Unseen Anomaly Detection from System Logs

YANNI TANG, School of Computer Science, The University of Auckland, New Zealand
ZHUOXING ZHANG, School of Computer Science, The University of Auckland, New Zealand
LANTING FANG, School of Computer Science and Technology, Beijing Institute of Technology, China
SEBASTIAN LINK, School of Computer Science, The University of Auckland, New Zealand
WU CHEN, College of Computer and Information Science, Southwest University, China
KAIQI ZHAO*, Shenzhen Key Laboratory of Internet Information Collaboration, Harbin Institute of Technology, Shenzhen, China

Detecting anomalies in system logs effectively is crucial for ensuring software reliability. However, most existing log anomaly detection methods operate under the assumption that anomalous patterns remain consistent between the training and testing phases. This limitation hinders their ability to identify previously unseen anomalies. In real-world scenarios, software upgrades may introduce novel anomalous patterns that deviate from the training distribution, a challenge we refer to as anomaly shift. To address this, we propose UnseenLog, a novel log anomaly detection framework designed to identify anomalous code files whose representations shift from the training distribution. UnseenLog introduces a MinMax strategy to select augmented anomaly samples that strike a balance between reliability and novelty, enriching the training set with additional pseudo-label samples. Furthermore, we propose Recurrent Iterative Selection and Enhancement (RISE), a training scheme that progressively enhances the graph model through competitive model selection and adaptive data enhancement, thereby improving robustness against novel anomalies. Extensive experiments on real-world datasets demonstrate that UnseenLog significantly outperforms state-of-the-art baselines, achieving consistent improvements in multiple real-world datasets. The code is available at *https://github.com/ZhuoxingZhang/UnseenLog*.

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

Additional Key Words and Phrases: Log Anomaly Detection; Unseen Anomalies; MinMax; Anomaly Shift

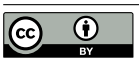
ACM Reference Format:
Yanni Tang, Zhuoxing Zhang, Lanting Fang, Sebastian Link, Wu Chen, and Kaiqi Zhao. 2026. Unseen Anomaly Detection from System Logs. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 91 (February 2026), 23 pages. https://doi.org/10.1145/3786705

1 Introduction

As the complexity of software systems continues to increase, effectively monitoring and analyzing system logs to identify potential anomalies has become an important research topic during maintenance phases [10, 13]. System logs, which are automatically generated during runtime, provide a

*Corresponding author.

Authors' Contact Information: Yanni Tang, School of Computer Science, The University of Auckland, New Zealand, ytan370@aucklanduni.ac.nz; Zhuoxing Zhang, School of Computer Science, The University of Auckland, New Zealand, zzha969@aucklanduni.ac.nz; Lanting Fang, School of Computer Science and Technology, Beijing Institute of Technology, China, ltfang@bit.edu.cn; Sebastian Link, School of Computer Science, The University of Auckland, New Zealand, s.link@auckland.ac.nz; Wu Chen, College of Computer and Information Science, Southwest University, China, chenwu@swu.edu.cn; Kaiqi Zhao, Shenzhen Key Laboratory of Internet Information Collaboration, Harbin Institute of Technology, Shenzhen, China, zhaokaiqi@hit.edu.cn.



ACM 2836-6573/2026/2-ART91
https://doi.org/10.1145/3786705

non-intrusive and real-time view of the system's dynamic behavior. In contrast, although analysis of source code can be useful in some contexts, source code is often inaccessible in production environments due to deployment constraints or security policies. Moreover, many issues that are manifested through dynamic system behavior, such as unexpected interactions between components or subtle business logic bugs, are difficult to detect through code analysis, but often leave identifiable traces in the logs [33]. Consequently, various automated anomaly detection methods based on logs have emerged, aiming to identify anomalous patterns that deviate from normal behavior, thereby assisting people in efficiently locating and diagnosing issues [3, 5, 16, 22]. Particularly, when anomalies can be accurately traced to specific code files, the efficiency of debugging and fixing problems is significantly improved [28]. Therefore, achieving more efficient anomaly localization at the code-file level based on system logs has become an important direction.

Early research on system log anomaly detection primarily focused on rules or expert-defined performance metrics [19, 24, 38]. However, these methods require substantial domain knowledge, limiting their coverage and accuracy [8, 14, 15, 18, 35]. Recently, several deep learning–based log anomaly detection approaches have been proposed, which can be broadly categorized into sequence-based models and graph-based models. Sequence-based models represent log events as sequences to capture their temporal dependencies [3, 6, 7, 23, 37]. Although these methods have demonstrated certain effectiveness, they often struggle to capture the inherent structural relationships within logs. In contrast, graph-based models represent logs as graphs and leverage graph representation learning techniques to automatically learn both the features and internal structural correlations of the logs [30, 34, 41]. By understanding the system interaction patterns embedded in logs, graph-based methods enable more fine-grained anomaly detection [28].

Despite notable progress, most existing log anomaly detection methods assume that *the distribution of log patterns remains unchanged between training and inference phases*. However, this assumption often fails to hold in real-world systems. In modern software systems, continuous iteration is the norm, including functionality enhancements, module replacements, and other structural changes. Such modifications often introduce new code paths and interaction patterns, leading to an evolution in the system's overall behavioral patterns. These behavioral-level changes in the software system may produce new types of anomaly patterns that were never observed during training. We refer to such anomalies as *unseen-pattern anomalies*, meaning novel anomalous graph patterns that the model has not encountered or learned during training.

Fig. 1 illustrates three representative cases of anomalous log graphs encountered during the inference phase in a forum system [17, 28]. In each graph, a node denotes a log message associated with a specific code file, and an edge indicates an invocation (call) relationship between corresponding code files. For each case, the left and middle log graphs correspond to log segments observed during the training phase, while the right graph depicts an anomalous log segment that appears only during inference. Cases 1 and 2 demonstrate scenarios in which the model encounters unseen anomalous patterns during inference. Taking code file 15 in Case 1 as an example, this node corresponds to a code file responsible for performing identity and permission checks when a user attempts to post a message (article or comment). During the training phase, only log graphs corresponding to the normal execution of these code files are included, as shown in the left and middle graphs of Case 1. However, during inference, the right graph presents a novel anomalous pattern. Specifically, in this anomalous case, a bug in the code file represented by node 15 leads to a failure in user authentication, which then propagates error information to node 1. As a result, node 1 fails to execute subsequent actions, such as posting an article or a comment, and does not invoke downstream nodes. These include node 8 and node 9 (responsible for posting an article on the left log graph), and node 13 (responsible for posting a comment on the middle log graph). In contrast, Case 3 reflects a more common scenario where the anomalous pattern encountered

during inference was already learned by the model during training. Specifically, node 12 behaves normally when invoked solely by node 20. However, if node 20 simultaneously invokes both node 18 and node 12, node 12 becomes anomalous. This co-invocation pattern and its corresponding label were already captured in the training phase.

Existing graph-based methods typically learn a mapping from graph patterns (i.e., structural and feature-based representations) to labels. While effective in cases like Case 3, where patterns remain consistent between training and inference, they often struggle to generalize to cases like Cases 1 and 2 in Fig. 1, where anomalies introduce previously unseen graph patterns that were absent during training. Although some graph out-of-distribution (OOD) methods offer partial solutions to this problem, they primarily focus on capturing features of the entire graph or local subgraphs, such as by mining invariant substructures [25, 39]. Furthermore, these methods are generally designed to distinguish out-of-distribution nodes from in-distribution classes, rather than to provide explicit and interpretable categorization of anomalous nodes, which is often essential in anomaly detection tasks.

To characterize Cases 1 and 2, we propose the concept of *anomaly shift*, which refers to a distributional shift in anomalous data. Importantly, anomaly shift not only changes the distribution of anomalies but can also indirectly affect the representation space of normal samples, potentially changing their distribution boundaries. This shift presents a major challenge for anomaly detection, as it undermines their ability to generalize to novel log patterns encountered during inference. Our research focuses on tackling this problem in log anomaly detection, specifically by identifying anomaly patterns that are not implied in the training data.

Addressing unseen-pattern anomalies introduces two significant challenges. First, since such anomalies do not appear during training, it is difficult for the model to generalize effectively based solely on the observed data. A reasonable solution is to introduce data augmentation techniques that generate high-quality synthetic samples to enhance the model's ability to adapt to new anomalous patterns. Rather than replicating unseen anomalies, the augmented data serve to enrich the diversity of the training distribution, thereby improving the model's generalization and robustness to previously unobserved behaviors. Second, data augmentation itself may introduce noise, particularly in graph-based settings, where small perturbations in node attributes or topological structure may alter the graph's characteristics and mislead the model. Therefore, an effective method is needed to mitigate the adverse impact of noisy augmented samples.

To address the aforementioned challenges, we propose a log anomaly detection framework, Unseen Anomalous Log Pattern Detection at Code File Level (UnseenLog), to identify unseen anomalous code files from the system logs. UnseenLog adopts an iterative optimization strategy to progressively enhance the generalization capability of the model. To address the first challenge, UnseenLog introduces a MinMax strategy to select high-quality augmented anomalous samples that potentially contain novel patterns, thereby enriching the training set. To address the second challenge, we further propose a Recurrent Iterative Selection and Enhancement (RISE) scheme. RISE conducts competitive candidate model evaluation and adaptive sample selection in each iteration, enabling the graph model to incrementally incorporate informative anomalies and become more robust to previously unseen patterns.

