18]. Li et al. re-implement six state-of-the-art bug localization approaches and report their effectiveness on 10 Huawei projects [25]. Lee et al. [24] and Li et al. [25] analyzed the same five state-of-the-art approaches and found lower average results than the original ABLoTS results (e.g., MAP less than 0.4, and MRR less than 0.53).

Despite these several empirical studies described above, none of them included ABLoTS, which strengthens the usefulness of our study. Moreover, to the best of our knowledge, there is no study that investigate the performance of different composers on combining the suspiciousness scores of the three components. Thus, our work address both limitation of related work, by replicating ABLoTS and exploring the use of different composers.

7 Conclusion

In this paper, we conduct a replication study of the ABLoTS approach for bug localization. We recreated the original approach, both on the original dataset and two extended datasets (one Java dataset and one Python dataset). Furthermore, we also conduct empirical analysis on the performance of various composers. We found that the core component of ABLoTS, i.e., TraceScore, is replicable and generalizable under a *relaxed cut-off* constraint, but irreplicable under a *strict cut-off* constraint. ABLoTS is neither replicable nor generalizable because of the adoption of an incorrect cut-off date in the BugCache subcomponent, leading to test data leaking into training data. Also, the chosen technique to combine multiple scores yielded poor results when applied to the correctly derived scores. Our study emphasizes the importance of choosing the proper cut-off dates in evaluating bug localization techniques. The comparison between different programming languages suggests that bug localization on Python projects might be more straightforward than on Java projects. Our empirical analysis reveals that LR, fixed weight, and CombSUM demonstrate superior performance in combining different components, while DT and RF exhibit poor performance.

As part of future work, we will start investigating alternative information sources and techniques to improve bug localization performance. We will also explore the effectiveness of large language models in bug localization, aiming to achieve higher accuracy in bug localization.

Data Availability. Artifacts of the experiment are available in an online replication package: https://github.com/feifeiniu-se/Replication2

Acknowledgments. This work is supported by Natural Science Foundation of Jiangsu Province, China (BK20201250), Cooperation Fund of Huawei-NJU Creative Laboratory for the Next Programming, and also supported in part by NSF Grant

2034508 (USA), by a Sam Taylor Fellowship Award, the Austrian Science Fund (FWF) grant P31989-N31 and P34805-N as well as the LIT Secure and Correct System Lab sponsored by the province of Upper Austria.

Compliance with Ethical Standards

Conflict of Interest The authors declared that they have no conflict of interest.

References

[1] Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C.: Bug characteristics in open source software. Empirical software engineering **19**(6), 1665–1705 (2014)

[2] Aslan, Ö., Samet, R.: Mitigating cyber security attacks by being aware of vulnerabilities and bugs. In: 2017 International Conference on Cyberworlds (cw), pp. 222–225 (2017). IEEE

[3] Piessens, F.: A taxonomy of causes of software vulnerabilities in internet software. In: Supplementary 13th International Symposium on Software Reliability Engineering, pp. 47–52 (2002). Citeseer

[4] Anvik, J., Hiew, L., Murphy, G.C.: Coping with an open bug repository. In: 2005 OOPSLA Workshop on Eclipse Technology eXchange, pp. 35–39 (2005)

[5] Kim, S., Zimmermann, T., Whitehead Jr, E.J., Zeller, A.: Predicting faults from cached history. In: 29th International Conference on Software Engineering (ICSE'07), pp. 489–498 (2007). IEEE

[6] Zhang, H.: An investigation of the relationships between lines of code and defects. In: 2009 IEEE International Conference on Software Maintenance, pp. 274–283 (2009). IEEE

[7] Anvik, J., Hiew, L., Murphy, G.C.: Who should fix this bug? In: 28th International Conference on Software Engineering, pp. 361–370 (2006)

