Learning without Forgetting: Towards Continual learning of Fault Localization Models in Industrial Software Systems

Chun Li
Nanjing University
State Key Laboratory for Novel Software Technology
Nanjing, China
chunli@smail.nju.edu.cn

Hui Li
Samsung Electronics (China) R&D Centre
Nanjing, China
hui.li@samsung.com

Zhong Li
Nanjing University
State Key Laboratory for Novel Software Technology
Nanjing, China
lizhong@nju.edu.cn

Minxue Pan*
Nanjing University
State Key Laboratory for Novel Software Technology
Nanjing, China
mxp@nju.edu.cn

Xuandong Li
Nanjing University
State Key Laboratory for Novel Software Technology
Nanjing, China
lxd@nju.edu.cn

Abstract

Learning-based fault localization has achieved promising results. However, as software and tests are constantly evolving, models trained on old data become ineffective on new data. Particularly, in the context of system testing for large-scale software, each iteration generates a large volume of new data. This makes retraining the model from scratch incur an unacceptable time overhead, while merely fine-tuning on new data leads to catastrophic forgetting. Continual learning offers an effective method for models to avoid catastrophic forgetting during this iterative process. However, existing continual learning methods are not specifically designed for fault localization or for large-scale software system testing scenarios, which leads to their direct application yielding sub-optimal effectiveness. In response, we propose Ciallo, a novel continual learning framework specifically designed for large-scale software fault localization. Ciallo first extracts fine-grained program semantics from logs, then utilizes fault characteristics to enhance the weights of certain semantics. Finally, Ciallo uses an unsupervised algorithm to obtain corresponding embeddings and selects representative exemplars based on clustering. Subsequently, Ciallo mixes the representative exemplars with new samples for training and adjusts the loss weight according to the model's degree of mastery over the sample. This allows the model to focus more on samples that are not yet well-mastered during the training process, thereby enabling it to learn new faults while mitigating the forgetting of old ones. In extensive evaluations against 6 continual learning baselines, Ciallo demonstrates superior performance, improving overall effectiveness by 17.30% to 45.23%.

*Corresponding author.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

ACM Reference Format:
Chun Li, Hui Li, Zhong Li, Minxue Pan, and Xuandong Li. 2026. Learning without Forgetting: Towards Continual learning of Fault Localization Models in Industrial Software Systems. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3787809

1 Introduction

Context. Fault localization is the first step in debugging, and manual fault localization is typically time-consuming, costly, and labor-intensive [2]. This issue is even more pronounced in industrial settings, particularly during the system testing of large-scale software. In this scenario, engineers often perform fault localization by inspecting logs, which can require them to review tens of thousands of log lines [23, 47]. This significantly increases the challenges of debugging and creates a substantial industry demand for automated fault localization models. Currently, learning-based automated fault localization techniques have achieved state-of-the-art progress in log-based industrial software fault localization [23]. However, existing work primarily focuses on how to train the initial fault localization model, with little attention paid to the model iterations.

In real-world scenarios, as software and tests continuously evolve, new faults emerge [18]. In this context, the data distribution shifts, causing the effectiveness of models trained on old data to degrade when localizing faults in new data. This problem is particularly severe in the system testing of large-scale software. As shown in Figure 1, the performance of the no-update model degrades over the course of the iterations, eventually becoming unusable. In industrial settings, each iteration generates a massive amount of data [23]. Retraining the model from scratch after every iteration, therefore,

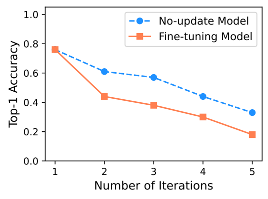


Figure 1: Top-1 accuracy of the model [23] under no-update across all test sets, and under fine-tuning on the first test set.

causes unacceptable time overhead, with the average time cost being four times that of training on a single dataset. Another straightforward solution is to fine-tune the model on new data; however, previous research—as well as our experiments—has shown that direct fine-tuning leads to catastrophic forgetting [12, 46]. As shown in Figure 1, fine-tuning leads to obvious performance degradation on the first dataset. In this scenario, it is important for the model to remember faults that have occurred in the past, as these errors may recur in the future.

Limitations. Continual learning (also known as lifelong learning) [46] techniques are typically used to solve the aforementioned problem of enabling a model to learn new knowledge while avoiding catastrophic forgetting. Currently, there are two mainstream categories of continual learning techniques. The first is replay-based methods [31, 38, 45], which add a subset of old data to new data to mitigate forgetting. The second category involves adding regularization to prevent the model's parameters from changing significantly, thereby combating forgetting [20, 28]. In software engineering, research has been conducted on continual learning, including code models [12] and test case selection [3, 9, 13, 29]. However, these methods were not designed with the characteristics of the fault localization scenario in mind, leading to significant limitations when adapting them to a log-based large-scale software fault localization scenario. These limitations primarily stem from the following aspects.

First, when selecting representative exemplars, the feature extraction component of existing methods is challenging to adapt to log-based fault localization. This hinders efficient feature extraction, causing the replay of these representative exemplars to yield only sub-optimal effectiveness. For example, REPEAT [12] uses TF-IDF to extract sample features. However, term frequency cannot effectively represent the program semantics within logs, as this method struggles to capture key features required for log-based fault localization, such as the relationships between different program entities or the execution order of methods [23]. Test case selection has also been used to extract features and select representative samples for replay in continual learning [9, 29, 52]. However, such methods primarily select samples based on the model's output probabilities (i.e., uncertainty), which may lead to the selection of samples with similar program semantics or faults [29]. This redundancy reduces the diversity of the representative exemplars, causing the replay process to achieve sub-optimal effectiveness given the limited size of the replay dataset [12, 29, 46]. *Second*, current methods primarily adopt a simple replay strategy, which involves directly training the model on a mix of representative exemplars and new data. This strategy, however, overlooks the unmastered faults within the data. In the context of large-scale software system testing, faults that have occurred previously may also reappear in new data. This can cause the model to perform well on some portions of the new data and poorly on others. We need to enable the model to focus more on those poorly performing (i.e., not yet mastered) samples during the continual learning process. More generally, both faults that may have been catastrophic forgetting and new faults constitute knowledge that the model has not yet mastered.

Proposal. To address this problem, we propose CIALLO (**C**ont**I**nual le**A**rning of fau**L**t **L**ocalizati**O**n), a novel continual learning framework specifically designed for log-based fault localization in large-scale software and system testing. The key insights underlying CIALLO are two-fold. *First*, feature extraction tailored to log-based fault localization can effectively enhance the selection of representative exemplars. To this end, CIALLO employs more fine-grained program semantic extraction from logs and utilizes fault characteristics to enhance certain program semantics during feature extraction. Specifically, CIALLO extracts coverage information and method call information from logs. It then increases the weights of different method calls based on their frequency and whether they involve a faulty entity. Finally, we obtain embeddings for the logs using the extracted program semantics and then select representative exemplars via clustering. *Second*, focusing more on yet-to-be-mastered samples during the training process can enable the model to better learn new knowledge and mitigate catastrophic forgetting. This is because both potentially forgotten faults and new faults are cases that it has not yet mastered well. To achieve this, we adaptively calculate corresponding weights during model training based on the model's confidence (i.e., the suspiciousness score). When the confidence is high, indicating the model has mastered the sample well, its weight is reduced; otherwise, the weight is increased. In this way, the model focuses more on learning from the samples it has mastered relatively poorly, thereby achieving better continual learning performance.

Evaluation. Our evaluation of CIALLO's performance in a large-scale software and system testing involved over three thousand logs and eight iterations from system tests of eleven software projects written in C/C++, each exceeding one million lines of code, supplied by a global corporation. We benchmarked CIALLO against six representative continual learning baselines, demonstrating its exceptional efficacy by significantly outperforming all compared methods. Specifically, CIALLO achieves an improvement of 17.30% to 45.23% in overall effectiveness, 14.89% to 63.63% in learning new knowledge, and 55.55% to 76.47% in mitigating forgetting, over the studied baselines.

Contributions. The main contributions of this paper are as follows:

- We propose CIALLO, a novel continual learning framework specifically designed to improve the continual learning performance of fault localization models in the context of large-scale software and system testing.
- We develop a representative exemplars selection method that leverages more fine-grained program semantics and fault characteristics, and an adaptive weight loss function that enables the model to better learn new faults while retaining knowledge of old ones.
- We conducted extensive evaluations of CIALLO within a continual learning setting on a large-scale dataset, encompassing

over three thousand logs and eight iterations, and the results demonstrate that Ciallo effectively improves the continual learning capability of fault localization models in the large-scale software and system testing scenario.