This paper makes the following key contributions:

- This paper presents UnseenLog, the first framework in log anomaly detection specifically designed to address unseen anomaly patterns that do not appear in the training data.
- We introduce a MinMax strategy to select generalized augmented samples by jointly considering their similarity to and difference from known anomalous patterns. This strategy effectively

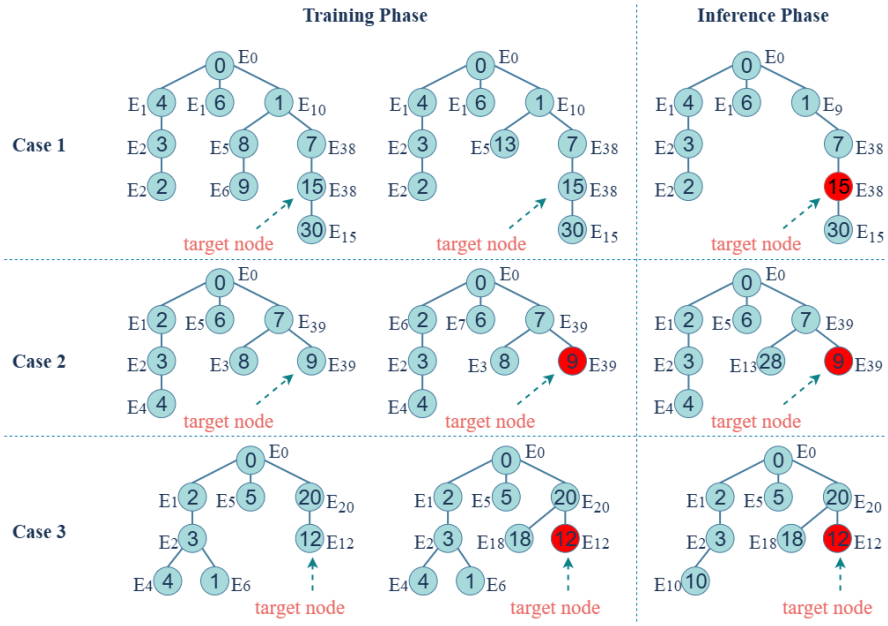


Fig. 1. Three cases. In each case, the left and middle graphs correspond to log segments that appear during the training phase, while the right graph corresponds to an anomalous log segment that appears during the inference phase. Each node represents a log message associated with a specific code file, and each edge denotes the invocation (call) relationship between code files. The number on each node indicates the ID of the associated code file, and E_i denotes the log event type. Red nodes indicate anomalous nodes, while blue nodes represent normal nodes.

balances anomalous patterns' reliability and novelty, improving the generalization capability of the anomaly detection model.

- We propose a Recurrent Iterative Selection and Enhancement (RISE) training scheme that progressively optimizes the anomaly detection model. By repeatedly filtering and incorporating anomalous samples through competitive evaluation and adaptive enhancement, RISE reduces pseudo-label noise and significantly enhances the model's ability to generalize to unseen anomalies.
- Extensive experiments demonstrate that UnseenLog outperforms state-of-the-art methods, achieving at least a 6% improvement in F1 score.

2 Preliminary

System logs are automatically generated records that capture various events and activities during software execution. Each log entry—also referred to as a log message—is produced by a specific code file. These messages serve as valuable information for developers, system administrators, and operators to monitor system behavior, diagnose issues, trace anomalies, and audit user activities.

Definition 2.1 (Log Entry). We model each recorded log entry as a 6-tuple

$$(cf, reqID, et, invcf, \delta, exc),$$

where:

- $cf \in \mathcal{CF}$ is the code file in which this log message is generated;

- $reqID$ represents the unique identifier corresponding to a specific request;
- $et \in \mathcal{ET}$ indicates the type of event, where the finite set $\mathcal{ET} = \{et \mid et$ is a natural-language string describing the system's runtime behavior$\}$;
- $invocf \in \mathcal{CF}$ refers to the upstream code file that invokes the component represented by cf;
- $\delta \in \mathcal{R}^+$ captures the execution time with the event.
- $exc \in \mathcal{EI}$ specifies the exception type, where the finite set $\mathcal{EI} = \{exc \mid exc$ is a natural-language string designed for describing system exceptions$\}$.

Although both the event type (et) and the exception type (exc) are natural-language strings, the sets $\mathcal{ET}$ and $\mathcal{EI}$ are finite because the functional scope and operational space of a software system are limited. These sets are an integral part of the system design, predefined by software developers to characterize the system's runtime behaviors. For instance, in an e-commerce system, users may place orders, make payments, or submit product reviews. These semantically diverse actions can be categorized into a finite set of event types, such as "Order Creation" or "Payment Processing". whereas runtime failures like "AuthenticationFailureException" or "SQLTimeoutException" are categorized into predefined exception types. This abstraction enables the system to uniformly record and manage in a structured manner, facilitating effective logging, analysis, and monitoring.

Concretely, an individual log message can be mapped to this 6-tuple representation. For example, a raw log message such as L3 in Fig. 2 can be represented as the 6-tuple, where cf = "Novel.UserService", $reqID$ = "5031f43971994", et = "return user id:* information", $invocf$ = "Novel.GlobalInterceptor", δ = "4ms", exc = "Java class doesn't match". Here, the log event "return user id:15745 information" can be parsed as "return user id:* information", with the specific user ID omitted. In this manner, all events related to user information retrieval can be categorized under the same event type.

Definition 2.2 (Log Graph). A log graph is defined as an attributed directed graph $G = (V, E)$, where V and E are specified as follows:

- V denotes the set of log messages and every node $v \in V$ is represented as a 4-tuple (cf, et, δ, exc);
- E represents the set of invocation relationships between the cf associated with the corresponding log messages. An edge from v_i to v_j (i.e., $(v_i, v_j) \in E$) indicates that the code file $v_{i.cf}$ invokes the code file $v_{j.cf}$.

A log graph G is a directed, tree-structured graph that represents program invocations within a single request. The graph contains a unique root node $v_{root} \in G$, which serves as the entry point of program execution. Let $deg(v)$ denote the degree of a node v in G. A node v is referred to as a leaf node, denoted v_{leaf}, if $deg(v) = 1$.

Definition 2.3 (Invocation Path). An invocation path is defined as the shortest directed path from the root node v_{root} to a leaf node v_{leaf}, denoted as:

$$v_{root} \longrightarrow \ldots \longrightarrow v_i \longrightarrow \ldots \longrightarrow v_{leaf}.$$

For notational simplicity, we use $inv(v)$ to represent the set of invocation paths in which the node v appears. As shown in the log graph in Fig. 2, node 3 belongs to two invocation paths: $root \longrightarrow 4 \longrightarrow 3 \longrightarrow 1$ and $root \longrightarrow 4 \longrightarrow 3 \longrightarrow 2$.

In our study, the task of log anomaly detection is formulated as a node-level binary classification problem within the log graph.

Definition 2.4 (Log Anomaly Detection). Given a log graph G, where each node is associated with a code file, the task of log anomaly detection is to predict binary class labels $y_v \in \{0, 1\}$ for all nodes $v \in V$. A node is *normal* ($y_v = 0$) if its corresponding code file executes as expected without

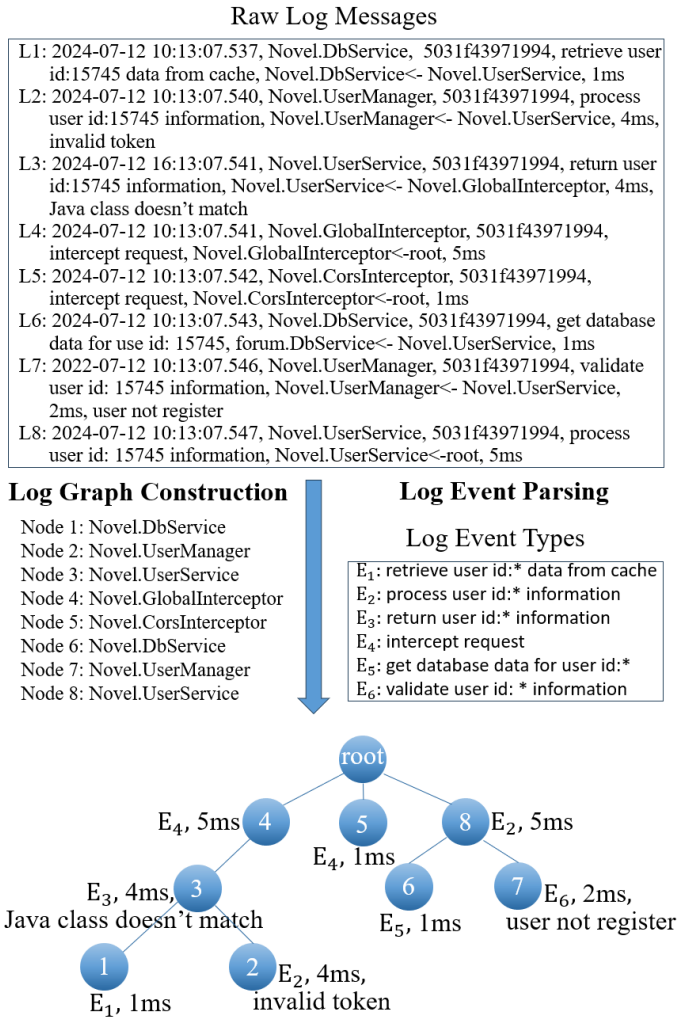


Fig. 2. **Log Graph Construction.** The raw log messages are first parsed into log event types, which are then used to construct the log graph. In this graph, each node represents a log message associated with the code file that generated it, and edges indicate invocation relationships between the code files corresponding to the log messages.

triggering any faulty behavior, and is *anomalous* ($y_v = 1$) if the code file executes a buggy code segment, leading to unexpected runtime behaviors.

In our study, a node's anomalous status depends on the behavior of its corresponding code file under a specific execution environment. In log graphs, an anomalous node represents the corresponding code file whose execution involves defective code segments, leading to unexpected behaviors recorded in the logs, such as altered invocation relationships. These anomalous behaviors deviate from the expected workflow, which may originate in various cases. Below, we present several examples in which anomalous behaviors can be observed in logs.

Example 2.1 (Data-induced Anomaly)**.** In a Database Management System (DBMS), a query optimizer may rely on a statistics catalog to estimate query costs. When the catalog becomes inconsistent due to metadata corruption, a robust optimizer should handle the issue gracefully. However, if the implementation fails to validate the data and continues with invalid statistics, runtime errors such as division-by-zero may occur, leading to missing invocations or unexpected termination in system logs.

Example 2.2 (High Concurrency-induced Anomaly)**.** Similarly, in a connection management module of a DBMS, which maintains a pool of active connections, latent synchronization defects (e.g., a missing lock or improper resource cleanup) may only be triggered under high concurrency, causing abnormal invocation patterns and prolonged response times in the logs.

In particular, certain scenarios, such as Example 2.2, occur infrequently during regular operation, making the corresponding anomalies rare in practice. As a result, these anomalies cannot always be anticipated or explicitly modeled in advance. Therefore, the ability to identify previously unseen anomalous behaviors is crucial for maintaining the reliability, stability, and overall performance of systems. This motivates us to study unseen anomaly detection from system logs. To formalize this concept, we define *unseen anomaly* and *anomaly shift* to describe the distributional differences of anomalous patterns across training and inference phases.