[8] Bettenburg, N., Premraj, R., Zimmermann, T., Kim, S.: Duplicate bug reports considered harmful... really? In: 2008 IEEE International Conference on Software Maintenance, pp. 337–345 (2008). IEEE

[9] Ciborowska, A., Damevski, K.: Fast changeset-based bug localization with bert. In: 44th International Conference on Software Engineering, pp. 946–957 (2022)

[10] Huo, X., Thung, F., Li, M., Lo, D., Shi, S.-T.: Deep transfer bug localization. IEEE Transactions on software engineering **47**(7), 1368–1380 (2019)

[11] Li, G., Liu, H., Chen, X., Gunawi, H.S., Lu, S.: Dfix: automatically fixing timing bugs in distributed systems. In: 40th ACM SIGPLAN Conference on

Programming Language Design and Implementation, pp. 994–1009 (2019)

[12] Jeffrey, D., Feng, M., Gupta, N., Gupta, R.: Bugfix: A learning-based tool to assist developers in fixing bugs. In: 2009 IEEE 17th International Conference on Program Comprehension, pp. 70–79 (2009). IEEE

[13] Loyola, P., Gajananan, K., Satoh, F.: Bug localization by learning to rank and represent bug inducing changes. In: 27th ACM International Conference on Information and Knowledge Management, pp. 657–665 (2018)

[14] Wang, S., Lo, D.: Version history, similar report, and structure: Putting them together for improved bug localization. In: 22nd International Conference on Program Comprehension, pp. 53–63 (2014)

[15] Lukins, S.K., Kraft, N.A., Etzkorn, L.H.: Bug localization using latent dirichlet allocation. Information and Software Technology **52**(9), 972–990 (2010)

[16] Zhou, J., Zhang, H., Lo, D.: 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), pp. 14–24 (2012). IEEE

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

[18] Akbar, S.A., Kak, A.C.: A large-scale comparative evaluation of ir-based tools for bug localization. In: 17th International Conference on Mining Software Repositories, pp. 21–31 (2020)

[19] Saha, R.K., Lease, M., Khurshid, S., Perry, D.E.: Improving bug localization using structured information retrieval. In: 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 345–355 (2013). IEEE

[20] McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., Xie, Q.: Exemplar: A source code search engine for finding highly relevant applications. IEEE Transactions on Software Engineering **38**(5), 1069–1087 (2011)

[21] Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H.: Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 181–190 (2014). IEEE

[22] Youm, K.C., Ahn, J., Kim, J., Lee, E.: Bug localization based on code change histories and bug reports. In: 2015 Asia-Pacific Software Engineering Conference (APSEC), pp. 190–197 (2015). IEEE

[23] Rath, M., Lo, D., Mäder, P.: Analyzing requirements and traceability information to improve bug localization. In: 15th International Conference on Mining Software Repositories, pp. 442–453 (2018)

[24] Lee, J., Kim, D., Bissyandé, T.F., Jung, W., Traon, Y.L.: Bench4bl: Reproducibility study of the performance of ir-based bug localization. In: 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2018, pp. 1–12 (2018). https://doi.org/10.1145/3213846.3213856

[25] Li, W., Li, Q., Ming, Y., Dai, W., Ying, S., Yuan, M.: An empirical study of the effectiveness of ir-based bug localization for large-scale industrial projects. Empirical Software Engineering **27**(2), 1–31 (2022)

[26] Shepperd, M., Ajienka, N., Counsell, S.: The role and value of replication in empirical software engineering results. Information and Software Technology **99**, 120–132 (2018)

[27] Carver, J.C.: Towards reporting guidelines for experimental replications: A proposal. In: 1st International Workshop on Replication in Empirical Software Engineering, vol. 1, pp. 1–4 (2010)

[28] Haben, G., Habchi, S., Papadakis, M., Cordy, M., Le Traon, Y.: A replication study on the usability of code vocabulary in predicting flaky tests. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 219–229 (2021). IEEE