Data availability statement. Code and configuration of Ciallo are publicly available at https://github.com/pppppkun/Ciallo.

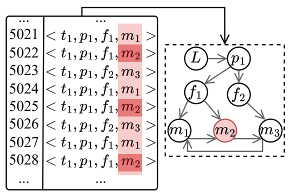
2 Methodology

In this section, we first introduce the background of industrial software fault localization and continual learning. Then, we define our problem and discuss the key observations behind this work to motivate the idea of Ciallo.

2.1 Background

Industrial Software Fault Localizations. The industrial software we consider primarily operates continuously in real-world production environments, and as a result, it constantly generates logs and requires ongoing maintenance based on that log data. The software itself is also typically complex and large-scale compared to toy examples. After the system testing of large-scale industrial software, engineers primarily rely on logs to localize faults [47]. Logs chronologically record the software's behavior during the testing process. Each log entry contains information about the currently executing method, as well as the file, package, and thread associated with that method [23]. Currently, log-based fault localization efforts focus on parsing program semantics from logs into graphs and training localization models using contrastive learning [23]. Figure 2 shows a real log obtained from our industrial partner, which has been simplified and anonymized, and the corresponding graph representation. t, p, f and m denote the thread, package, file, and method. m_2 is the faulty method. As software continuously iterates, the corresponding tests and faults also evolve, necessitating that the models be updated as well, as discussed in Section 1. This motivates us to investigate the problem of continual learning for automated fault localization models in industrial software systems.

Continual Learning. To adapt to dynamic real-world environments, data-driven models must be continuously updated for new scenarios [46]. Continual learning enables models to incrementally learn new knowledge from newly collected data while effectively preventing the catastrophic forgetting of old knowledge, thereby adapting to data distribution shifts. The two mainstream categories in continual learning are replay-based and regularization-based methods. Replay-based methods [38, 45] achieve continual learning by selecting representative exemplars from the old data and training on a mixture of these exemplars and new data to mitigate forgetting. Regularization-based methods [20, 28], on the other hand, introduce a regularization term to the loss function. This term penalizes significant changes to network parameters that are crucial for previous tasks, thereby mitigating forgetting. To the best of our knowledge, there is currently no work on applying continual learning to fault localization. In the domain of code models, REPEAT [12] combines both replay and regularization for continual learning. Test case selection [9, 52] can be applied to replay-based methods to select the representative exemplars from old data for replay, thereby enabling continual learning.



```
...   ...
5021  < t1, p1, f1, m1 >
5022  < t1, p1, f1, m2 >
5023  < t1, p1, f2, m3 >
5024  < t1, p1, f1, m1 >
5025  < t1, p1, f1, m2 >
5026  < t1, p1, f2, m3 >
5027  < t1, p1, f1, m1 >
5028  < t1, p1, f1, m2 >
...   ...
```

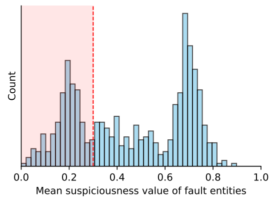


Figure 2: A simplified log and its graph representation.

Figure 3: The distribution of mean suspiciousness value of fault entities of each sample.

2.2 Problem Statement

Following prior work on fault localization in industrial software [23], we formally define a log as $L = (l_1, l_2, \ldots, l_n)$, where $(l_1, l_2, \ldots, l_n)$ denotes the chronologically ordered lines of the log L. Each line l_i, in general, comprises the running thread number, the package, file, and method executed. That is, $l_i = (t, p, f, m)$, where t, p, f, m represent the thread, package, file, and method, respectively. In the continual learning scenario, the model must learn continuously from a series of datasets, each containing logs and their corresponding fault entities. We assume that the software undergoes N testing iterations, and a dataset is collected after each iteration. Let D_i be the dataset collected after the i-th iteration, such that $D_i = \{T_i, V_i\}$, where T_i and V_i are the training and test sets, respectively. Any T or V is represented as a set of pairs $\{L, Y\}$, where L is a log and Y is the set of fault entities (superclass of package, file, and method). At the i-th time step, the model from step $i{-}1$ is updated using the cumulative training data $T_{1:i}$, and is expected to perform well on all test sets up to this point, $V_{1:i}$. The main problem tackled in this paper is *how to effectively and efficiently conduct continual learning of fault localization models in industrial software systems.*

We assume that the model can access all preceding training sets at the i-th time step rather than just a subset of them. This differs from typical continual learning assumptions, where prior work often presumes that the model cannot directly access past datasets due to storage constraints [12, 46]. However, in the development process, all tests and their corresponding faults are saved for subsequent review and tracking. This practice is reasonable and common in real-world scenarios. For instance, the Apache Software Foundation's issue tracking system[1] records nearly all faults and defects that arise in the software it develops, enabling developers to more easily track and query existing issues. Having access to the complete data allows us to achieve better performance.

2.3 Key Observations

Based on the characteristics of software testing and fault localization, we observe that there is room to improve the model's effectiveness in a continual learning scenario in the following aspects:

Observation I: Better feature extraction contributes to the advancement of representative exemplars selection. Selecting representative exemplars from old data for replay is an effective method to mitigate catastrophic forgetting in continual learning [12, 45, 46]. Through study cases on the logs, we found that effective feature extraction is crucial for determining which of

[1]https://issues.apache.org/jira

these samples are the most representative. In the context of fault localization, selecting representative exemplars is, to some extent, equivalent to selecting representative faults. Therefore, when feature extraction is performed, we must not only better represent the program semantics within the logs but also emphasize the features that are relevant to the fault. As shown in Figure 2, the calls to methods m_1, m_2 and m_3 form a loop related to the faulty method m_2. If the feature extraction cannot capture the number of method calls and method calls related to the faulty method, it becomes impossible to distinguish between different fault scenarios that are related to the number of loop iterations or whether to call the faulty method, leading to sub-optimal exemplar selection effectiveness. In such a case, the prior works [12] that used features such as term frequency statistics or the probability outputted by the model [3, 9, 13, 29] struggle to effectively extract the feature and program semantics. Furthermore, the feature extraction in prior log-based fault localization work [23, 32] has been relatively coarse, which only considers coverage information and execution order, failing to consider fault-related information and the fine-grained semantics. As a result, the feature extraction methods utilized in existing methods lead to sub-optimal representative exemplar selection. Thus, for more effective representative exemplars selection and replay, it is desirable to design an improved feature extraction that better represents program semantics and fault-related features.
Observation II: Enhanced learning on unmastered faults helps to mitigate catastrophic forgetting and acquire new knowledge. By investigating the distribution of the average suspiciousness score of the fault entities for each sample, we found that both entirely new faults and those forgotten by the model belong to the unmastered samples. Figure 3 plots the number of samples at each average suspiciousness score. We observe that the red-highlighted region contains approximately 85% of all entirely new and forgotten faults. The average suspiciousness scores for the fault entities in these samples are exceptionally low, indicating that both types of faults are almost entirely comprised of samples that the model has not yet mastered. For faults that the model has already mastered, it can exhibit strong performance on the corresponding data. For unmastered faults, however, its performance is relatively poor, and it must intensify its learning on these samples to achieve better performance. Therefore, during the training process, the model should continuously focus more on learning the faults it has not yet mastered, whether they originate from new or old data. This strategy serves to both mitigate forgetting on previously seen faults and facilitate the mastery of new faults contained in the new data, thereby achieving better fault localization performance in a continual learning setting.