Definition 2.5 (Unseen Anomaly). Let $g : V \rightarrow \mathcal{S}$ be a structure encoder mapping log graph nodes v to latent representations $s = g(v) \in \mathcal{S}$, where $\mathcal{S} \subseteq \mathbb{R}^d$ denotes a d-dimensional continuous latent space. Let $S^{ano}_{train} \subset \mathcal{S}$ denote the set of anomalous latent representations observed in the training phase, and $p^{ano}_{train}(s)$ be the probability density function of S^{ano}_{train} defined on the latent space $\mathcal{S}$. A log graph node v' in the inference phase is with an unseen anomalous pattern $s' = g(v'), y_{v'} = 1$ if and only if

$$\mathbb{P}[p^{ano}_{train}(s) > p^{ano}_{train}(s'), s \in S^{ano}_{train}] > 1 - \alpha,$$

where α denotes the significance level in statistical analysis, which is typically 0.05.

Intuitively, this means that with a high confidence level (i.e., above 95%), s' does not fall in the high-density region of the anomalous patterns in the training set ($s \in S^{ano}_{train}$), suggesting the emergence of a novel anomalous pattern that was not present in the training data.

Definition 2.6 (Anomaly Shift). An *anomaly shift* occurs when there exists at least one *unseen anomaly* in inference.

3 Methodology

In this paper, we propose a novel framework, UnseenLog, designed to detect previously unseen anomalous code files from system logs.

3.1 Overall Framework

Fig. 3 illustrates the overall architecture of UnseenLog. It comprises two key components that work in tandem to enhance the model's generalization capability: (1) MinMax strategy; (2) Recurrent Iterative Selection and Enhancement.

- **MinMax Strategy for Sample Selection:** To improve the model's generalization to previously unseen anomalies, this component aims to expand the training set with diverse anomalous patterns, helping the model learn to adapt to new anomaly patterns. At the core of this process is our MinMax strategy, which serves as a **filtering and re-labeling** mechanism, ensuring that reliable and informative augmented samples are selected. The strategy selects high-quality anomalous samples from the augmented data by jointly considering their similarity to known

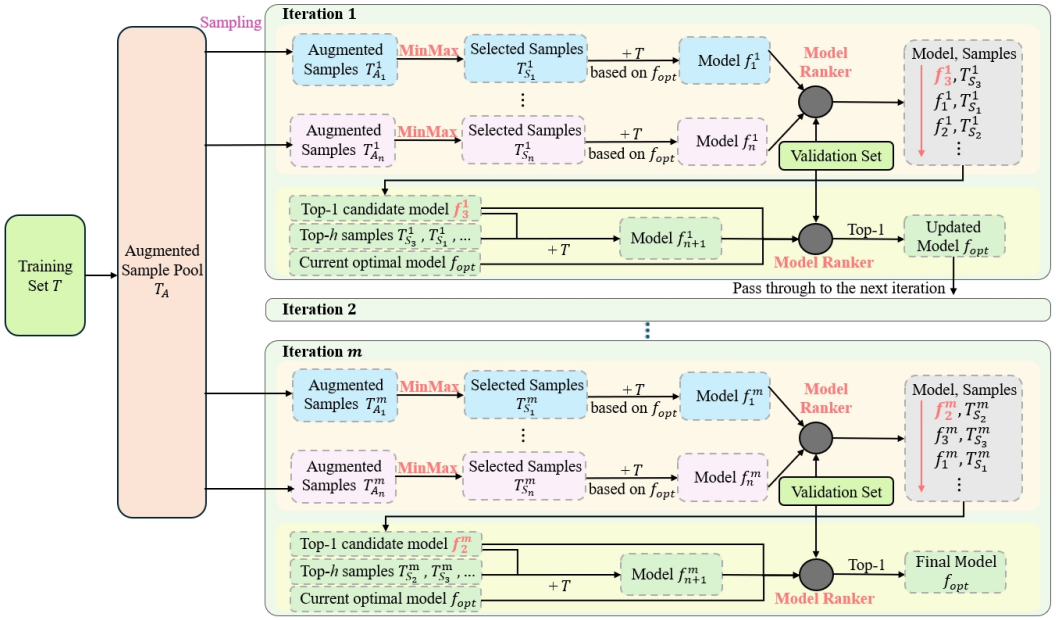


Fig. 3. The main framework of UnseenLog, which mainly consists of (1) MinMax strategy for augmented sample selection and (2) recurrent iterative selection and enhancement.

anomalies and their potential novelty. Specifically, the Min strategy prioritizes augmented samples that exhibit similarity to known anomalies, ensuring reliable anomaly characteristics, while the Max strategy enhances differences by selecting samples with maximally distinct contextual invocation environments, captured through invocation path representations of code files. This ensures augmented samples reflect both stable system behaviors and evolving execution contexts. In addition, unlike image data, graph data augmentation often modifies graph structures and node features, which can introduce label shifts and potentially mislead model training. In this context, MinMax also acts as a re-labeling mechanism to mitigate potential label shifts and reduce misleading signals during training. Through this combination of filtering and re-labeling, the MinMax strategy helps the model improve the robustness and generalization capacity of the anomaly detection.

- **Recurrent Iterative Selection and Enhancement (RISE):** To mitigate the adverse impact of noisy augmented samples, RISE introduces a recurrent training scheme that iteratively enhances the anomaly detection model. In each iteration, based on the MinMax strategy, RISE trains multiple candidate models and only the best-performing model, i.e., the one contributing most positively to generalization, is retained. In the next iteration, this model guides the selection and aggregation of more reliable pseudo-labeled samples. This progressive optimization process offers three key benefits. First, it effectively mitigates the noise introduced by inaccurate pseudo-labels. Second, it enables the framework to gradually incorporate informative and previously unseen anomalous patterns, helping the model adapt to this distribution shift. Third, by updating the model through fine-tuning, the framework incrementally enhances its capability to handle newly emerging anomalies, facilitating smoother adaptation to future runtime environments. As a result, the model becomes increasingly robust and better adapted to detect novel anomalies.

Given a training set $T = \{G_i\}_{i=1}^{N_G}$, where each node $v_j \in G_i$ is associated with a class label $y_{v_j} \in \{0, 1\}$, the training phase consists of two steps: pre-training and iterative training. During the pre-training stage, UnseenLog learns a graph anomaly detector f, which can use any GNN-based architecture g as its backbone, such as the Graph Transformer Neural Network (GTNN) [26]. During the iterative training stage, UnseenLog first constructs an unlabeled augmented sample pool T_A through data augmentation techniques based on T. The iterative training process of UnseenLog consists of repeated rounds of sampling on T_A, MinMax Strategy, candidate model fine-tuning, and model rank & pick.

In the $k\text{-}th$ iteration, it performs n rounds of random sampling from T_A to obtain n subsets of unlabeled augmented samples $\{T_{A_1}^k, \ldots, T_{A_n}^k\}$. The sampled subset $T_{A_r}^k$ $(1 \le r \le n)$ in the $r\text{-}th$ round is then processed using a MinMax strategy to select only a portion of the highest-quality nodes. These nodes are assigned anomalous pseudo-labels to form the selected sample set $T_{S_r}^k$. Next, $T_{S_r}^k$ is combined with the training set T, and used to fine-tune the current best model f_{opt}, producing a candidate model f_r^k. Later, the n candidate models $\{f_1^k, \ldots, f_n^k\}$ are evaluated on a validation set, and the one with the highest performance, denoted by f_{best}^k, is selected. Finally, the selected pseudo-labeled samples associated with the top-h candidate models are aggregated and used, along with the original training set T, to fine-tune f_{best}^k. After fine-tuning, a new model denoted as f_{n+1}^k is obtained. Thus far, we have obtained two models f_{best}^k and its fine-tuned version f_{n+1}^k. If any of them outperforms the current f_{opt}, we replace f_{opt} by the best of the two. In this way, UnseenLog never selects a model that underperforms the current best at each iteration. This process is repeated for m iterations, gradually improving the quality of the anomaly detector by leveraging diverse pseudo-labeled anomaly samples.

3.2 MinMax Strategy for Sample Selection

In this work, we aim to improve a model's ability to detect anomalous log patterns that deviate from those appeared during training. We refer to anomalies with such patterns as "unseen anomalies". Detecting these anomalies is particularly challenging because they are inherently out-of-distribution, requiring the model to generalize to anomalous patterns it has never encountered before. A common strategy is data augmentation, where perturbations or transformations are applied to the training data to generate synthetic samples. In our scenario, however, the augmented data does not necessarily cover the unseen anomaly patterns encountered during inference. Rather, the goal of augmentation is to help the model learn to adapt to new anomalous patterns. In practice, this introduces two main challenges:

First, augmented samples should be sufficiently different from the original data to increase pattern diversity, while still preserving a degree of consistency with the training distribution. This balance ensures that the model learns to generalize from existing knowledge without being overwhelmed by irrelevant or noisy variations. Striking a trade-off between novelty and similarity is thus a key difficulty in augmentation design and selection.

Second, labeling augmented graph nodes poses a unique challenge. In graph-based settings, data augmentation may involve changes to graph structure or node attributes. Such changes may lead to semantic drift or label ambiguity. Thus, augmented samples must be carefully assessed or re-labeled before being used for training; otherwise, they may introduce misleading signals and degrade model performance.

To address these challenges, our analysis reveals that logs exhibit a notable coexistence of both differences and similarities. On one hand, as software systems undergo continuous version updates and iterations, their technical implementations, user interfaces, and certain functional modules often experience varying degrees of adjustment, refactoring, or extension. These changes are

typically driven by evolving user requirements, advancements in the technological environment, and performance optimization needs. As a result, new system behaviors may emerge, contributing to differences in anomaly patterns. On the other hand, despite observable differences in external features and specific functional components, the core functional architecture and business logic of the system generally remain highly consistent across versions. This consistency reflects the system's stability in fulfilling essential business requirements. Consequently, even though anomalies may appear in different forms across versions, their underlying causes often share common or similar structural characteristics.