[29] Da Silva, F.Q., Suassuna, M., França, A.C.C., Grubb, A.M., Gouveia, T.B., Monteiro, C.V., Santos, I.E.: Replication of empirical studies in software engineering research: a systematic mapping study. Empirical Software Engineering **19**(3), 501–557 (2014)

[30] Niu, F., Mayr-Dorn, C., Assunção, W.K., Huang, L., Ge, J., Luo, B., Egyed, A.: The ablots approach for bug localization: is it replicable and generalizable? In: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), pp. 576–587 (2023). IEEE

[31] Gómez, O.S., Juristo, N., Vegas, S.: Understanding replication of experiments in software engineering: A classification. Information and Software Technology **56**(8), 1033–1048 (2014)

[32] Rath, M., Mäder, P.: The seoss 33 dataset—requirements, bug reports, code history, and trace links for entire projects. Data in brief **25**, 104005 (2019)

[33] Muvva, S., Rao, A.E., Chimalakonda, S.: Bugl–a cross-language dataset for bug localization. arXiv preprint arXiv:2004.08846 (2020)

[34] Rahman, F., Posnett, D., Hindle, A., Barr, E., Devanbu, P.: Bugcache for inspections: hit or miss? In: 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 322–331 (2011)

[35] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. ACM SIGKDD explorations newsletter **11**(1), 10–18 (2009)

[36] Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge university press, ??? (2008)

[37] Schütze, H., Manning, C.D., Raghavan, P.: Introduction to Information Retrieval vol. 39. Cambridge University Press Cambridge, ??? (2008)

[38] Voorhees, E.M., *et al.*: The trec-8 question answering track report. In: Trec, vol. 99, pp. 77–82 (1999)

[39] Rath, M., Lo, D., Mäder, P.: Replication Data for: Analyzing Requirements and Traceability Information to Improve Bug Localization (2018)

[40] JIRA: Jira Issue Tracking Software. https://www.jira.com (2018)

[41] SCM, G.: Git SCM. https://www.git-scm.com (2018)

[42] Bachmann, A., Bernstein, A.: Software process data quality and characteristics: a historical view on open and closed source projects. In: Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, pp. 119–128 (2009)

[43] Ye, X., Bunescu, R., Liu, C.: Learning to rank relevant files for bug reports using domain knowledge. In: 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 689–699 (2014)

[44] B. Le, T.-D., Lo, D., Le Goues, C., Grunske, L.: A learning-to-rank based fault localization approach using likely invariants. In: 25th International Symposium on Software Testing and Analysis, pp. 177–188 (2016)

[45] Benton, S., Ghanbari, A., Zhang, L.: Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 47–50 (2019). IEEE

[46] Baeza-Yates, R., Ribeiro-Neto, B., *et al.*: Modern Information Retrieval vol. 463. ACM press New York, ??? (1999)

[47] NLTK: NLTK library. http://www.nltk.org (2022)

[48] Jones, K.S.: A statistical interpretation of term specificity and its application in

retrieval. Journal of documentation (1972)

[49] scikit-learn: scikit-learn. https://scikit-learn.org/stable/ (2022)

[50] Lewis, C., Lin, Z., Sadowski, C., Zhu, X., Ou, R., Whitehead, E.J.: Does bug prediction support human developers? findings from a google case study. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 372–381 (2013). IEEE

[51] Chris Lewis, R.O.: Bug Prediction at Google. [EB/OL]. http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html Accessed 12, 2011

[52] Strohman, T., Metzler, D., Turtle, H., Croft, W.B.: Indri: A language model-based search engine for complex queries. In: International Conference on Intelligent Analysis, vol. 2, pp. 2–6 (2005). Citeseer

[53] Imblearn: Imblearn. https://imbalanced-learn.org/stable/ (2022)

[54] Fox, E.A., Koushik, M.P., Shaw, J., Modlin, R., Rao, D., *et al.*: Combining evidence from multiple searches. In: The First Text Retrieval Conference (TREC-1), pp. 319–328 (1993)

