Software Quality Journal (2024) 32:821–845
https://doi.org/10.1007/s11219-024-09670-8

RESEARCH

KAD: a knowledge formalization-based anomaly detection approach for distributed systems

Xinjie Wei[1] · **Chang-ai Sun**[1] · **Xiao-Yi Zhang**[1]

Accepted: 21 March 2024 / Published online: 10 May 2024

Abstract
Large-scale distributed systems are becoming key engines of the IT industry due to their scalability and extensibility. A distributed system often involves numerous complex interactions among components, suffering anomalies such as data inconsistencies between components and unanticipated delays in response times. Existing anomaly detection techniques, which extract knowledge from system logs using either statistical or machine learning techniques, exhibit limitations. Statistical techniques often miss implicit anomalies that are related to complex interactions manifested by logs, whereas machine learning techniques lack explainability and they are usually sensitive to log variations. In this paper, we propose KAD, a knowledge formalization-based anomaly detection approach for distributed systems. KAD includes a general knowledge description language (KDL), leveraging the general structure of system logs and extended Backus-Naur form (EBNF) for complex knowledge extraction. Particularly, the semantic set is constructed based on the bidirectional encoder representation from the transformer (BERT) model to improve the expressive capabilities of KDL in knowledge description. In addition, KAD incorporates distributed scheduling computation module to improve the efficiency of anomaly detection processes. Experimental results based on two widely used benchmarks show that KAD can accurately describe the knowledge associated with anomalies, with a high F1-score in detecting various anomaly types.

Keywords Distributed systems · Domain-specific language · Anomaly detection · Industrial evidence

1 Introduction

In commercial IT systems, distributed architectures, such as Hadoop (Apache Hadoop, 2023), Spark (Apache Spark, 2023), and Ray (Moritz et al., 2018; Liang et al., 2023), have been widely adopted. However, the interactions among numerous

✉ Chang-ai Sun
casun@ustb.edu.cn

Xinjie Wei
xinjiewei@xs.ustb.edu.cn

Xiao-Yi Zhang
xiaoyi@ustb.edu.cn

[1] School of Computer and Communication Engineering, University of Science and Technology Beijing, 30 Xueyuan Road, Haidian District, Beijing 100083, China

components in these systems often lead to anomalies, such as data inconsistencies and delayed response times. These anomalies may lead to application crashes or degradation of user satisfaction, potentially resulting in significant economic losses (Haoming & Yuguo, 2020).

Anomaly detection traditionally involves two primary techniques: statistical-based and learning-based techniques. Statistical-based techniques (Ali et al., 2023) analyze the statistical properties such as averages or standard deviations from system logs, metrics, code, or traces and then identify some patterns based on these properties as features. Finally, they try to detect anomalies based on the assumption that anomalies often deviate from the expected patterns. Learning-based techniques (Gómez et al., 2023) leverage machine learning algorithms to identify anomalies. They train a model on a dataset that represents normal behaviors, and then use the model to identify the anomaly data that deviate significantly from the learned knowledge.

In distributed systems, logs are commonly used for troubleshooting (Le & Zhang, 2022) since they record system states and critical events at runtime. The hidden abundant information in logs offers a good view to analyze system problems. Hence, by mining knowledge in a large amount of logs, log-based anomaly detection techniques can help to enhance system health, stability, and availability. To obtain the knowledge that can be used to detect anomalies from logs, statistical-based anomaly detection techniques convert information in logs into a piece of interpretable knowledge based on specific language for identifying anomalies within distributed systems. For example, Majeed et al. (2019) utilize the Ariel Query Language (AQL) to describe knowledge for monitoring network activities and identifying potential threats. Similarly, Hristov et al. (2021) employ the Search Processing Language (SPL) to describe knowledge in Splunk Enterprise to detect anomalies. These techniques provide precise and interpretable anomaly detection results. However, they may overlook implicit anomalies, those that are not directly discernible or fall outside the existing domain knowledge. Conversely, learning-based techniques leverage machine learning models, such as support vector machines, transformer models, and convolutional neural networks (CNNs), to extract knowledge from logs. For example, Nedelkoski et al. (2020) use a transformer model to differentiate between normal and abnormal log representations. Lu et al. (2018) apply word embedding and CNNs to encode logs into feature matrices for knowledge mining. These techniques can reveal complex knowledge hidden within logs. However, they suffer from limited interpretability; the knowledge extracted by these models often lacks clear interpretation, making it challenging to comprehend the rationale behind detected anomalies. Additionally, these techniques may fail to capture the fundamental nature of distributed systems due to the absence of domain-specific knowledge. Moreover, learning-based models are inherently sensitive to the quality of input data: noisy data can lead to erroneous knowledge extraction.