Motivated by these differences and similarities, we propose a MinMax strategy that automatically selects a subset of reliable yet diverse augmented anomalous graph node samples to improve the model's generalization. This strategy jointly considers the novelty and similarity of each sample, ensuring that selected instances are sufficiently distinct from the training data while still preserving invariant characteristics. As illustrated in Fig. 4, it demonstrates the core idea of the MinMax strategy. The samples selected by the Min strategy (represented by yellow nodes) exhibit a distribution similar to that of the training samples (represented by blue nodes). Based on this, the Max strategy further selects samples that maintain a certain degree of distributional differences from the training data, which are highlighted by red-circled nodes.

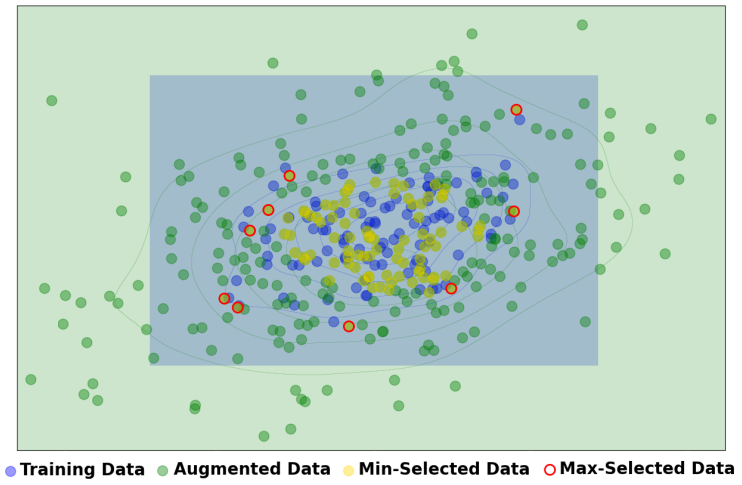


Fig. 4. Overview of MinMax Selection Strategy

In each *r-th* round of the *k-th* iteration in the training process, UnseenLog samples an unlabeled subset of augmented graph nodes $T^k_{A_r} \subset T_A$. To extract a set of anomalous candidate samples that reflect both differences and similarities, we apply the proposed MinMax strategy, which consists of two steps:

3.2.1 Min Strategy: Similarity-based Anomaly Sample Selection. Our first aim is to identify augmented nodes that exhibit similarity to known anomalies in the training set T. Specifically, we use the current model f_{opt} to assign an anomaly score to each node $v \in T^k_{A_r}$. Nodes with scores exceeding a predefined threshold t are retained to form the intermediate candidate sample set:

$$T^k_{Min_r} = \{v \in T^k_{A_r} \mid f_{opt}(v) > t\} \tag{1}$$

Our Min strategy ensures that the selected augmented samples retain the overall characteristics of the ground truth samples, thereby guaranteeing their reliability.

3.2.2 Max Strategy: Novelty-based Anomaly Sample Selection. To further ensure the novelty among anomaly candidate nodes, we introduce the Max strategy, which is inspired by domain-specific insights from software. Specifically, when a log message is produced, it typically corresponds to the execution of a specific function or module. However, its meaning and significance are not confined to the source code file. Rather, they depend on which component triggered it, the invocation (call) path under which it occurred, and its position within the overall execution flow of the program. By capturing the invocation relationships among code files involved in log generation, it is possible to reconstruct the execution trace and thus understand the contextual environment in which each log message occurs. In real-world systems, anomalies rarely follow a single pattern. Instead, they often manifest differently depending on the execution context. In log anomaly detection, simulating such contextual variations enables the generation of more diverse and realistic anomaly samples, thereby improving the generalization capability of detection models.

Building on this insight, we aim to extend the Min strategy by further selecting anomaly candidate samples with maximally diverse contextual call environments.

For each $v \in T^k_{\text{Min}_r}$, the *invocation path representation* is computed via the GNN encoder as $g(\text{inv}(v))$. If v participates in multiple invocation paths, $g(\text{inv}(v))$ is computed by mean-pooling the representations of all nodes contained in v's paths. Then, we measure its minimum Euclidean distance to nodes in the training set, based on their invocation representations.

Let $A = \{u \in G_i \mid G_i \in T\}$ be the set of nodes with ground truth label. The difference score for each node $v \in T^k_{Min_r}$ is computed as:

$$D(v) = \min_{u \in A} \|g(\text{inv}(v)) - g(\text{inv}(u))\|_2 \tag{2}$$

We then select the top-a nodes from $T^k_{\text{Min}_r}$ with the largest difference scores $D(v)$ to form the final selected set $T^k_{S_r}$. These nodes are assigned anomalous pseudo labels based on the current f_{opt} and used to fine-tune the model during iterative training.

3.3 Recurrent Iterative Selection and Enhancement

The MinMax strategy introduced in Section 3.2 aims to expand the training set by incorporating novel anomalous patterns, thereby enhancing the model's ability to detect anomalies. However, the effectiveness of this strategy critically depends on the predictive quality of the current graph model. If the graph model's performance is inadequate, the pseudo-labels generated may be inaccurate or noisy, which can mislead the learning process and degrade overall performance. To address this challenge, we propose a training scheme called Recurrent Iterative Selection and Enhancement (RISE). This scheme employs a three-step recurrent iteration that combines candidate graph model generation, candidate graph model competition, and adaptive enhancement. In each iteration, RISE mines high-confidence anomalous samples from augmented data, selects the best-performing candidate graph model based on validation metrics, and aggregates reliable pseudo-labeled samples to further enhance the model. Through this recurrent process, RISE continuously filters and incorporates reliable, novel, and previously unseen anomalous patterns, enabling the graph model to progressively improve and enhance its generalization capability.

Specifically, in each k-th iteration, the iterative training consists of the following three steps:

Step 1: Candidate Graph Model Generation. Let n be the number of rounds in each iteration, indexed by $r = 1, \ldots, n$. For each round r, the procedure is as follows:

(1). A unlabeled augmented sample set $T^k_{A_r} \subset T_A$ is sampled from the augmented pool T_A.

(2). A MinMax strategy (Section 3.2) is applied to $T^k_{A_r}$ to select a subset of samples. These nodes are assigned anomalous pseudo-labels, forming the selected sample set $T^k_{S_r}$.

(3). The current f_{opt} is fine-tuned on the union of the original training set T and the selected sample set $T_{S_r}^k$, resulting in a candidate f_r^k.

Step 2: Candidate Graph Model Competition. All candidates $\{f_1^k, ..., f_n^k\}$ are evaluated on a validation set. The model with the best performance is selected, denoted by f_{best}^k, as a candidate model for further evaluation in Step 3.

Step 3: Adaptive Enhancement. The selected sample sets associated with the top-performing h candidate models (ranked by their validation performance in Step 2) are aggregated to form an enhanced pseudo-labeled dataset. This aggregated set is then combined with the original training set T to further fine-tune the best model found at the k-th iteration, i.e., f_{best}^k. Denote the resulting updated model as f_{n+1}^k. From f_{best}^k, f_{n+1}^k, and the current f_{opt}, we select the model achieving the best performance on the validation set as the new optimal model f_{opt} for the next iteration. Our motivation for this design is that higher-performing models are likely trained on more informative and reliable data. Aggregating their sample sets thus yields a higher-quality pseudo-labeled dataset that better supports model enhancement.

This iterative process enables UnseenLog to gradually update by selectively incorporating informative anomalous patterns discovered through augmentation, thereby achieving enhanced generalization to novel anomalies.

3.4 Objective Function

The loss function of UnseenLog contains two components, supervised loss $\mathcal{L}_{CrsEnt}^{ture}$ and pseudo-labeled loss$\mathcal{L}_{CrsEnt}^{pseudo}$. For supervised loss, it is the standard cross-entropy loss computed on the original training samples with ground-truth labels. To compute the pseudo-label loss, we treat the model-generated pseudo-labels for the augmented data as targets and calculate the corresponding cross-entropy loss.

Formally, the total loss $\mathcal{L}$ is defined as:

$$\mathcal{L} = \alpha \mathcal{L}_{CrsEnt}^{ture} + (1 - \alpha)\mathcal{L}_{CrsEnt}^{pseudo}, \tag{3}$$

$$\mathcal{L}_{CrsEnt}^{ture} = \frac{1}{N_v} \sum_{i=1}^{N_v} CrsEnt(\mathsf{UnseenLog}(x_{v_i}), y_{v_i}), \tag{4}$$

$$\mathcal{L}_{CrsEnt}^{pseudo} = \frac{1}{|T_S|} \sum_{j=1}^{|T_S|} CrsEnt(\mathsf{UnseenLog}(x_{v_j}), \tilde{y}_{v_j}), \tag{5}$$

where N_v is the number of nodes in the training set, T_S refers to selected pseudo-labeled samples, $\mathsf{UnseenLog}(\cdot)$ represents our proposed framework as a function, $\tilde{y}$ denotes pseudo label.

The pseudo-code for UnseenLog during the training and inference phases is presented in Alg. 1 and Alg. 2, respectively. As shown in Lines 1–2, UnseenLog first pre-trains a GNN model and constructs an augmented sample pool from the original training set. In each training iteration, augmented samples are selected and labeled using the MinMax strategy (Line 6), and used to fine-tune the current model (Line 7), producing multiple candidate models. The model with the best validation performance is selected as the new model (Line 9), which is further enhanced using data associated with the top-performing candidates (Line 10). At inference time, the trained model f_{opt} is applied to predict the label.