[55] Fox, E., Shaw, J.: Combination of multiple searches. NIST special publication SP, 243–243 (1994)

[56] Wu, S.: Data Fusion in Information Retrieval. Adaptation, Learning, and Optimization, vol. 13. Springer, ??? (2012). https://doi.org/10.1007/978-3-642-28866-1 . https://doi.org/10.1007/978-3-642-28866-1

[57] Aslam, J.A., Montague, M.H.: Models for metasearch. In: Croft, W.B., Harper, D.J., Kraft, D.H., Zobel, J. (eds.) SIGIR 2001: 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, September 9-13, 2001, New Orleans, Louisiana, USA, pp. 275–284. ACM, ??? (2001). https://doi.org/10.1145/383952.384007 . https://doi.org/10.1145/383952.384007

[58] Lucia, Lo, D., Xia, X.: Fusion fault localizers. In: 29th ACM/IEEE International Conference on Automated Software Engineering, pp. 127–138 (2014)

[59] Massey Jr, F.J.: The kolmogorov-smirnov test for goodness of fit. Journal of the American statistical Association **46**(253), 68–78 (1951)

[60] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, USA (2000). https://doi.org/10.1007/978-1-4615-4625-2

[61] Sisman, B., Kak, A.C.: Incorporating version histories in information retrieval based bug localization. In: 2012 9th IEEE Working Conference on Mining

Software Repositories (MSR), pp. 50–59 (2012). IEEE

[62] Niu, F., Assunçao, W.K.G., Huang, L., Mayr-Dorn, C., Ge, J., Luo, B., Egyed, A.: Rat: A refactoring-aware traceability model for bug localization. In: 2023 45th International Conference on Software Engineering (ICSE) (2023). IEEE. accepted

[63] Wen, M., Wu, R., Cheung, S.-C.: Locus: Locating bugs from software changes. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 262–273 (2016). IEEE

[64] Ciborowska, A., Damevski, K.: Fast changeset-based bug localization with bert. In: Proceedings of the 44th International Conference on Software Engineering, pp. 946–957 (2022)

[65] Han, J., Huang, C., Sun, S., Liu, Z., Liu, J.: bjxnet: an improved bug localization model based on code property graph and attention mechanism. Automated Software Engineering **30**(1), 12 (2023)

[66] Yong, J., Zhu, Z., Li, Y.: Decomposing source codes by program slicing for bug localization. In: 2023 International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2023). IEEE

[67] Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 218–229 (2017). IEEE

[68] Xiao, Y., Keung, J., Bennin, K.E., Mi, Q.: Improving bug localization with word embedding and enhanced convolutional neural networks. Information and Software Technology **105**, 17–29 (2019)

[69] Sangle, S., Muvva, S., Chimalakonda, S., Ponnalagu, K., Venkoparao, V.G.: Drast–a deep learning and ast based approach for bug localization. arXiv preprint arXiv:2011.03449 (2020)

[70] Cao, J., Yang, S., Jiang, W., Zeng, H., Shen, B., Zhong, H.: Bugpecker: Locating faulty methods with deep learning on revision graphs. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 1214–1218 (2020)

[71] Yang, S., Cao, J., Zeng, H., Shen, B., Zhong, H.: Locating faulty methods with a mixed rnn and attention model. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), pp. 207–218 (2021). IEEE

[72] Qi, B., Sun, H., Yuan, W., Zhang, H., Meng, X.: Dreamloc: A deep relevance matching-based framework for bug localization. IEEE Transactions on Reliability **71**(1), 235–249 (2021)

[73] Shi, X., Ju, X., Chen, X., Lu, G., Xu, M.: Semirfl: Boosting fault localization via combining semantic information and information retrieval. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), pp. 324–332 (2022). IEEE