In summary, both statistical-based and learning-based anomaly detection techniques offer unique strengths but are constrained by significant limitations. This paper seeks to introduce a more robust, interpretable, and high-quality knowledge extraction technique for distributed systems, addressing the limitations of existing techniques. Specifically, we identify three major challenges:

1. **The absence of a general knowledge description language**: Existing techniques often rely on domain-specific knowledge description languages such as Ariel Query Language (AQL) (IBM, 2023) for IBM QRadar and Search Processing Language (SPL) (Splunk

Enterprise, 2023) for Splunk, which limits their applicability. Therefore, a general knowledge description language is desired to describe complex knowledge for various distributed systems.

2. **The lack of a general approach for knowledge extraction**: Existing techniques typically use system-specific knowledge description languages and extraction methods, requiring a specific knowledge extraction approach for each system.
3. **The difficulty in capturing knowledge changes**: Existing techniques struggle to capture knowledge changes due to system updates.

To address these challenges, we propose KAD, a knowledge formalization-based anomaly detection approach for distributed systems. KAD introduces a general knowledge description language, KDL, to standardize the knowledge and simplify its representation, thus improving the interpretability. KDL uses the extended Backus-Naur form (EBNF) (Tietz & Annighoefer, 2022) to formalize elements extracted from the general log structures, such as timestamps, log templates, and variants. The log template is comprised of fixed strings manifesting scheduling actions. The timestamps and variants describe the relationship between log templates. KDL enhances expressive abilities of the knowledge, and can describe complex knowledge, such as temporal order, quantitative relationships, parameter associations, non-associations, logical disjunctions, recursive structures, and so on. Additionally, to improve the robustness of KAD, KDL defines the semantic set, which employs the BERT model (Devlin et al., 2019) to describe sets of log templates with similar semantic information. It is labeled with unique and human-readable strings that capture the semantic information. After obtaining the KDL knowledge, KAD employs a distributed scheduling computation method to further improve the efficiency of anomaly detection.

We evaluate KAD by conducting a series of experiments on two benchmark distributed systems—Ray and Hadoop. The results show that KDL is versatile in extracting complex knowledge from various distributed systems. In addition, KAD achieves a high F1-score (1 in Hadoop and 0.95 in Ray), and it can capture knowledge changes related to system updates.

In summary, this paper presents the following contributions:

1. We propose a general knowledge description language, KDL, for describing complex knowledge extracted from distributed logs using the general log structure and EBNF.
2. We introduce the semantic set, enhanced by the BERT model, to improve the expressive capabilities of KDL.
3. We propose KAD, a knowledge formalization-based anomaly detection approach for distributed systems, which employs distributed scheduling computation to enhance the efficiency of anomaly detection.
4. We report on a series of experiments to evaluate the effectiveness, efficiency, and robustness of KAD on Ray and Hadoop distributed systems.

The rest of this paper is organized as follows. Section 2 introduces the background about knowledge-based anomaly detection and the motivation of this paper. Section 3 presents the KAD framework. Section 4 reports on the experimental evaluation. Related works are reviewed in Section 5. Finally, Section 6 concludes the paper and outlines some future work directions.