3.5 Complexity Analysis

We analyze the computational complexity of UnseenLog by decomposing its training phase into three key stages: pre-training, data augmentation, and recurrent iterative training. Let $T = \{G_i\}_{i=1}^{N_G}$

Algorithm 1 Overview of UnseenLog for training phase

Require: Training set $T = \{G_i\}_{i=1}^{N_G}$ with nodes $\{(v_j, y_{v_j}) \mid v_j \in G_i\}$, validation set T_V, iteration m, round n in each iteration
Ensure: The trained UnseenLog for anomaly detection
1: Pre-train the model f_{opt} with training set
2: Augment training set based on T to get augmentation pool T_A
3: **for** Training iteration $k = 1, 2, 3, \ldots, m$ **do**
4: **for** Round $r = 1, \ldots, n$ in each iteration **do**
5: Sampling augmented data $T_{A_r}^k$ from the pool T_A
6: Selecting and labeling $T_{S_r}^k$ by f_{opt} with MinMax strategy
7: Fine-tuning f_{opt} with training set T and pseudo-labeled dataset $T_{S_r}^k$ to get candidate model f_r^k
8: **end for**
9: Among candidate models, selecting candidate model with best performance on T_V as f_{best}^k
10: Selecting pseudo-labeled samples associated with top-performing h models plus T to fine-tune f_{best}^k as f_{n+1}^k
11: Selecting best model among f_{opt}, f_{n+1}^k and f_{best}^k as f_{opt}
12: **end for**

Algorithm 2 Overview of UnseenLog for inference phase

Require: A log graph G with nodes $v_j \in G$, well-trained model f_{opt}
Ensure: Predicted label $\hat{y}_{v_j}$ for each node $v_j \in G$
1: Using f_{opt} outputs predicted label $\hat{y}_{v_j} = f_{opt}(v_j)$ for each $v_j \in G$

denote the training set with a total of $|T|$ nodes, and let T_A denote the augmented sample pool containing $|T_A|$ nodes. We denote the node feature dimension as d, the number of GNN (we use GTNN here) layers as L, and the average number of nodes per graph as n_v.

We also define the L-layer complexity of a GTNN as:

$$C_{GT} = L \cdot (n_v^2 \cdot d + n_v \cdot d^2).$$

- **Pre-training Phase.** In this phase, a model f is trained on the original training set T. Assuming each graph has n_v nodes and uses full attention, the total complexity of pre-training is:

$$O(|T| \cdot C_{GT}).$$

- **Augmentation Phase.** The augmentation phase generates the pool T_A using three transformation strategies: edge addition, edge deletion, and node feature perturbation with Gaussian noise. For each graph, edge modification (addition or deletion) requires $O(n_v^2)$ operations, and node feature perturbation costs $O(n_v \cdot d)$. Therefore, generating $|T_A|$ nodes through augmentation incurs:

$$O(|T_A| \cdot (n_v + d)).$$

- **Iterative Training Phase.** This phase involves m outer iterations, each performing n inner rounds and one aggregation. In each round r of iteration k, the following steps occur:
  - *Sampling:* A subset $T_{A_r}^k$ is randomly sampled from T_A, costing $O(1)$.
  - *MinMax Strategy:* Each node in $T_{A_r}^k$ is scored by the current model f_{opt}. This leads to $O(|T_{A_r}^k| \cdot C_{GT})$. For the selected subset $T_{\text{Min}_r}^k$, the Max strategy computes the minimum

Euclidean distance to every node in the ground-truth node set A, giving:

$$O(|T^k_{A_r}| \cdot C_{GT} + |T^k_{\text{Min}_r}| \cdot |A| \cdot d).$$

– *Fine-tuning:* The f_{opt} is fine-tuned on $T \cup T^k_{S_r}$, yielding:

$$O((|T| + |T^k_{S_r}|) \cdot C_{GT}).$$

– *Validation:* Evaluation on the validation set T_V with size $|T_V|$ costs:

$$O(|T_V| \cdot C_{GT}).$$

After obtaining n candidate models $\{f^k_r\}^n_{r=1}$, the best performing one is retained. Additionally, pseudo-labeled samples from the top-h candidates (total size $h \cdot |T^k_{S_r}|$) are used for further fine-tuning:

$$O((|T| + h \cdot |T^k_{S_r}|) \cdot C_{GT}).$$

Overall Complexity for Training. Across all m iterations, the simplified training complexity of UnseenLog is:

$$O\left(m \cdot n \cdot |T| \cdot (|T^k_{\text{Min}_r}| \cdot n_v \cdot d + C_{GT})\right) \tag{6}$$

Overall Complexity for Inference. During inference phase, UnseenLog uses the trained model to perform anomaly prediction over incoming log graphs. Unlike the training phase, the inference does not involve data augmentation, MinMax strategy, or iterative pseudo-labeling. The complexity therefore depends solely on a forward pass through the model.

Giving a log graph for anomaly detection, the total inference complexity is:

$$O(C_{GT}) \tag{7}$$

Analysis. In practice, m and n are small constants (e.g., 1–3), and $|T^k_{\text{Min}_r}|$ is relatively small comparing to $|T|$. Therefore, the scalability of UnseenLog is primarily governed by the size and number of training graphs and the complexity of GNN backbone C_{GT}. In contrast, during the inference phase, efficiency is significantly improved, as it relies solely on a well-trained model, resulting in very fast inference.

4 Experiment

4.1 Experimental Setup

4.1.1 Benchmark. We evaluate UnseenLog on 3 real-world log datasets: Forum, Halo, and Novel [17, 28]. They are collected from 3 different open-source systems, respectively. Forum contains 6,785,624 log messages, with 13.19% labeled as anomalous. Halo consists of 37,528,612 log messages, among which 10.58% are anomalous. Novel includes 14,492,161 log messages, with 6.4% identified as anomalous.

Table 1. The statistics of our benchmarks.

	|G|	|V|	|E|	# Anomalous Graphs	% Anomalous Graphs	# Anomalous Nodes	% Anomalous Nodes
Forum	214,233	6,455,450	6,241,217	97,825	45.66%	854,988	13.24%
Halo	100,000	1,563,462	1,687,186	45,181	45.18%	54,448	3.48%
Novel	149,499	1,926,609	1,777,110	68,466	45.80%	68,471	3.80%

4.1.2 Data Preprocessing. Following by the previous study [28], we primarily perform log parsing and log grouping. Log parsing aims to remove irrelevant information and extract event types, converting log messages with similar structures into standardized event templates. Log grouping is to divide consecutive log messages into distinct segments, each representing an independent collection of log messages. In our experiments, we adopt the trace window approach for log grouping, whereby log messages sharing the same trace ID are grouped together, and a corresponding log graph is constructed for each log group. After preprocessing, the statistical information of the three datasets is presented in Table 1.

4.1.3 Baselines. We compare UnseenLog with 5 categories of baseline methods, including: (1) General GNNs: GAT [29], GCN [9], and GTNN [26]; (2) Substructure-based GNNs: GNNAK [44] and SAGNN [40]; (3) Log anomaly detection methods: LogGD [34], TP-GNN [17], LogFormer [5], and SLAD [28], DeepLog [10], LogRobust [43]; (4) Graph-based anomaly detection methods: BWGNN [27] and GHRN [4]; Graph OOD detection methods: LiSA [39], OOD-GMixup [20], and OpenIMA [31].

4.1.4 Experimental Settings. All experiments are conducted on a server running Red Hat Enterprise Linux 8.10. The machine is equipped with an NVIDIA A100 GPU (80GB VRAM), 256-core CPU, and 1TB of system memory.

For three real-world datasets, the node feature dimension is 247, 484, and 499 for Forum, Halo, and Novel, respectively. Then, we apply dataset-specific configurations for sampling, GNN architecture, and MinMax selection. For data augmentation, we randomly add or remove 30% of the edges and apply Gaussian noise (mean = 0, standard deviation = 0.3) to node features. The rationale is that normal and anomalous samples may exhibit distinct invocation behaviors. Therefore, the goal of augmentation is to introduce perturbations that simulate potential anomalies. Making such modifications to invocation semantics can provide diverse anomalous variants, thereby improving the model's robustness and generalization during training. The total augmentation sampling pool is set to 60× the number of training graphs, from which 6× the training size samples are drawn in each augmentation process. For the GNN encoder, we adopt a 2-layer architecture with a hidden size of 512 on *Forum*, a single-layer GNN with hidden size 512 on *Halo*, and a 2-layer GNN with hidden size 128 on *Novel*. For MinMax-based augmentation selection, we use different thresholds across datasets: on *Forum*, the confidence lower bound and top-a ratio are both set to 0.8; on *Halo*, they are set to 0.8 and 0.6, respectively; and on *Novel*, we set them to 0.6 and 0.8, respectively. For the top-h parameter, we set it to 2 to allow proper augmented data to participate in model fine-tuning during each iteration. For RISE, we set the number of training iterations and rounds to 3, respectively. All experiments are repeated five times, and we report the average performance along with the standard deviations.

4.1.5 Dataset Partitioning. To simulate unseen anomalies, we randomly select a group of code files and place all log graphs containing anomalous node labels corresponding to these selected code files into the test set. In this way, the anomalous behaviors of these code files are not observed during training. For each dataset, we repeat this random partitioning five times. On average, the proportion of unseen anomalous code files to the total number of code files is 7.37% for Forum, 6.25% for Halo, and 7.57% for Novel. This partitioning represents an intentionally strict setting, preventing potential leakage and ensuring unseen anomalies in the test set. For those code files that are normal during training but anomalous during inference, the model has only observed their normal patterns during training. In the test set, these files may appear in both normal and anomalous states. Under such a setting, it becomes difficult for the model to rely solely on code-file

Table 2. The experiment results with 16 baselines and our UnseenLog with different GNN backbones. For each metric, the best performance on each metric is highlighted in bold font, and the second-best (excluding ours) is underlined.