[74] Xu, G., Wang, X., Wei, D., Shao, Y., Chen, B.: Bug localization with features crossing and structured semantic information matching. International Journal of Software Engineering and Knowledge Engineering (2023)

[75] Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: Combining deep learning with information retrieval to localize buggy files for bug reports (n). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 476–481 (2015). IEEE

[76] Wang, B., Xu, L., Yan, M., Liu, C., Liu, L.: Multi-dimension convolutional neural network for bug localization. IEEE Transactions on Services Computing **15**(3), 1649–1663 (2020)

[77] Anh, B.T.M., Luyen, N.V.: An imbalanced deep learning model for bug localization. In: 2021 28th Asia-Pacific Software Engineering Conference Workshops (APSEC Workshops), pp. 32–40 (2021). IEEE

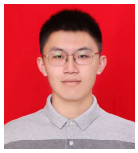
[78] Xiao, X., Xiao, R., Li, Q., Lv, J., Cui, S., Liu, Q.: Bugradar: Bug localization by knowledge graph link prediction. Information and Software Technology, 107274 (2023)

[79] Al-Aidaroos, A.S., Bamzahem, S.M.: The impact of glove and word2vec word-embedding technologies on bug localization with convolutional neural network. International Journal of Science and Engineering Applications, 108–111 (2023)

[80] Garnier, M., Garcia, A.: On the evaluation of structured information retrieval-based bug localization on 20 c# projects. In: XXX Brazilian Symposium on Software Engineering, pp. 123–132 (2016)

Author Biography

Feifei Niu is a Research Fellow at the University of Ottawa. She received her Doctorate from Nanjing University. Her research interests include software quality assurance, software testing, requirements engineering.



Enshuo Zhang is a master student at Nanjing University. His research interests include software engineering, natural language processing. He can be contacted at 2575357413@qq.

Christoph Mayr-Dorn is a senior researcher at the Institute for Software Systems Engineering at the Johannes Kepler University Linz, Austria. He holds a Ph.D. in Computer Science from the Technical University Vienna. His current research interests include software process monitoring and mining, change impact assessment, and software engineering of cyber–physical production systems.

Wesley Klewerton Guez Assunção is an Associate Professor with the Department of Computer Science at North Carolina State University. Wesley was a University Assistant in the Institute of Software Systems Engineering (ISSE) at Johannes Kepler University Linz (JKU), Austria (2021–2023); a Postdoctoral Researcher at Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil (2019–2023); and an Associate Professor at Federal University of Technology - Paraná, Brazil (2013 to 2020). He obtained his M.Sc. and Ph.D. in Computer Science from Federal University of Paraná (UFPR) also in Brazil. Further information: https://wesleyklewerton.github.io/.

Jidong Ge is an Associate Professor at Software Institute, Nanjing University. He received his PhD degree in Computer Science from Nanjing University in 2007. His current research interests include workflow modeling, process mining, cloud computing, workflow scheduling, software engineering. His research results have been published in more than 90papers in international journals and conference proceedings including IEEE TPDS, IEEE TSC, JASE, COMNET, JPDC, FGCS, JSS, Inf. Sci., JNCA, JSEP, ESA, ICSE, IWQoS etc

Bin Luo is a full Professor at the Software Institute, Nanjing University. His main research interests include cloud computing, computer network, workflow scheduling, and software engineering. His research results have been published in more than 50 papers in international journals and conference proceedings including IEEE TSC, ACM TIST, JSS, FGCS, Inf Sci, ESA, ICTAI, etc. He is leading the institute of applied software engineering at Nanjing University.

Alexander Egyed is a Full Professor and Chair for Software-Intensive Systems at the Johannes Kepler University, Austria (JKU). He received a Doctorate degree from the University of Southern California, USA in 2000 and then worked for industry for many years before joining the University College London, UK in 2007 and JKU in 2008. He is most recognized for his work on software and systems design — particularly on variability, consistency, and traceability.