3 Design

3.1 Our Approach

Based on the aforementioned observations, we propose a novel continual learning framework named Ciallo, designed for fault localization models in industrial software systems. Specifically, the design idea of Ciallo is twofold:
Enhancing feature extraction and representative sample selection through fine-grained extraction of program semantics and fault-related features. To better select representative exemplars for replay, we propose to refine the program semantic extraction process and leverage information related to faulty program entities to enhance feature extraction. The intuition here is that a more refined extraction of program semantics, combined with an emphasis on fault-related features, leads to a better vector representation of both the log and its associated fault. Based on this, given a log from a failed test, we first extract its basic program semantics. This includes the program entities (e.g., packages, files, methods) covered by the test and the intra-thread method execution sequence. As demonstrated in prior work [23], these features effectively reflect the program semantics in logs and are beneficial for fault localization. However, relying solely on these features is relatively coarse, as this approach overlooks both the frequency of method executions and whether the call sequence involves a faulty entity. Utilizing the method call frequency provides a more fine-grained characterization of the differences between various method calls and the program's execution state. Simultaneously, considering whether a faulty entity is involved effectively enriches different method calls with fault-related information, thereby emphasizing the fault features. Therefore, we further extract features from the frequency of method calls and whether the call sequence involves the faulty entity. This process serves to both refine the program semantics and emphasize crucial fault information. We will present details about how Ciallo extracts the feature from the log and selects representative exemplars in Section 3.3.
Adaptive adjusting the loss weights of individual samples during training based on the confidence calculated by the model. As stated in Observation II, to improve the model's overall continual learning performance, we want it to focus more on faults that have not been fully mastered during training. To achieve this, we propose adaptively adjusting loss weights during the training process based on the model's output probabilities. The insight behind this is that the model's output probability reflects its confidence in classifying a program entity as faulty; high confidence suggests the fault is well-mastered, and low confidence suggests the fault is poorly mastered [30]. Taking inspiration from this, during training, we calculate the model's confidence in predicting the ground-truth faulty entity and then use this confidence to adaptively adjust the weight of the corresponding sample's loss value. Through this mechanism, the model is prompted to continuously review old faults where forgetting may have occurred and to learn new faults, as these samples will exhibit relatively low confidence. This leads to better continual learning performance. To better mitigate catastrophic forgetting on the non-replayed samples and prevent overfitting to the representative exemplars, we also incorporate regularization to penalize excessive modifications to the model's parameters. Further details on the implementation of adaptive weighted loss and regularization will be discussed in Section 3.4.

3.2 Overview

In this section, we first provide an overview of the Ciallo workflow. Then, we elaborate on the technical details of each stage in Ciallo. Figure 4 presents the workflow of Ciallo. Our approach consists of two stages: *training exemplars generation* and *model iteration*.

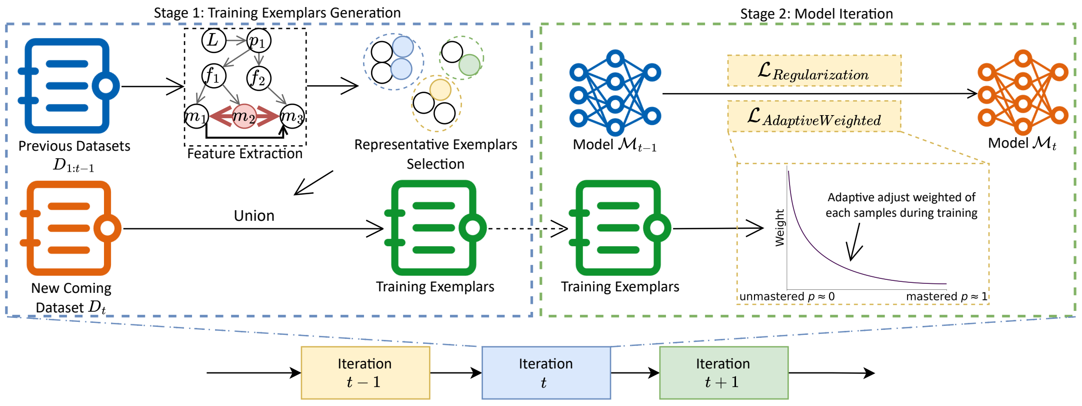

Stage 1: Training Exemplars Generation
Stage 2: Model Iteration
Previous Datasets $D_{1:t-1}$
Feature Extraction
Representative Exemplars Selection
New Coming Dataset D_t
Union
Training Exemplars
Model $\mathcal{M}_{t-1}$
Model $\mathcal{M}_t$
$\mathcal{L}_{Regularization}$
$\mathcal{L}_{AdaptiveWeighted}$
Adaptive adjust weighted of each samples during training
Weight
unmastered $p \approx 0$
mastered $p \approx 1$
Training Exemplars
Iteration $t-1$
Iteration t
Iteration $t+1$

Figure 4: The workflow of Ciallo.

The two stages work as follows. Suppose we are at the t-th iteration, with access to the past datasets $D_{1:t-1}$, the newly collected dataset D_t, and the model $\mathcal{M}_{t-1}$ from the previous iteration. In the *training exemplars generation* stage, we extract features from the $D_{1:t-1}$ and then select representative exemplars. Specifically, in the feature extraction, we convert logs into graph-structured data, treating program entities as nodes and using coverage information and the intra-thread method execution sequence as edges. Then, we increase the weight of edges by the method call frequency and whether they connect to a faulty node to achieve fine-grained program semantics and emphasize fault-related features. Subsequently, we employ an unsupervised graph learning algorithm to generate embeddings for these graphs. We then cluster these graph embeddings and select representative exemplars from the resulting clusters. Finally, these selected exemplars are combined with the D_t to form the training exemplars. In the *model iteration* stage, we adjust the weight of a sample's corresponding loss based on the model's confidence in that sample. As illustrated by the curve, let p denote the confidence of the model for a sample. The lower the model's confidence in a sample, the less it is considered to be mastered, and thus its corresponding loss weight is set higher, compelling the model to learn more from this sample. Conversely, for samples with higher confidence, the model's focus on them is reduced. Therefore, through the adaptive weighted loss, we can make the model focus more on old faults where forgetting has occurred and on new faults from the latest data. This serves to mitigate forgetting, facilitate the acquisition of new knowledge, and enhance the effectiveness of continual learning. To further alleviate forgetting on non-replayed samples and prevent overfitting to the representative exemplars, we also employ a regularization term to penalize significant changes in the model's parameters. Finally, the model $\mathcal{M}_{t-1}$ is optimized using a combination of the adaptive weighted loss and the regularization loss, yielding the updated model $\mathcal{M}_t$. Next, we describe each stage in detail.

3.3 Training Exemplars Generation

3.3.1 Feature Extraction. In the feature extraction stage, we first extract fine-grained and fault-enhanced program semantics from the logs in past datasets and convert them into a graph structure. We then employ an unsupervised graph embedding algorithm to generate a corresponding vector representation. Below, we explain in more detail of the feature extraction.

Program Semantic Extraction. First, we extract basic program semantics from the logs, including the program entities covered by the test and the intra-thread method execution order. Then, we further refine these semantics and incorporate fault-related features by also extracting the frequency of method calls and whether the call process involves a faulty entity. The intuition here is that extracting fine-grained program semantics and leveraging fault-related features enables a better representation of the log and its associated fault information. By using method call frequency, we can better differentiate between various method calls and their respective weights among all calls, thereby achieving a more fine-grained extraction of program semantics. On the other hand, whether a faulty entity is involved effectively enriches different method calls with fault-related information, thereby emphasizing the fault features. This, in turn, allows us to create more effective log embeddings and facilitate the subsequent selection of representative exemplars.

Specifically, as shown in the feature extraction in Figure 4, we first convert each log into a corresponding graph structure where each node represents a program entity, namely a package, a file, or a method. A dedicated node is also used to represent the log as a whole. Connections between entities at different hierarchical levels represent containment relationships and coverage information, while connections between method nodes represent the intra-thread method execution sequence. Next, to refine the program semantics and emphasize fault-related information, we calculate weights for the method execution edges based on their call frequency and whether the call involves the faulty method.

More specifically, given a log L, let T, P, F and M respectively denote the sets of threads, packages, files, and methods which are formed by extracting t, p, f, m from each line $l = (t, p, f, m)$ in L. Then, let G_L represent the graph for log L and the node set of G_L is defined as $V_L = T \cup P \cup F \cup M \cup \{n_L\}$, where n_L denote the special node representing the log itself. Let (u, v) denote node u point to node v. Then, we represent the containment relationships and coverage information by constructing (p, f) and (f, m) for each line, and (n_L, p) for all $p \in P$. Next, we represent the intra-thread method execution sequence by constructing (S_i^t, S_{i+1}^t), where $S^t = (m_1, m_2, \ldots, m_k)$ denotes the sequence of methods executed by t and $i \in [1, k-1]$. k is the length of S^t. The edge set E_L of G_L

is constructed by $H \cup I$, where H denotes the edges that represent the containment relationships and coverage information, and I denotes the edges that represent the intra-thread method execution sequence. After that, for each (m_i, m_j) in I, let p denote the number of times this call sequence appears in the log. The weight w of the edge (m_i, m_j) is then calculated by the following formula:

$$w = \log(p) + \log(p) \times (\mathbb{I}(m_i \text{ is faulty}) + \mathbb{I}(m_j \text{ is faulty})) \quad (1)$$

Here, $\mathbb{I}(\cdot)$ is an indicator function that returns 1 if the condition is true and 0 otherwise. In addition to reflecting the call frequency, the edge's weight is further increased if the method call involves the faulty method. Through this approach, we can achieve a more fine-grained extraction of program semantics and utilize fault features to further increase the weight of certain semantic elements. This allows the subsequent embedding process to emphasize this fault-related information, ensuring that the selection of representative exemplars can cover a more diverse range of samples.