	Forum				Halo				Novel			
	F1	Recall	Precision	PR-AUC	F1	Recall	Precision	PR-AUC	F1	Recall	Precision	PR-AUC
GAT	0.31±0.18	0.24±0.16	0.57±0.21	0.30±0.14	0.34±0.09	0.26±0.09	0.55±0.17	0.18±0.05	0.32±0.14	0.31±0.15	0.49±0.20	0.33±0.14
GCN	0.32±0.19	0.25±0.19	0.57±0.16	0.32±0.15	0.37±0.02	0.25±0.01	0.70±0.11	0.28±0.03	0.33±0.21	0.26±0.16	0.50±0.23	0.32±0.05
GTNN	0.45±0.10	0.36±0.13	0.73±0.18	0.49±0.12	0.30±0.13	0.21±0.10	0.60±0.22	0.21±0.10	0.36±0.15	0.28±0.15	0.69±0.11	0.44±0.07
GNNAK	0.48±0.10	0.47±0.11	0.56±0.19	0.45±0.14	0.40±0.15	0.30±0. 15	0.67±0.02	0.32±0.13	0.53±0.10	0.65±0.09	0.46±0.12	0.37±0.13
SAGNN	0.45±0.10	0.37±0.11	0.63±0.12	0.43±0.15	0.20±0.04	0.16±0.00	0.31±0.16	0.14±0.04	0.63±0.04	0.63±0.11	0.67±0.10	0.63±0.05
LogGD	0.44±0.14	0.37±0.18	0.60±0.13	0.41±0.17	0.41±0.01	0.29±0.03	0.77±0.11	0.33±0.02	0.58±0.10	0.67±0.19	0.54±0.08	0.60±0.10
LogFormer	0.28±0.18	0.26±0.23	0.37±0.10	0.31±0.10	0.15±0.01	0.18±0.05	0.14±0.03	0.08±0.01	0.58±0.06	0.56±0.13	0.66±0.17	0.63±0.02
TP-GNN	0.47±0.06	0.42±0.18	0.63±0.12	0.38±0.14	0.39±0.07	0.33±0.06	0.56±0.25	0.30±0.07	0.56±0.12	0.50±0.18	0.66±0.07	0.40±0.10
SLAD	0.40±0.15	0.36±0.17	0.50±0.19	0.42±0.18	0.39±0.05	0.35±0.07	0.55±0.22	0.24±0.10	0.65±0.09	0.69±0.07	0.62±0.12	0.62±0.11
DeepLog	0.34±0.04	0.35±0.09	0.39±0.07	0.34±0.02	0.32±0.05	0.22±0.03	0.56±0.12	0.33±0.04	0.49±0.04	0.43±0.04	0.56±0.05	0.49±0.05
LogRobust	0.42±0.05	0.40±0.06	0.47±0.09	0.37±0.05	0.40±0.03	0.39±0.06	0.45±0.14	0.38±0.03	0.52±0.04	0.48±0.03	0.57±0.06	0.52±0.04
BWGNN	0.34±0.15	0.26±0.16	0.60±0.18	0.39±0.18	0.37±0.04	0.33±0.07	0.50±0.19	0.28±0.08	0.52±0.08	0.54±0.13	0.57±0.14	0.56±0.09
GHRN	0.44±0.13	0.34±0.14	0.69±0.06	0.50±0.06	0.39±0.02	0.27±0.02	0.70±0.04	0.44±0.11	0.43±0.13	0.48±0.25	0.49±0.11	0.49±0.09
LiSA	0.39±0.05	0.29±0.04	0.63±0.12	0.33±0.10	0.24±0.00	0.18±0.02	0.37±0.06	0.16±0.01	0.52±0.08	0.66±0.11	0.45±0.09	0.32±0.13
OOD-GMixup	0.49±0.12	0.36±0.11	**0.81±0.09**	–	0.50±0.13	0.38±0.11	0.74±0.21	–	0.44±0.19	0.35±0.17	0.69±0.23	–
OpenIMA	0.55±0.06	0.71±0.10	0.47±0.08	–	0.57±0.02	0.64±0.14	0.54±0.06	–	0.56±0.04	0.70±0.05	0.48±0.05	–
UnseenLog(GCN)	0.65±0.05	0.64±0.05	0.67±0.06	0.66±0.07	0.71±0.08	0.76±0.01	0.68±0.13	0.68±0.08	0.62±0.05	0.61±0.06	0.63±0.03	0.60±0.04
UnseenLog(GAT)	0.68±0.03	0.65±0.02	0.72±0.04	0.66±0.05	0.75±0.09	0.73±0.08	0.77±0.09	0.73±0.08	0.66±0.01	0.70±0.05	0.63±0.01	0.62±0.02
UnseenLog(GTNN)	**0.73±0.01**	**0.73±0.07**	0.74±0.07	**0.71±0.09**	**0.84±0.11**	**0.90±0.06**	**0.79±0.17**	**0.79±0.15**	**0.71±0.03**	**0.75±0.11**	**0.70±0.09**	**0.67±0.10**

embeddings to make accurate predictions. Therefore, this design ensures that the evaluation truly reflects the model's ability to generalize beyond memorized code-file embeddings.

4.2 Overall Comparison

Table 2 presents the overall performance comparison between our UnseenLog and 16 representative baselines across three real-world log datasets: Forum, Halo, and Novel. We evaluate all methods using four metrics: F1 score, Recall, Precision, and PR-AUC. The best performance on each metric is highlighted in bold. These results collectively demonstrate that UnseenLog achieves remarkable effectiveness and efficiency, especially when GTNN serves as the backbone, as its stronger graph representation capability compared to GCN and GAT leads to better overall performance within our framework. Overall, the results confirm that our framework achieves consistent improvements across different GNN backbones, which can be attributed to its ability to detect unseen anomalies. Notably, the F1 score increased by at least 6% for our best model. Among the baselines, OpenIMA, as an OOD approach, emerges as a competitive approach. However, it relies on a clustering-based strategy that implicitly assumes each node belongs to a specific category. This assumption may not hold for anomalous nodes, which are often sparsely distributed and difficult to group. As a result, such methods tend to overlook this nature of anomalous nodes, potentially leading to degraded detection performance. General GNNs, such as GAT, GCN, and GTNN, show limited anomaly detection performance because these models are not specifically designed for anomaly detection. Their F1 scores mostly stay below 0.45 with relatively large standard deviations. Substructure-based approaches, GNNAK and SAGNN, achieve moderate results in most cases. This can be attributed to their tendency to emphasize changes in the global substructures of the graph, rather than capturing anomaly patterns from the perspective of individual nodes. So, their performance is limited. For log anomaly detection approaches, although SLAD exhibited relatively good performance on Novel, its overall results across all datasets are unstable. This can be attributed to their heavy reliance on the distribution of the training data, which limits their ability to generalize to unseen anomaly patterns. In addition, LogRobust is designed to leverage language models for learning the semantics of unseen log events. However, it primarily addresses textual novelty at the event level and thus cannot capture the unseen anomalies focused in our study. Graph-based anomaly detection methods, such as BWGNN and GHRN, address anomaly detection by analyzing the graph spectral energy

Table 3. Ablation results of different variants of UnseenLog. "GNN Encoder only" denotes only using GNNs for anomaly detection; "UnseenLog w/o MinMax" and "UnseenLog w/o Max" represent the removal of the MinMax strategy and the Max strategy, respectively; "UnseenLog w/o top-h" denotes the removal of module of top-h in RISE.

	Forum				Halo				Novel			
	F1	Recall	Precision	PR-AUC	F1	Recall	Precision	PR-AUC	F1	Recall	Precision	PR-AUC
GNN Encoder only	0.45±0.09	0.36±0.13	0.73±0.18	0.49±0.12	0.30±0.13	0.21±0.10	0.60±0.22	0.21±0.10	0.36±0.15	0.28±0.15	0.69±0.11	0.44±0.07
UnseenLog w/o MinMax	0.47±0.10	0.36±0.09	0.72±0.20	0.47±0.15	0.38±0.15	0.27±0.12	0.74±0.18	0.37±0.20	0.52±0.06	0.52±0.14	0.57±0.11	0.50±0.04
UnseenLog w/o Max	0.72±0.06	0.69±0.08	0.76±0.07	0.67±0.07	0.80±0.14	0.76±0.16	0.85±0.11	0.75±0.19	0.68±0.03	0.71±0.10	0.68±0.11	0.62±0.03
UnseenLog w/o top-h	0.68±0.05	0.67±0.04	0.69±0.06	0.68±0.04	0.77±0.09	0.78±0.07	0.77±0.12	0.75±0.07	0.66±0.02	0.71±0.03	0.63±0.07	0.62±0.02
UnseenLog(full)	**0.73±0.01**	**0.73±0.07**	**0.74±0.07**	**0.71±0.09**	**0.84±0.11**	**0.90±0.06**	**0.79±0.17**	**0.79±0.15**	**0.71±0.03**	**0.75±0.11**	**0.70±0.09**	**0.67±0.10**

distribution. These methods focus on capturing global information but fail to account for unseen anomaly patterns.

4.3 Ablation Study

To evaluate the contribution of each component in UnseenLog, we conduct ablation studies by progressively removing or modifying key modules. The corresponding results are presented in Table 3. The ablations cover the following variants: (1) GNN Encoder only, which uses only the GNN encoder followed by an MLP with Softmax for anomaly detection; (2) UnseenLog w/o MinMax, which retains the GNN encoder and data augmentation but removes the MinMax strategy from the iterative training; (3) UnseenLog w/o Max, which keeps the GNN encoder and data augmentation while excluding only the Max component of the MinMax strategy in the iterative training; and (4) UnseenLog w/o top-h, where the framework performs without the module of top-h. The ablation results indicate that each component of UnseenLog plays a vital role in achieving strong overall performance. The MinMax strategy helps retain reliable and diverse augmented anomalous samples, while the module top-h helps model fine-tuning with proper augmented data to enhance capabilities for unseen anomaly detection.

4.4 Performance on Seen and Unseen Anomalies

In anomaly detection tasks, the majority of samples are typically normal. However, the objective of anomaly detection is not merely to correctly classify the majority class, but more importantly, to accurately identify the minority class—anomalous samples. These anomalies often indicate system failures, security breaches, or other high-risk events, and thus have a critical impact on system stability and security. Motivated by this, we further evaluate the performance of UnseenLog in detecting both seen (such as Case 3 in Fig. 1) and unseen (such as Cases 1 and 2 in Fig. 1) anomalies, in order to assess its generalization ability. Table 4 summarizes the anomaly distribution across the training and test sets. In detail, "# Ano." denotes the number of anomalous samples, "# Nor." is the number of normal samples, "# Useen Ano." represents the number of anomalous samples that are unseen during the training phase. Among the anomalous samples in the test set, a substantial proportion consists of unseen anomalies, ranging from 57.86% to 84.90%. Table 5 compares the performance of our framework UnseenLog with the most competitive baseline, OpenIMA, on both seen and unseen anomalies, where TP refers to the number of anomalous samples that are correctly predicted as anomalies. Each sample in our datasets is associated with one code file. TP (Unseen)/(Seen) refers to the number of anomalous code files unseen/seen during training that are correctly identified as anomalies. The results highlight the effectiveness and generalization of our approach, particularly when encountering previously unseen anomalies.

Table 4. The statistics on seen and unseen anomalies. "# Ano./Nor." denotes the ratio of anomalous to normal samples; "# Unseen Ano./# Ano." is the ratio of Unseen Anomalies to all anomalies.