2 Background and motivation

In this section, we introduce the anomaly detection of the distributed system and the motivation of this paper.

2.1 Anomaly detection of distributed systems

A distributed system consists of software components spreading over different computers but running as a single entity (Hidayati et al., 2023). It clusters a group of computers working together to appear as a single computer to the end user. These machines have a shared state, operate concurrently, and can fail independently without affecting the whole system's running status.

In distributed systems, *logs* play an important role in data replication (Le & Zhang, 2022). Multiple components work together to ensure the high availability of the provided services. To enable replication of system activities, a *log* is generated to record the occurrence of every event, which normally includes a timestamp, a template, associated parameter values, and other information. The *log template* is comprised of fixed strings manifesting some scheduling actions, while the parameter values specify the context. For example, the log data "*2021-07-14 09:30:48, 800 sservice_based_accessor.cc:392: Subscribing update operations of actor, actor id=cad70dc, job id=0080*" describes the update of the actor component. The log template of this log is "Subscribing update operations of actor, actor id=, job id=" and the associated parameter values include the values of "actor id" and "job id."

An anomaly is an unexpected event or observation significantly deviating from the normal data. It may result in poor system performance such as high CPU consumption and high latency. Anomaly detection in a distributed system is typically enacted based on runtime information, such as *logs*.

Classical statistical-based or learning-based anomaly detection techniques attempt to extract behavioral knowledge of anomalies from system logs (Le & Zhang, 2022). They normally first collect logs during execution and obtain some knowledge from these logs based on different techniques, such as statistic analysis techniques or machine learning techniques. The knowledge is then used to detect anomalies when logs deviate from the normal knowledge or match the anomaly knowledge.

2.2 Motivation

Consider a case in the Ray distributed system, where a simple change in the syntax of a log. For instance, the log for reporting node disconnections originally is as follows:

"*Node [NodeID] disconnected*,"

and it is updated to the following:

"*Disconnected: Node [NodeID]*."

Despite the change in syntax, the semantic meaning of reporting a node disconnection remains the same.

However, this modification poses some challenges for existing anomaly detection techniques: Statistical-based techniques typically rely on predefined knowledge from logs. In this case, a change in the log might not align with the established knowledge, leading to the technique missing this anomaly. Learning-based techniques depend on training models with historical data, which includes predefined logs. When the log changes, as in this case, the model might fail to recognize the new log as it deviates from the “learned” knowledge.

This case highlights a critical challenge in existing anomaly detection techniques: the lack of a general and robust technique to detect anomalies in distributed systems. It is valuable to construct more effective knowledge that can be used to detect more types of anomalies. Therefore, we propose a knowledge formalization approach for anomaly detection. It aims to improve the expressiveness of knowledge and to detect anomalies in evolving distributed systems.

3 Overview

KAD aims to effectively, efficiently, and robustly detect anomalies in various distributed systems by leveraging knowledge from logs. As illustrated in Fig. 1, KAD comprises four parts:

(i) *Knowledge construction*: KAD employs the extended Backus-Naur form (EBNF) to establish the general knowledge description language—KDL, and then use KDL to extract some knowledge sets based on the experience library provided by distributed system experts.

(ii) *Knowledge parsing*: The extracted knowledge sets are then transformed into log *template vectors* and *knowledge entities* based on a knowledge parsing algorithm. Particularly, during the construction of template vectors, the BERT model is used to extract the semantic information of log templates to improve the expressive capabilities.