Graph Embedding. After converting the logs into graphs, we need to further transform these graphs into corresponding vectors to facilitate the subsequent selection of representative exemplars. Since we assume access to all past data—a reasonable assumption in software testing, where all historical test results are preserved for tracking—we require an efficient embedding algorithm. Furthermore, as the graph embedding process is unlabeled, an unsupervised algorithm is necessary. Based on these two requirements, we select Node2Vec as our graph embedding algorithm. Node2Vec [14] is an effective and efficient unsupervised method that captures both local and global information within a graph through a weighted random walk strategy, allowing each node to learn a vector representation that reflects its role in the network structure. By using Node2Vec, we can encode the program semantics and fault information embedded in our graphs into corresponding vectors effectively.

Specifically, we first use SentenceBERT [39], a general-purpose natural language encoding tool, to encode the names of the various packages, files, methods, and the log itself, as their naming typically follows natural language conventions. The resulting vectors serve as the initial features for the corresponding nodes. We then apply Node2Vec to perform weighted learning on the graph. During this process, the fine-grained program semantics and fault features are more effectively embedded into the node vectors via the weighted random walks. Finally, we obtain a vector representation for the entire graph by averaging the learned node vectors.

3.3.2 Representative Exemplars Selection. In this step, we select representative exemplars based on the graph vectors generated in the preceding step. These exemplars are used to help the model review previously encountered data, thereby preventing catastrophic forgetting. Thus, the selected representative exemplars must be sufficiently diverse to cover a variety of faults. To this end, we first cluster the graph vectors and then perform sample selection. By clustering, we can partition samples according to different program semantics and fault features extracted before, and then select samples from these different clusters to achieve diversity.

Clustering. We adopt the DBScan algorithm (Density-Based Spatial Clustering of Applications) [8] to cluster the graph vectors of all past datasets into different groups. The reasons why utilizing the DBScan are twofold: First, specifying the number of clusters in the feature space is a non-trivial task. This renders clustering algorithms that require pre-definition of the number of clusters, such as the K-means algorithm used in previous work [12], unsuitable. DBScan is a density-based clustering algorithm, which groups together data points in high-density regions that are spatially close together, and thus, it does not need to pre-define the number of clusters. Second, DBScan has been demonstrated to be effective and efficient, and has been widely used in many domains [19, 42]. Thus, we employ DBScan to cluster the graph embeddings.

Exemplars Selection. Based on the feature clusters, we further identify the representative exemplars. Specifically, assume that after clustering we have obtained n clusters, from which we need to select a total of K representative exemplars. First, we count the number of samples in each cluster to determine how many exemplars should be selected from it. This allows us to dynamically select samples based on the size of clusters to ensure more representativeness [5]. More specifically, if the sample counts in the n clusters are $(c_1, c_2, \ldots, c_n)$, we randomly select samples from each cluster i that is proportional to its size relative to the total number of past samples, $|D_{1:t-1}|$. The number of samples to select from cluster i is calculated as $K \times \frac{c_i}{|D_{1:t-1}|}$. Finally, the selected samples are designated as the representative exemplars and are merged with the new dataset D_t to form the training exemplars. Note that throughout the iterative development process, software undergoes refactoring but retains much of its previous functionality and logic, rather than being completely rewritten. Therefore, reusing data from past versions is a meaningful approach, a conclusion that our experiments also support. Furthermore, even though the model may learn features of certain defects that are subsequently fixed in newer versions, this knowledge does not impair its ability to localize other faults.

3.4 Model Iteration

Adaptive Weighted Loss. In this step, we adaptively determine the loss weight for each sample by assessing the model's degree of mastery over it. The intuition here is that mitigating forgetting and mastering new knowledge can be unified into a single objective of enhancing the model's learning on samples it has not yet mastered. To achieve this, we estimate the model's degree of mastery over a sample based on its confidence (i.e., suspiciousness score) for each faulty entity within that sample [30]. Specifically, let $Y = (e_1, e_2, \ldots, e_n)$ be the set of all ground-truth faulty entities for a given sample, and let $P = (p_1, p_2, \ldots, p_n)$ be the corresponding suspiciousness scores calculated by the model for these entities. p_i is between 0 and 1. The weight w for this sample is then given by

$$w = (\frac{1}{n}\sum_{i=1}^{n} 1 - p_i)^{\gamma}, \quad (2)$$

where γ is a focusing parameter that controls the degree to which the model attends to unmastered versus mastered faults. The larger the value of γ, the more the model will neglect samples that are already well-mastered. When the model's degree of mastery over the involved faulty entities is higher (i.e., higher confidence), the weight calculated will be smaller. Conversely, the weight will increase. Through Eq. 2, we can dynamically adjust the loss weight during the training process based on the model's degree of mastery over a sample, thereby achieving more effective continual learning.

Finally, let $\mathcal{L}_i$ be the loss calculated by the model on this sample, and the corresponding weight is w_i. Then the adaptive weighted loss is defined as

$$\mathcal{L}_{AdaptiveWeighted} = \sum_{i=1}^{n} w_i \mathcal{L}_i, \tag{3}$$

where n is the number of training exemplars. In this way, we can enhance the model's learning on samples it has not yet mastered, which in turn mitigates forgetting and facilitates the acquisition of new knowledge, making continual learning more effective. Note that in learning-based fault localization, various loss functions can be used to calculate $\mathcal{L}$, such as *listwise*, *pairwise*, and *pointwise* loss function [32]. Our method operates on the model's final outputted suspiciousness scores and is therefore not coupled with any particular loss function. This makes our approach adaptable and compatible with these different loss formulations.

Parameter Regularization. Since we only select a subset of representative exemplars from past data for replay, the model may experience forgetting on the non-replayed samples or overfit to the representative exemplars, leading to sub-optimal effectiveness [12, 15]. Therefore, we add a quadratic penalty term to the loss to restrict large modifications to the network weights that are crucial for previous data. Specifically, our regularization loss is formulated as

$$\mathcal{L}_{Regularization} = \sum_{i} F_i (\theta_{t,i} - \theta_{t-1,i})^2 \tag{4}$$

Here, F_i represents the importance of parameter i for previous knowledge, and $\theta_{t,i}$ is the value of parameter i at iteration t. In Ciallo, we compute F_i using the Fisher Information Matrix [11] that $F_i = \nabla^2 \mathcal{L}(x, y|\theta_{t-1,i})$ where ∇^2 is the Laplace operator. The reason for choosing the Fisher Information Matrix is that it is simple, efficient, and widely used in continual learning to estimate the effect of each sample on the parameters [12, 20, 46]. The use of a squared penalty results in a smoother and more stable optimization process, thereby enhancing the effectiveness of the regularization [16, 21].

More specifically, at the end of the i−1-th iteration, we estimate how much information each model parameter holds about the learned faults using the current training exemplars through the Fisher Information Matrix. Then, in the i-th iteration, we can use this matrix to constrain parameter modifications. That is, if a parameter was deemed significant in the i−1-th iteration, any subsequent significant changes to it will be penalized more heavily, thereby reducing the model's catastrophic forgetting. The final loss $\mathcal{L}$ we use to optimize the model $\mathcal{M}_{t-1}$ is

$$\mathcal{L} = \mathcal{L}_{AdaptiveWeighted} + \mathcal{L}_{Regularization} \tag{5}$$

4 Evaluation

We evaluate Ciallo on the following research questions:

- RQ1: How does Ciallo's effectiveness compare to that of state-of-the-art techniques on continual learning? RQ1 consists of the following three sub-RQs:
  - RQ1.1: Overall effectiveness on all datasets.
  - RQ1.2: Effectiveness in learning from new data.
  - RQ1.3: Effectiveness in mitigating forgetting.
- RQ2: How do different components and main parameters within Ciallo affect its overall effectiveness?
- RQ3: How does the training efficiency of Ciallo compare to other techniques?