	Training Set	Test Set	
	# Ano./# Nor.	# Unseen Ano./# Ano.	# Ano./# Nor.
Forum	6.65%	57.86%	17.21%
Halo	3.33%	75.99%	3.86%
Novel	4.08%	84.90%	6.05%

Table 5. The detection performance on seen and unseen anomalies. "TP/# Ano." is the ratio of true positive on all anomalies; "TP (Seen)/# Seen Ano." refer to the ratio of true positive for seen anomalies on all seen anomalies; "TP (Unseen)/# Unseen Ano." refer to the ratio of true positive for unseen anomalies on all unseen anomalies. TP refers to the number of anomalous samples correctly predicted as anomalies. Each sample in our datasets corresponds to a single code file. TP (Unseen)/(Seen) denotes the number of unseen/seen anomalous code files during training that are correctly identified as anomalies.

	UnseenLog			OpenIMA		
	TP/# Ano.	TP (Seen)/# Seen Ano.	TP (Unseen)/# Unseen Ano.	TP/# Ano.	TP (Seen)/# Seen Ano.	TP (Unseen)/# Unseen Ano.
Forum	73.02%	82.18%	69.65%	70.59%	76.12%	51.23%
Halo	90.46%	92.73%	80.49%	63.63%	70.12%	41.25%
Novel	74.75%	75.29%	70.62%	70.19%	72.52%	51.11%

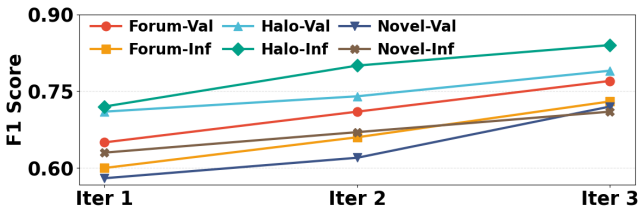


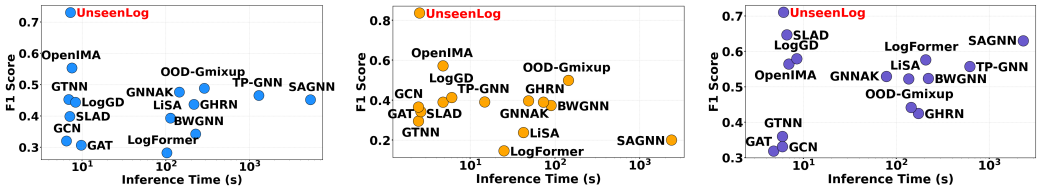
Fig. 5. F1 Trends on Validation (Val) and Inference (Inf) with Increasing Iterations (Iter)

4.5 Performance over Iterations

To demonstrate how RISE enhances model performance and stability, we report the model's F1 score on both validation and test sets across iterations. As shown in Fig. 5, the performance of UnseenLog consistently improves after each iteration on both validation and test sets, indicating the effectiveness of RISE in enhancing model robustness and adaptation to unseen anomalies.

4.6 Inference Efficiency and Efficacy Analysis

We further analyze the trade-off between inference efficiency and detection efficacy. The results for all methods are presented in Fig. 6. In this figure, methods that are closer to the top-left corner achieve a better balance between inference time and predictive performance. Across all datasets, UnseenLog consistently occupies the top-left position, indicating its superior trade-off between efficiency and effectiveness. Among the compared baselines, general GNNs (GAT, GTNN, and GCN), log anomaly detection methods (SLAD and LogGD), and the OOD method OpenIMA generally fall into the first tier in terms of inference efficiency, with relatively low inference times. However, when considering the F1 score, none of these methods surpass UnseenLog. In contrast, the remaining baseline methods exhibit both lower inference efficiency and suboptimal detection performance.



(a) Forum (b) Halo (c) Novel

Fig. 6. Inference Efficiency Comparison. For each dataset, a method appearing closer to the top-left corner reflects superior performance with respect to both inference efficiency and predictive effectiveness.

Overall, UnseenLog demonstrates strong performance in both inference efficiency and detection accuracy.

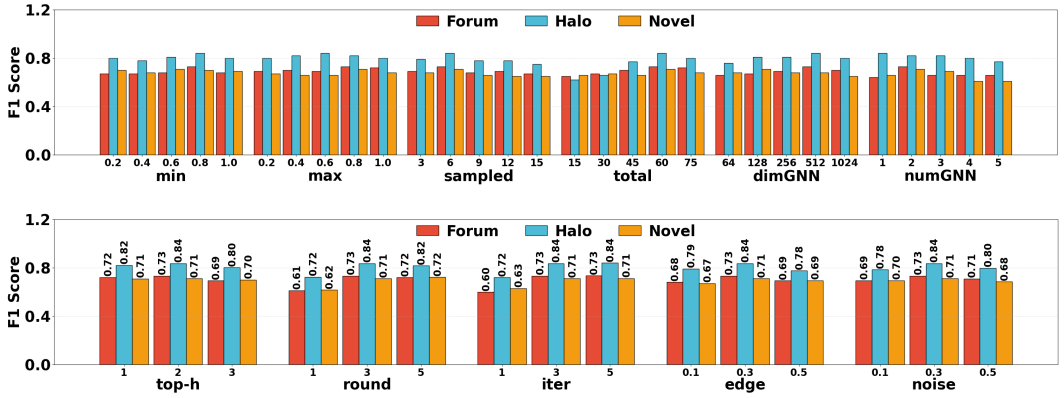


Fig. 7. Impact of Hyperparameters. "min"/"max": threshold t/top-a ratio for MinMax, "sampled"/"total": sampled/total magnitude of training graphs for data augmentation, "dimGNN"/"numGNN": dimension/number of GNN layers, "top-h": aggregated sample sets of the top-performing h candidate models, "round"/"iter": number n/m of rounds/iterations in RISE, "edge": perturbations with different ratios of edges, "noise": perturbations with varying strength of Gaussian noise on features.

4.7 Impact of Hyperparameters

We systematically investigate the impact of various hyperparameters on the performance of UnseenLog. To evaluate the individual effect of each hyperparameter, we adopt a controlled experimental setup in which only one hyperparameter is varied at a time while keeping others fixed (see Section 4.1.4). Each experiment is repeated five times to ensure stability. The performance trends of UnseenLog under different hyperparameter settings are illustrated in Fig. 7.

For the threshold t and top-a ratio used in MinMax, the best performance is achieved when the Min threshold t is set to 0.8 for Halo and Forum, 0.6 for Novel, and the Max threshold top-a is set to 0.6 for Halo, 0.8 for Forum and Novel. UnseenLog shows stable performance with respect to MinMax thresholds. For the sampled/total magnitude of training graphs for data augmentation, the performance first increases and then decreases. Small magnitudes provide insufficient diversity in augmented graphs, limiting generalization, while excessively large magnitudes may introduce

redundant or uninformative samples, potentially harming performance. For the dimension and number of GNN layers, the overall trend also shows an initial increase followed by a decrease, except for the number of GNN layers on the Halo dataset. This is because moderate dimensions and depths allow the GNN to effectively capture graph structures, while overly large dimensions or too many layers can lead to overfitting or over-smoothing, thus degrading performance. Due to the small graph radius in the Halo dataset, fewer GNN layers are sufficient. The performance of top-h is largely insensitive to parameter settings, suggesting that the augmentation samples selected by MinMax are relatively reliable. For the number n/m of rounds/iterations, the overall trend shows a rapid increase in performance at first, followed by gradual stabilization. When the n and m are small, the model is under-trained. Although larger values can yield good performance, they also increase training cost. Therefore, we set the n and m to 3 to balance efficiency and effectiveness. For perturbations on edges and features, too small perturbations lead to insufficient diversity in the augmented samples, while overly large perturbations make the augmented samples too different from the training data, hindering model learning. Hence, moderate values are chosen for both.

5 Related Work

5.1 Log Anomaly Detection

Initially, log anomaly detection primarily rely on manually defined rules, such as threshold settings [19, 24, 38]. However, these rule-based methods require extensive domain expertise and are difficult to generalize across different datasets. To address these limitations, some studies have begun to explore machine learning techniques for log anomaly detection, including Support Vector Machines [14], Principal Component Analysis [35], and Clustering [15]. Nonetheless, these conventional approaches fail to capture the complex patterns and contextual dependencies inherent in logs.

With the advancement of deep learning, a growing number of studies have incorporated deep learning techniques into log anomaly detection. These approaches can be broadly categorized into sequence-based methods and graph-based methods. Sequence-based log anomaly detection methods treat logs as sequences of events and aim to learn the patterns of normal log sequences in order to predict the next likely event. If the actual event deviates significantly from the model's prediction, it is considered anomalous. Such sequence modeling approaches include Long Short-Term Memory [1, 3, 12, 22, 42, 43], Gated Recurrent Units [37], as well as Transformer and Transformer's variants [5–7, 11, 21, 23]. Graph-based log anomaly detection methods aim to represent log segments as graphs and detect anomalies by learning structural relationships within these graphs. For example, Xie et al. employ a Graph Transformer Neural Network for log anomaly detection through the graph classification task [34]. Zhang et al. utilize gated graph neural networks to extract graph representations and apply Deep Support Vector Data Description for anomaly detection [41]. Yan et al. incorporate the temporal sequence of the log graph into graph-based models for log anomaly detection [36]. Building upon this idea, a more recent work, SLAD, approaches log anomaly detection from the perspective of identifying log substructure patterns associated with normal and anomalous behaviors, aiming to detect anomalous code files from logs at a finer granularity [28].

Although existing log anomaly detection methods have achieved remarkable performance improvements, they typically rely on a key assumption that the distribution of log data remains consistent between the training and inference phases. However, in real-world deployments, log data distributions often shift over time due to system upgrades. Such distributional shifts lead to the presence of out-of-distribution (OOD) samples during inference. Since the model has never encountered these samples during training, it struggles to recognize their anomalous patterns, which significantly undermines the robustness and reliability of anomaly detection. In addition,

prior studies have explored the notion of unseen log events, which primarily refers to previously unobserved semantic content within log entries [10, 43]. These methods typically leverage language models to interpret new event semantics, focusing on textual novelty at the event level. In contrast, our study focuses on unseen log anomalies in terms of novel structural changes manifested in log graphs. Hence, they are unable to capture the unseen anomalies in our research.