4.1 Experimental Setup

Benchmark. To evaluate our approach within the backdrop of large-scale software and system testing, we collaborated with our industrial partner, which operates multiple digital product lines worldwide. They provided us with logs from failed system tests for eleven software projects, collected over one year. Each software project consists of over one million lines of code and contains, on average, 170 packages, 2,000 files, and 50,000 methods. Each log corresponds to multiple faulty methods. We have a total of 3,800 logs, with an average of 101.24 threads, 122.68 packages, 361.28 files, and 1057.91 methods involved in each log. The engineers partitioned this data into 8 iterations based on the development timeline. Each iteration involves 475 logs on average, which we then randomly split into training and test sets using a 9:1 ratio. Our tool can also be applied to open-source software, provided that the software has been deployed, is in continuous operation, and requires log-based fault localization.

Baselines. To the best of our knowledge, we are the first to attempt continual learning for fault localization. Therefore, we select representative baselines from the continual learning literature and related work on continual learning for code models as our comparison methods. Additionally, we also include DeepGini, a representative method from test case selection for comparison. Specifically, we select the following methods as our baselines:

•**Upper** retrained the model from scratch on the current and all historical datasets. Following prior work [12, 46], we consider this as the performance upper bound in our scenario.

•**FT** directly fine-tune the model only on the new data at each iteration.

•**EMR** [45] randomly selects a subset of past data and combines it with the new data to fine-tune the model at each iteration.

•**EWC** [20] add regularization item through Fisher Information Matrix when fine-tuning on new data at each iteration.

•**REPEAT** [12] select representative exemplars for replay based on TF-IDF, K-means clustering, and model loss, and applies regularization based on the difference between old and new data.

•**DeepGini** [9] assesses sample uncertainty based on the model's output probabilities to select the most valuable samples, which are then combined with new data for replay.

We evaluate the effectiveness of our approach and the baselines on Falcon [23], a state-of-the-art backbone for log-based fault localization in large-scale software and system testing. Falcon operates by first converting logs into graphs and then applying contrastive learning on these graphs to perform fault localization, representing the current state-of-the-art framework in this domain.

Evaluation Metric. Following previous work, we adopt **Recall at Top-N** (N=1,3,5), **Mean First Rank** (MFR), and **Mean Average Rank** (MAR) as our evaluation metrics. Recall at Top-N denotes the proportion of all failed tests in which at least one faulty element is located within the first N positions. MFR measures the mean rank of the first faulty element across all failed tests. MAR is the mean of the average ranking of all the faulty entities of all failed tests. Higher Top-N and lower MFR and MAR suggest that the faulty entities are located closer to the top of the ranked list, indicating better localization performance. We use the *worst* ranking for the tied elements that have the same suspiciousness scores.

To evaluate the model's forgetting of old data, following previous work [46], we use the **Forgetting Measure** (FM) as a metric. Specifically, let $f_{j,k}$ denote the absolute difference between the best performance the model previously achieved on the j-th dataset and its performance on the j-th dataset at the k-th iteration. The FM at the k-th iteration is then given by $\frac{1}{k-1}\sum_{j=1}^{k-1} f_{j,k}$. A smaller FM value indicates a better ability to mitigate forgetting.

For RQ1.1, we evaluate the overall effectiveness of different methods by their performance across all datasets after the entire continual learning process, i.e., all iterations, is complete. For RQ1.2, we assess the model's effectiveness in learning new knowledge by recording its performance on the new dataset after each iteration, and then averaging these results across all iterations. For RQ1.3, we evaluate the model's effectiveness in mitigating forgetting by calculating the FM on various metrics during each iteration, and then averaging these FM results over all iterations.

Implementation. We build Ciallo based on PyTorch Geometric [10] and Scikit-Learn [36]. We use the open-source implementation [7] of Node2Vec [14] to perform graph embedding. Ciallo can scale to larger datasets because its feature extraction stage is inherently parallelizable. Consequently, we can employ multi-threading and GPU acceleration for program semantic extraction and graph embedding, allowing for efficient representative sample selection. Furthermore, during the model training phase, the adaptive weight for each sample within a mini-batch can be computed in parallel before the optimization step, thereby reducing time overhead.

We employed grid search to determine the optimal parameter combination. We set the focusing parameter γ in adaptive weighted loss to 2. We also explore the influence of the main parameters in Ciallo in the ablation study. For the baselines, we directly adopt their open-source implementations and empirically tune the hyperparameters by grid search because the original hyperparameters haven't been tested on our task. For all methods that need replay, we follow prior work [12] that sets the size of replayed exemplars K to 1% of the whole training data for comparison, which is considerably smaller than the whole size of the training data. We perform fault localization at the method level. Furthermore, all experiments are conducted with a fixed random seed to reduce variations across the experiments. Each method is trained until convergence to achieve optimal effectiveness and efficiency. More specifically, when the model converges, we select the checkpoint from the epoch that yielded the lowest loss as the final model. The final training time is recorded as the total time consumed from the start of training up to that epoch.

All experiments are conducted on a workstation with AMD Ryzen 9 3900XT, 32GB memory, and two RTX 4090 GPUs, running Ubuntu 20.04.

4.2 RQ1: Effectiveness

In this RQ, the primary objective is to evaluate the effectiveness of the Ciallo in continual learning. We conduct a comparative analysis of Ciallo against 6 baselines outlined in Section 4.1. Specifically, we evaluate the effectiveness of Ciallo from the following three perspectives.

Table 1: Performance of different baselines across all datasets after continual learning.

Techniques	Top-1 (↑)	Top-3 (↑)	Top-5 (↑)	MFR (↓)	MAR (↓)
Upper	0.68	0.75	0.81	14.7	20.1
No-update	0.36	0.4	0.46	47.3	52.8
FT	0.44	0.51	0.59	38.4	43.0
EMR	0.48	0.54	0.59	34.7	39.2
EWC	0.42	0.49	0.55	41.8	49.2
REPEAT	0.52	0.6	0.67	24.6	32.8
DeepGini	0.50	0.56	0.61	27.2	36.2
Ciallo	**0.61**	**0.68**	**0.74**	**19.4**	**24.9**

RQ1.1: Overall Effectiveness. In RQ1.1, we evaluate the effectiveness of different methods based on their overall performance across all datasets after the continual learning is complete (i.e., after all iterations have finished). Table 1 presents the experimental results. From the table, we observe that: **First**, First, the no-update method achieved the lowest effectiveness among all approaches. For the no-update model, we train it on the data from the first iteration and then freeze its parameters. This result underscores the necessity of retraining or continual learning; otherwise, the model will become unusable during the software iteration process. **Second**, compared to the straightforward method FT that directly fine-tunes the model on new data, Ciallo achieves more effective continual learning. Specifically, Ciallo improves upon FT by 38.63%, 33.33%, 25.42%, 49.47% and 42.09% on Top-1, Top-3, Top-5, MFR, and MAR, respectively. These results indicate that Ciallo can effectively improve the model's performance in the continual learning scenario compared to only fine-tuning. **Third**, compared to the baseline methods, Ciallo consistently outperforms the other approaches across various metrics, and Ciallo is also the method that achieves performance closest to the upper bound among all baselines studied. Notably, Ciallo achieves Top-1 localization accuracy of 0.61, representing significant improvements of 27.08%, 45.23%, 17.30%, 21.99% over the EMR, EWC, REPEAT, and DeepGini, respectively. Additionally, MFR and MAR metrics also exhibit substantial enhancements, with Ciallo showing an 21.13% improvement in MFR and a 24.08% improvement in MAR compared to the best-performing baseline, REPEAT. This performance improvement can be partially attributed to the feature extraction and representative sample selection methods of Ciallo, which are better suited for continual learning in the context of fault localization. REPEAT utilizes TF-IDF for feature extraction, while DeepGini only uses the model's probability. The former struggles to capture program semantics and fault characteristics from logs, whereas the latter may lead to the selection of representative exemplars with repetitive features. In contrast, Ciallo extracts fine-grained program semantics from logs and enhances the weights of specific calls using fault features, thereby effectively extracting features and boosting its continual learning performance.

RQ1.2: Effectiveness in learning new data. In RQ1.2, we evaluate the ability of different continual learning methods to learn from new data by assessing their performance on new data after each iteration and then averaging these results across all iterations. Table 2 shows the experimental results. From the table, we observe that Ciallo also surpasses the other methods in its performance on

Table 2: Average performance of different baselines on new datasets after each iteration.

Techniques	Top-1 (↑)	Top-3 (↑)	Top-5 (↑)	MFR (↓)	MAR (↓)
Upper	0.62	0.69	0.77	17.6	25.8
FT	0.47	0.56	0.65	31.8	37.4
EMR	0.39	0.44	0.53	39.7	46.8
EWC	0.33	0.41	0.47	45.8	54.3
REPEAT	0.41	0.52	0.61	33.1	39.5
DeepGini	0.43	0.51	0.59	35.3	42.8
Ciallo	**0.54**	**0.64**	**0.72**	**24.8**	**30.6**

Table 3: Forgetting measure on old data for different baselines after each iteration. $FM_{t1}, FM_{t3}, FM_{t5}, FM_F$, and FM_A denote the forgetting measures on Top-1, Top-3, Top-5, MFR, and MAR, respectively.

Techniques	FM_{t1} (↓)	FM_{t3} (↓)	FM_{t5} (↓)	FM_F (↓)	FM_A (↓)
Upper	0.01	0.01	0.02	1.2	2.9
FT	0.17	0.22	0.25	12.4	16.8
EMR	0.12	0.16	0.14	9.7	11.1
EWC	0.14	0.17	0.15	10.4	13.5
REPEAT	0.09	0.10	0.08	6.1	8.7
DeepGini	0.11	0.12	0.12	6.3	9.4
Ciallo	**0.04**	**0.06**	**0.05**	**4.5**	**6.2**

learning new knowledge when compared to the studied baselines. Specifically, the improvements of Ciallo over all the baselines are remarkable, ranging from 14.89% to 63.63% on Top-1, ranging from 14.28% to 56.09% on Top-3, ranging from 10.76% to 53.19% on Top-5, ranging from 22.01% to 45.85% on MFR, and ranging from 18.18% to 43.64% on MAR. The results underscore Ciallo's effectiveness in learning the new data compared to other studied baselines across different iterations.

RQ1.3: Effectiveness in mitigating forgetting. In RQ1.3, we evaluate the effectiveness of different baselines in mitigating forgetting during the continual learning process by measuring the FM at each iteration. The FM assesses how much a model's performance on a specific metric for a given dataset has dropped after the current iteration compared to its previous best performance, which helps us evaluate the phenomenon of model forgetting. A large FM indicates that the model has suffered from catastrophic forgetting. We calculate the FM for various metrics at each round of iteration and then average the results across all iterations for our evaluation. The experimental results are shown in Table 3, where each column represents the average performance forgotten by the model on that specific metric. From the table, we have the following observation: **First**, simply fine-tuning on new data leads to severe forgetting. Specifically, the FT method's performance dropped by 0.17, 0.22, 0.25, 12.4, and 16.8 on Top-1, Top-3, Top-5, MFR, and MAR, respectively. This indicates that catastrophic forgetting also occurs during the iterative training of fault localization models. This is because FT is only fine-tuned on new data without any review of old data. This leads to FT being more effective at learning new faults but suffering from catastrophic forgetting. For example, in Table 2, we observe that FT is one of the best-performing methods; its Top-1 accuracy even surpasses that of REPEAT and DeepGini. However, in Tables 1 and 3, FT's effectiveness is severely compromised due to forgetting. **Second**, compared to other continual learning methods, Ciallo achieves the best effectiveness in resisting catastrophic forgetting. Particularly, compared to the

Table 4: The effectiveness of different variants.

Techniques	Top-1 (↑)	Top-3 (↑)	Top-5 (↑)	MFR (↓)	MAR (↓)
Ciallo$_e$	0.56	0.60	0.68	22.1	29.4
Ciallo$_f$	0.53	0.58	0.62	24.7	30.5
Ciallo$_a$	0.51	0.55	0.64	26.7	35.4
Ciallo$_r$	0.55	0.61	0.67	22.5	30.1
Ciallo	**0.61**	**0.68**	**0.74**	**19.4**	**24.9**

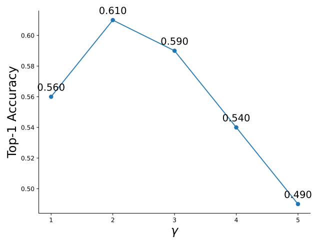


Figure 5: The effect of γ.

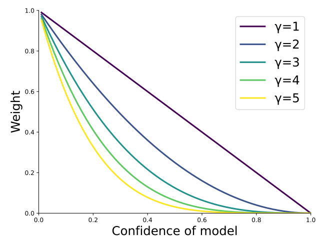


Figure 6: The weight under different γ.

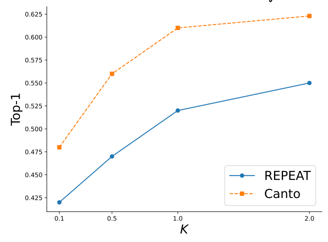


Figure 7: The effect of K.

best-performing baseline, REPEAT, Ciallo shows an improvement of 55.55%, 40%, 37.5%, 26.29% and 28.73% on the FM for Top-1, Top-3, Top-5, MFR, and MAR. Similar trends are also observed among the other baselines. Based on the experimental results from RQ1.2 and RQ1.3, Ciallo not only effectively resists forgetting during the continual learning process but also excels at learning new knowledge. This indicates that Ciallo's adaptive weight loss function is better than the original loss function at enabling the model to focus on unmastered samples, including both forgotten and new faults. In contrast, REPEAT assigns the same weight to all samples, which leads to sub-optimal effectiveness compared to Ciallo.

To further confirm the observations in RQ1, we have followed previous works [23, 32] to perform the Wilcoxon signed-rank test [48] with Bonferroni corrections [6] to investigate the statistical significance between Ciallo and other baselines. The results show that Ciallo is significantly better than all studied baselines at the significance level of 0.05.

4.3 RQ2: Ablation Study

In this RQ, we conduct a series of ablation studies to further analyze the impact of each component of Ciallo and study the main parameters in Ciallo. We adopt the same experimental setup as in RQ1.1, evaluating the model's effectiveness on the entire dataset after all iterations are complete.

Effect of different components. In particular, we consider the following four variants of Ciallo: Ciallo$_e$ removes the weight derived from the method call frequency (i.e., the first term of Eq. 1). Ciallo$_f$ removes the weight derived from the fault entities (i.e., the

Table 5: Training time cost of studied techniques. CE presents the carbon emissions per iteration for each method.

Techniques	Cost	CE	Techniques	Cost	CE	Techniques	Cost	CE
Upper	4h57min	1906g	EWC	1h47min	687g	FT	1h21min	520g
REPEAT	2h33min	982g	EMR	2h14min	860g	DeepGini	2h03min	789g
Ciallo	**2h43min**	1046g						

second term of Eq. 1). Ciallo$_a$ removes the adaptive weight from the loss function. Ciallo$_r$ removes the regularization item from the loss function. Table 4 summarizes the study results, and we have the following observations. First, removing any single component leads to a noticeable decline in the effectiveness of Ciallo. This indicates that each module of Ciallo contributes to its effectiveness in continual learning. Second, when comparing Ciallo$_f$ and Ciallo$_e$, we find that removing the weights derived from fault nodes results in a more significant performance loss. Specifically, the Top-5 accuracy of Ciallo$_f$ decrease by 8.82% compared to Ciallo$_e$. This suggests that utilizing fault nodes to increase the weights of relevant program semantics is more effective than using method call frequency. Third, removing the adaptive loss weight causes the most substantial drop in performance. In particular, compared to Ciallo, the Top-1, Top-3, and Top-5 accuracies of Ciallo$_a$ decrease by 16.39%, 19.11% and 13.51%, respectively. This further demonstrates that our adaptive weight loss can effectively help mitigate catastrophic forgetting and better learn new knowledge.