5.2 Graph Out-of-Distribution Detection

Graph Out-of-Distribution (OOD) detection has attracted growing attention due to the vulnerability of graph neural networks (GNNs) to distributional shifts. Existing approaches can be broadly categorized into two main directions: invariant learning-based methods and causality-based methods [20]. Invariant learning methods aim to learn domain-invariant or stable representations across multiple environments to improve generalization under distributional shifts [25, 31, 39]. However, these methods rely on environment annotations, which are costly or even infeasible to obtain in graph settings due to the complex and hybrid nature of distribution shifts. Moreover, most of these methods focus on learning invariant representations at the global or subgraph level, making them less effective for detecting fine-grained, node-level anomalies. On the other hand, causality-based methods seek to exploit the underlying causal mechanisms that are stable across domains [2, 32]. However, the high computational cost of causal discovery and counterfactual reasoning can limit their scalability. Additionally, these methods tend to focus on global causal mechanisms and are generally designed to distinguish OOD nodes from in-distribution classes, rather than to offer explicit or interpretable categorization of anomalous nodes.

Comparison Our research aims to detect previously unseen anomalous code files from system logs by explicitly addressing the challenge of distribution shift. Our approach integrates data augmentation and iterative optimization to enhance model generalization. Specifically, it differs in two key aspects: (1) a novel MinMax strategy is introduced to select high-quality augmented anomalous samples that considers both similarity to and differences from known anomalies, thereby achieving a balanced trade-off between reliability and novelty in anomaly pattern selection; (2) an iterative training scheme is designed to progressively enhance the model. By evaluating candidate models and selecting augmented samples in each iteration, it incrementally integrates diverse anomaly patterns and enhances the model's robustness to OOD anomalies.

6 Conclusion

In this paper, we present UnseenLog, a novel log anomaly detection framework specifically designed to identify unseen anomalies that deviate from the training distribution. It introduces two key components: a MinMax strategy and a RISE scheme. The MinMax strategy effectively selects augmented anomaly samples that strike a balance between similarity to known patterns and novelty, improving the model's exposure to potential unseen patterns, while RISE further enhances the model through iterative candidate selection, progressively enhancing the graph model's robustness to unseen anomalies. Extensive experiments over several real-world datasets demonstrate the effectiveness of UnseenLog.

Acknowledgments

The project is primarily funded by the Shenzhen Science and Technology Program No. SYSPG202412 11173609009, and also partly supported by National Natural Science Foundation of China with grant No. 62472091. The first and second authors would like to thank the China Scholarship Council (CSC) for the support of their PhD scholarships.

References

[1] Rui Chen, Shenglin Zhang, Dongwen Li, Yuzhe Zhang, Fangrui Guo, Weibin Meng, Dan Pei, Yuzhi Zhang, Xu Chen, and Yuqing Liu. 2020. Logtransfer: Cross-system log anomaly detection for software systems with transfer learning. In *International Symposium on Software Reliability Engineering*. IEEE, 37–47.

[2] Yongqiang Chen, Yonggang Zhang, Yatao Bian, Han Yang, MA Kaili, Binghui Xie, Tongliang Liu, Bo Han, and James Cheng. 2022. Learning causally invariant representations for out-of-distribution generalization on graphs. *Advances in Neural Information Processing Systems* 35 (2022), 22131–22148.

[3] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of ACM conference on computer and communications security*. 1285–1298.

[4] Yuan Gao, Xiang Wang, Xiangnan He, Zhenguang Liu, Huamin Feng, and Yongdong Zhang. 2023. Addressing heterophily in graph anomaly detection: A perspective of graph spectrum. In *Proceedings of the ACM Web Conference 2023*. 1528–1538.

[5] Hongcheng Guo, Jian Yang, Jiaheng Liu, Jiaqi Bai, Boyang Wang, Zhoujun Li, Tieqiao Zheng, Bo Zhang, Junran Peng, and Qi Tian. 2024. Logformer: A pre-train and tuning pipeline for log anomaly detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 135–143.

[6] Haixuan Guo, Shuhan Yuan, and Xintao Wu. 2021. Logbert: Log anomaly detection via bert. In *International joint conference on neural networks*. IEEE, 1–8.

[7] Shangbin Han, Qianhong Wu, Han Zhang, Bo Qin, Jiankun Hu, Xingang Shi, Linfeng Liu, and Xia Yin. 2021. Log-based anomaly detection with robust feature extraction and online learning. *IEEE Transactions on Information Forensics and Security* 16 (2021), 2300–2311.

[8] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2017. Towards automated log parsing for large-scale log data analysis. *IEEE Transactions on Dependable and Secure Computing* 15, 6 (2017), 931–944.

[9] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[10] Van-Hoang Le and Hongyu Zhang. 2022. Log-based anomaly detection with deep learning: How far are we?. In *Proceedings of the 44th international conference on software engineering*. 1356–1367.

[11] Van-Hoang Le and Hongyu Zhang. 2024. Prelog: A pre-trained model for log analytics. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[12] Xiaoyun Li, Pengfei Chen, Linxiao Jing, Zilong He, and Guangba Yu. 2020. Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 92–103.

[13] Zeyan Li, Jie Song, Tieying Zhang, Tao Yang, Xiongjun Ou, Yingjie Ye, Pengfei Duan, Muchen Lin, and Jianjun Chen. 2025. Adaptive and Efficient Log Parsing as a Cloud Service. In *Companion of the 2025 International Conference on Management of Data*. 512–524.

[14] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 583–588.

[15] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *International Conference on Software Engineering Companion*. 102–111.

[16] Jialong Liu, Yanni Tang, Jiamou Liu, Kaiqi Zhao, and Wu Chen. 2023. Lglog: Semi-supervised graph representation learning for anomaly detection based on system logs. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 36–47.

[17] Jie Liu, Yanni Tang, Kaiqi Zhao, Jiamou Liu, and Wu Chen. 2024. TP-GNN: Continuous Dynamic Graph Neural Network for Graph Classification. *IEEE 40th International Conference on Data Engineering (ICDE)* (2024).

[18] Zhaoli Liu, Tao Qin, Xiaohong Guan, Hezhi Jiang, and Chenxu Wang. 2018. An integrated method for anomaly detection from massive system logs. *IEEE Access* 6 (2018), 30602–30611.

[19] Jian-Guang Lou, Qiang Fu, Shenqi Yang, Ye Xu, and Jiang Li. 2010. Mining invariants from console logs for system problem detection. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[20] Bin Lu, Ze Zhao, Xiaoying Gan, Shiyu Liang, Luoyi Fu, Xinbing Wang, and Chenghu Zhou. 2024. Graph out-of-distribution generalization with controllable data augmentation. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[21] Lei Ma, Lei Cao, Peter M VanNostrand, Dennis M Hofmann, Yao Su, and Elke A Rundensteiner. 2024. Pluto: Sample Selection for Robust Anomaly Detection on Polluted Log Data. *Proceedings of the ACM on Management of Data* 2, 4 (2024), 1–25.

[22] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.. In *IJCAI*, Vol. 19. 4739–4745.

[23] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2020. Self-attentive classification-based anomaly detection in unstructured logs. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1196–1201.
[24] John P Rouillard. 2004. Real-time Log File Analysis Using the Simple Event Correlator (SEC).. In *LISA*, Vol. 4. 133–150.
[25] Shiori Sagawa, Pang Wei Koh, Tatsunori B Hashimoto, and Percy Liang. 2020. Distributionally Robust Neural Networks. In *International Conference on Learning Representations*.
[26] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjing Wang, and Yu Sun. 2021. Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 1548–1554.
[27] Jianheng Tang, Jiajin Li, Ziqi Gao, and Jia Li. 2022. Rethinking graph neural networks for anomaly detection. In *International Conference on Machine Learning*. PMLR, 21076–21089.
[28] Yanni Tang, Zhuoxing Zhang, Kaiqi Zhao, Lanting Fang, Zhenhua Li, and Wu Chen. 2024. Substructure-aware Log Anomaly Detection. *Proceedings of the VLDB Endowment* 18, 2 (2024), 213–225.
[29] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).
[30] Yi Wan, Yilin Liu, Dong Wang, and Yujin Wen. 2021. Glad-paw: Graph-based log anomaly detection by position aware weighted graph attention network. In *Pacific-asia conference on knowledge discovery and data mining*. Springer, 66–77.
[31] Yanling Wang, Jing Zhang, Lingxi Zhang, Lixin Liu, Yuxiao Dong, Cuiping Li, Hong Chen, and Hongzhi Yin. 2024. Open-World Semi-Supervised Learning for Node Classification. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2723–2736.
[32] Ying-Xin Wu, Xiang Wang, An Zhang, Xiangnan He, and Tat-Seng Chua. 2022. Discovering invariant rationales for graph neural networks. *arXiv preprint arXiv:2201.12872* (2022).
[33] Wensheng Xia, Ying Li, Tong Jia, and Zhonghai Wu. 2019. Bugidentifier: An approach to identifying bugs via log mining for accelerating bug reporting stage. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 167–175.
[34] Yongzheng Xie, Hongyu Zhang, and Muhammad Ali Babar. 2022. LogGD: Detecting Anomalies from System Logs with Graph Neural Networks. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 299–310.
[35] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Largescale system problem detection by mining console logs. *Proceedings of SOSP'09* (2009).
[36] Lejing Yan, Chao Luo, and Rui Shao. 2023. Discrete log anomaly detection: A novel time-aware graph-based link prediction approach. *Information Sciences* 647 (2023), 119576.
[37] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1448–1460.
[38] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. 2013. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th annual computer security applications conference*. 199–208.
[39] Junchi Yu, Jian Liang, and Ran He. 2023. Mind the label shift of augmentation-based graph ood generalization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 11620–11630.
[40] Dingyi Zeng, Wanlong Liu, Wenyu Chen, Li Zhou, Malu Zhang, and Hong Qu. 2023. Substructure aware graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 11129–11137.
[41] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *Proceedings of the 44th International Conference on Software Engineering*. 623–634.
[42] Lingzhe Zhang, Tong Jia, Mengxi Jia, Ying Li, Yong Yang, and Zhonghai Wu. 2024. Multivariate log-based anomaly detection for distributed database. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4256–4267.
[43] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.
[44] Lingxiao Zhao, Wei Jin, Leman Akoglu, and Neil Shah. 2021. From stars to subgraphs: Uplifting any GNN with local structure awareness. *arXiv preprint arXiv:2110.03753* (2021).

Received July 2025; revised October 2025; accepted November 2025