Effect of focusing parameter γ. Furthermore, we conducted a parameter sensitivity analysis on the focusing parameter γ in the adaptive loss weight. γ is a key hyperparameter in our method. We evaluated γ with values ranging from 1 to 5. The experimental results are shown in Figures 5 and 6. Figure 5 presents the Top-1 accuracy of Ciallo under different γ values. Figure 6 illustrates the weight assigned for different model confidences under varying γ. From Figure 6, we can observe that as γ increases, the weight assigned for the same level of confidence decreases, which compels the model to focus more on samples with lower confidence. However, when γ is very large, it causes the model to nearly ignore high-confidence samples, leading to a drop in effectiveness. This is also reflected in Figure 5. We found that as γ becomes larger, the model's Top-1 accuracy declines significantly, and the best effectiveness is achieved when γ is set to 2.

Effect of number of representative exemplars K. We also analyze the number of representative exemplars K. Specifically, we compared the effectiveness of Ciallo and the best-performing method, REPEAT, under different values of K. Following prior work [12], we evaluated the Top-1 accuracy for K values of 0.1%, 0.5%, 1%, and 2% on the whole training data. The experimental results are shown in Figure 7. From the figure, we observe that the Top-1 accuracy of both methods improves as K increases. It is reasonable as the model's ability to mitigate forgetting becomes stronger with an increasing number of replay samples. Ciallo performs better than REPEAT across all different values of K, which confirms the effectiveness of our method.

4.4 RQ3: Efficiency

This RQ empirically analyzes the efficiency of Ciallo. Table 5 presents the average time cost and carbon emissions of a single iteration in the continual learning process. We refer to the ISO/IEC 21031:2024 standard[2] to calculate the carbon emissions As shown in the table, Ciallo requires more time to complete one iteration and produce more carbon compared to the studied baselines. This is mainly because Ciallo needs to build a graph from the log and regenerate graph embeddings using Node2Vec and select representative exemplars during each round of iteration. Nevertheless, it is important to emphasize that Ciallo achieves significantly better performance in locating faults than the baselines, as demonstrated in Section 4.2. Therefore, we believe such a time cost of Ciallo is worthwhile for achieving better continual learning effectiveness. Furthermore, compared to the upper, Ciallo still saves approximately half of the training time and half the carbon emissions, and this improvement will be even more pronounced in scenarios with larger data volumes, striking a balance between performance and energy consumption. We also conducted an experiment to evaluate the effectiveness of the retraining method versus Ciallo under the same time budget. Specifically, we constrained the training time for retraining to not exceed that of Ciallo, as Ciallo requires less training time. The experiment results in Table 6 show that under the same time budget, Ciallo performs better than full-data retraining, indicating that Ciallo provides a practical speedup.

Table 6: Compare Ciallo to Upper under the same time budget.

Techniques	**Top-1(↑)**	**Top-3(↑)**	**Top-5(↑)**	**MFR(↓)**	**MAR(↓)**
Upper	0.56	0.63	0.70	22.2	28.7
Ciallo	**0.61**	**0.68**	**0.74**	**19.4**	**24.9**

5 Discussion

5.1 Threats to Validity

The main threat to **internal** validity lies in the correctness of the implementation of Ciallo, the compared approaches, and experimental scripts. To reduce this threat, we adopt the open-source implementations of the compared approaches, build Ciallo based on widely used libraries, and carefully check the source code of Ciallo and the experimental scripts. The main threat to **external** validity lies in the selection of subjects in our study. To mitigate this threat, we perform experiments on an industrial dataset provided by our industrial partner, which contains 3800 logs collected from 8 iterations over one year, sourced from 11 large-scale software systems each exceeding one million lines of code. Furthermore, we compare Ciallo with 6 continual learning baselines in our experiments. Moreover, we collected software from diverse product lines, and Ciallo's design is independent of specific development guidelines or practices, so it should remain effective in other organizations with different development habits. Additionally, while development style may slightly alter how a fault manifests, the fundamental nature of different fault types is largely consistent, supporting the broader applicability of our approach. The main threat to **conclusion** validity lies in whether our conclusions are statistically reliable. To reduce this threat, apply statistical hypothesis testing to our findings. Specifically, we followed previous works

[2]https://www.iso.org/standard/86612.html

to perform the Wilcoxon signed-rank test with Bonferroni corrections to investigate the statistical significance between Ciallo and other baselines. The main threat to **construct** validity lies in the parameters in Ciallo and the metrics used in our experiments. To reduce the threat from the parameters in Ciallo, we present the detailed parameter setting in our project website and investigate the impact of the main parameters in Section 4.3. To reduce the threat from metrics, we employed various metrics that are widely used in prior fault localization studies and continual learning [46].

5.2 Implications for SE

While the growing accumulation of data from SE tasks has led to the emergence of numerous AI for SE methods, existing work has primarily focused on devising better training methodologies or more effective data processing techniques. However, given the dynamic nature of software and the real world, the maintenance and iteration of these software systems also pose a significant challenge. In addition, we show that simply applying continual learning to the SE domain yields only sub-optimal effectiveness. To achieve high effectiveness on SE tasks, tailored designs for SE are required.

6 Related work

Fault Localization. The goal of automated fault localization (FL) methods is to automatically identify faulty program entities. Generally, FL techniques utilize static or dynamic information to calculate a suspiciousness score and then rank program entities based on this score. Currently, FL methods can be classified into several categories, including spectrum-based [1, 4, 40], mutation-based [34, 35], change-based [2, 47], and learning-based [23, 25, 32] approaches. Among these, spectrum-based and learning-based methods are the most representative techniques. *Spectrum-Based fault localization* (SBFL) [1, 4, 17, 40, 49, 50, 53] methods count the number of passed and failed test cases that cover each program entity and then use a specific formula to calculate that entity's suspiciousness score. The key insight of such methods is that entities covered by more failed tests and fewer passed tests are more likely to be faulty. *Learning-based fault localization* [23–27, 32, 33, 37, 44, 54, 55] are currently the most popular FL techniques. They leverage deep learning to better learn fault-related features or combine features from different dimensions to achieve superior localization performance. For example, Falcon [23] utilizes contrastive learning to better leverage program semantics and embed them from logs for FL in large-scale software. Legato [24] concentrates on leveraging semi-supervised learning to enhance the performance of graph-learning-based fault localization models when labeled samples are insufficient. Compared with Falcon and Legato, Ciallo addresses a different scenario that uses continual learning to improve the effectiveness of fault localization models in the context of software evolution while simultaneously reducing the overhead of model training.

Continual Learning. Continual learning addresses the challenge of catastrophic forgetting, where acquiring new knowledge degrades previously learned information [46]. Continual learning methods can be divided into two main categories: replay-based and regularization-based approaches. *Replay-based methods* [31, 38, 43, 45] mitigate model forgetting by accessing past knowledge and mixing it with new data during training. *Regularization-based methods* [20, 22, 28, 41] impose constraints on the learning process to protect learned knowledge, typically by adding a penalty to the loss function. In software engineering, work related to continual learning has primarily focused on the iteration of code models [12, 51] and anomaly detection models [52], as well as on test case selection [3, 9, 13, 29]. For example, in the domain of code models, REPEAT [12] was the first work to focus on continual learning. In anomaly detection, recent studies [52] have focused on the data shift problem between training and test sets, viewing test case selection as a continual learning strategy to mitigate this issue. Test case selection methods typically estimate uncertainty based on the model's output to select the most valuable samples, which are then used to train the model, thereby achieving continual learning. For instance, DeepGini [9] assesses sample uncertainty using the model's probability distribution to select samples and then fine-tunes the model on these samples to boost model performance.

7 Conclusion

In this paper, we propose Ciallo, a novel continual learning framework specifically designed for large-scale software fault localization. In each iteration of continual learning, Ciallo first extracts fine-grained program semantics from the logs of past datasets and enhances them using fault characteristics. It then uses an unsupervised graph embedding algorithm to obtain corresponding embeddings and selects representative exemplars via clustering. After mixing the representative exemplars with new data to form the training exemplars, Ciallo estimates its degree of mastery over each sample based on the model's suspiciousness scores for the faulty entities. It increases the weights of samples with a lower degree of mastery, enabling the model to focus more on learning from them, which in turn mitigates catastrophic forgetting and facilitates the learning of new knowledge. Furthermore, Ciallo also employs regularization to prevent overfitting on the training exemplars and to mitigate the forgetting of old data not included in the representative samples. The experimental results demonstrate the effectiveness of Ciallo in large-scale software fault localization under the continual learning setting.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research is supported by the National Natural Science Foundation of China (62372227 and 62402214), the Natural Science Foundation of Jiangsu Province (BK20241194), the China Postdoctoral Science Foundation (2025T180420, 2025M781473), and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (KYCX25_0368). Zhong Li is also supported by the Postdoctoral Fellowship Program of CPSF (GZB20250386) and the Jiangsu Funding Program for Excellent Postdoctoral Talent (2025ZB317).

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *PRDC*. IEEE Computer Society, 39–46.
[2] An Ran Chen, Tse-Hsun (Peter) Chen, and Junjie Chen. 2022. How Useful is Code Change Information for Fault Localization in Continuous Integration?. In *ASE*. ACM, 52:1–52:12.
[3] Jialuo Chen, Jingyi Wang, Xingjun Ma, Youcheng Sun, Jun Sun, Peixin Zhang, and Peng Cheng. 2023. QuoTe: Quality-oriented Testing for Deep Learning Systems.

ACM Trans. Softw. Eng. Methodol. 32, 5 (2023), 125:1–125:33.

[4] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. 2002. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN*. IEEE Computer Society, 595–604.

[5] William G. Cochran. 1977. *Sampling Techniques, 3rd Edition.* John Wiley.

[6] Olive Jean Dunn. 1961. Multiple comparisons among means. *Journal of the American statistical association* 56, 293 (1961), 52–64. https://doi.org/10.2307/2282330

[7] Eliorc. 2025. Python3 implementation of the node2vec algorithm. https://github.com/eliorc/node2vec [Online; accessed 9-July-2025].

[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*. AAAI Press, 226–231.

[9] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. DeepGini: prioritizing massive tests to enhance the robustness of deep neural networks. In *ISSTA*. ACM, 177–188.

[10] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[11] Ronald A Fisher. 1922. On the mathematical foundations of theoretical statistics. *Philosophical transactions of the Royal Society of London. Series A, containing papers of a mathematical or physical character* 222, 594-604 (1922), 309–368.

[12] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping Pace with Ever-Increasing Data: Towards Continual Learning of Code Intelligence Models. In *ICSE*. IEEE, 30–42.

[13] Xinyu Gao, Yang Feng, Yining Yin, Zixi Liu, Zhenyu Chen, and Baowen Xu. 2022. Adaptive Test Selection for Deep Neural Networks. In *ICSE*. ACM, 73–85.

[14] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *KDD*. ACM, 855–864.

[15] Xu Han, Yi Dai, Tianyu Gao, Yankai Lin, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. 2020. Continual Relation Learning via Episodic Memory Activation and Reconsolidation. In *ACL*. Association for Computational Linguistics, 6429–6440.

[16] Arthur E. Hoerl and Robert W. Kennard. 2000. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics* 42, 1 (2000), 80–86.

[17] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*. ACM, 467–477.

[18] Chris F. Kemerer and Sandra Slaughter. 1999. An Empirical Approach to Studying Software Evolution. *IEEE Trans. Software Eng.* 25, 4 (1999), 493–509.

[19] Kamran Khan, Saif ur Rehman, Kamran Aziz, Simon Fong, Sababady Sarasvady, and Amrita Vishwa. 2014. DBSCAN: Past, present and future. In *ICADIWT*. IEEE, 232–238.

[20] James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2016. Overcoming catastrophic forgetting in neural networks. *CoRR* abs/1612.00796 (2016).

[21] Anders Krogh and John A. Hertz. 1991. A Simple Weight Decay Can Improve Generalization. In *NIPS*. Morgan Kaufmann, 950–957.

[22] Kibok Lee, Kimin Lee, Jinwoo Shin, and Honglak Lee. 2019. Overcoming Catastrophic Forgetting With Unlabeled Data in the Wild. In *ICCV*. IEEE, 312–321.

[23] Chun Li, Hui Li, Zhong Li, Minxue Pan, and Xuandong Li. 2025. Enhancing Fault Localization in Industrial Software Systems via Contrastive Learning. In *ICSE*. IEEE, 691–703.

[24] Chun Li, Hui Li, Zhong Li, Minxue Pan, and Xuandong Li. 2025. Improving Graph Learning-Based Fault Localization with Tailored Semi-supervised Learning. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE069 (June 2025), 23 pages.

[25] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 169–180.

[26] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 92:1–92:30.

[27] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *ICSE*. IEEE, 661–673.

[28] Zhizhong Li and Derek Hoiem. 2016. Learning Without Forgetting. In *ECCV (4) (Lecture Notes in Computer Science, Vol. 9908)*. Springer, 614–629.

[29] Zhong Li, Zhengfeng Xu, Ruihua Ji, Minxue Pan, Tian Zhang, Linzhang Wang, and Xuandong Li. 2024. Distance-Aware Test Input Selection for Deep Neural Networks. In *ISSTA*. ACM, 248–260.

[30] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. 2017. Focal Loss for Dense Object Detection. In *ICCV*. IEEE Computer Society, 2999–3007.

[31] David Lopez-Paz and Marc'Aurelio Ranzato. 2017. Gradient Episodic Memory for Continual Learning. In *NIPS*. 6467–6476.

[32] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *ESEC/SIGSOFT FSE*. ACM, 664–676.

[33] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving Fault Localization and Program Repair with Deep Semantic Features and Transferred Knowledge. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1169–1180.

[34] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *ICST*. IEEE Computer Society, 153–162.

[35] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test. Verification Reliab.* 25, 5-7 (2015), 605–628.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[37] Md Nakhla Rafi, Dong Jae Kim, An Ran Chen, Tse-Hsun (Peter) Chen, and Shaowei Wang. 2024. Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1937–1959.

[38] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. 2017. iCaRL: Incremental Classifier and Representation Learning. In *CVPR*. IEEE Computer Society, 5533–5542.

[39] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP/IJCNLP (1)*. Association for Computational Linguistics, 3980–3990.

[40] Sofia Reis, Rui Abreu, and Marcelo d'Amorim. 2019. Demystifying the Combination of Dynamic Slicing and Spectrum-based Fault Localization. In *IJCAI*. ijcai.org, 4760–4766.

[41] Hippolyt Ritter, Aleksandar Botev, and David Barber. 2018. Online Structured Laplace Approximations for Overcoming Catastrophic Forgetting. In *NeurIPS*. 3742–3752.

[42] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3 (2017), 19:1–19:21.

[43] Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. 2017. Continual Learning with Deep Generative Replay. In *NIPS*. 2990–2999.

[44] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 273–283.

[45] Hong Wang, Wenhan Xiong, Mo Yu, Xiaoxiao Guo, Shiyu Chang, and William Yang Wang. 2019. Sentence Embedding Alignment for Lifelong Relation Extraction. In *NAACL-HLT (1)*. Association for Computational Linguistics, 796–806.

[46] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. 2024. A Comprehensive Survey of Continual Learning: Theory, Method and Application. *IEEE Trans. Pattern Anal. Mach. Intell.* 46, 8 (2024), 5362–5383.

[47] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE Trans. Software Eng.* 47, 11 (2021), 2348–2368.

[48] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*. Springer, 196–202. https://doi.org/10.1007/978-1-4612-4380-9_16

[49] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Trans. Reliab.* 63, 1 (2014), 290–308.

[50] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.* 22, 4 (2013), 31:1–31:40.

[51] Prateek Yadav, Qing Sun, Hantian Ding, Xiaopeng Li, Dejiao Zhang, Ming Tan, Parminder Bhatia, Xiaofei Ma, Ramesh Nallapati, Murali Krishna Ramanathan, Mohit Bansal, and Bing Xiang. 2023. Exploring Continual Learning for Code Generation Models. In *ACL (2)*. Association for Computational Linguistics, 782–792.

[52] Jiongchi Yu, Xiaofei Xie, Qiang Hu, Bowen Zhang, Ziming Zhao, Yun Lin, Lei Ma, Ruitao Feng, and Frank Liauw. 2025. CAShift: Benchmarking Log-Based Cloud Attack Detection under Normality Shift. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE076 (June 2025), 23 pages.

[53] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *ICSM*. IEEE Computer Society, 23–32.

[54] Wei Zheng, Desheng Hu, and Jing Wang. 2016. Fault Localization Analysis Based on Deep Neural Network. *Mathematical Problems in Engineering* 2016 (01 2016), 1–11. https://doi.org/10.1155/2016/1820454

[55] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2021. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Trans. Software Eng.* 47, 2 (2021), 332–347.