(iii) *Log preprocessing*: KAD uses log-monitoring tools (e.g., Elastic Stack) to collect and parse the logs obtained from the distributed system. Then, the BERT model is

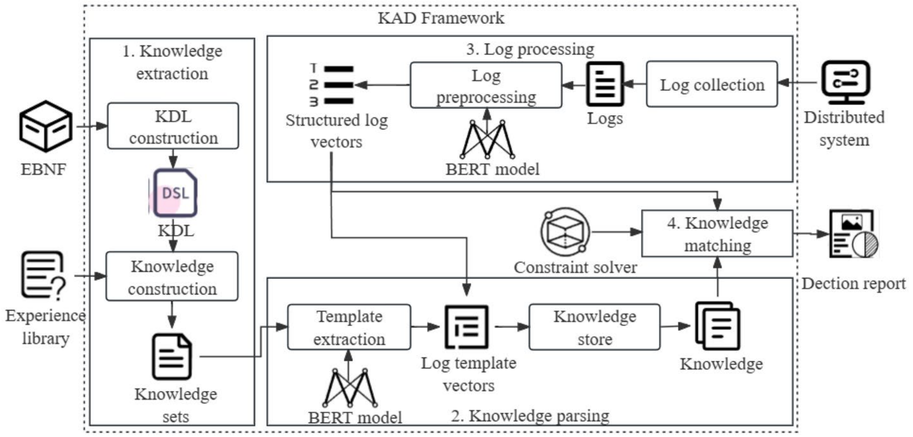


Fig. 1 Framework of KAD

used again to preprocess logs, expanding the log template vectors to improve the efficiency of subsequent knowledge matching.

(iv) *Knowledge matching*: Finally, the constraint solver is applied to match the structured log vectors and the KDL knowledge to generate the anomaly detection report.

In the following parts, each phrase in the KAD approach is introduced, including knowledge construction (Section 3.1), knowledge parsing (Section 3.2), log preprocessing (Section 3.3), and knowledge matching (Section 3.4).

3.1 Knowledge extraction

The knowledge extraction phase is illustrated in Fig. 2. Firstly, we identify the general elements among logs to obtain a general log structure. Then, elements related to the knowledge are extracted from the general log structure, such as timestamps, log types, log templates, variables, and other information. Next, the knowledge description language (KDL) is formally defined through the extended Backus-Naur form (EBNF). Finally, the experience library is used to generate the knowledge sets based on KDL.

3.1.1 Log structuring

The log structuring phase is fundamental to the knowledge extraction phase in the KAD framework, transforming unstructured or semi-structured logs into the format conducive to analysis.

Logs are typically unstructured or semi-structured text produced by logging statements in the source code, which poses a significant challenge in anomaly detection distributed systems due to their varied structures. The purpose of log structuring is to convert these logs into a structured format, thereby facilitating effective analysis. In our study, Drain (He et al., 2017) is used for the log transformation since it is a widely adopted log parsing technique that employs a fixed depth tree (FDT) and a clustering algorithm to automate the structuring of log data.

As illustrated in Fig. 3, a typical log comprises two main components: the log header and the log contents. The log header, determined by the logging framework (e.g., SLF4J), usually includes elements such as the timestamp (e.g., “2021-07-14 09:30:48800”), log type (e.g., “D”), process ID (e.g., “352591”), thread ID (e.g., “352767”), and output module (e.g., “service_based_accessor.cc:392”). The log content, written by developers, describes specific system runtime events. It generally consists of a log template, which is a fixed string indicating runtime events (e.g., “Subscribing update operations of actor, actor id=, job id=”), and variable values that provide context to these events (e.g., “Cda70dc-75c0f18c9c2bff494f9280080” and “f9280080”).

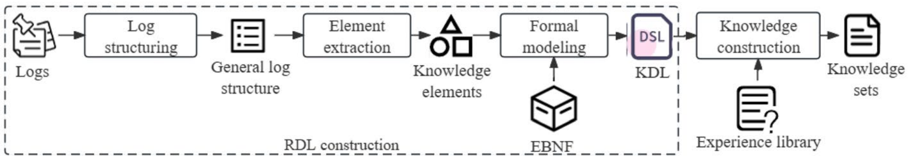
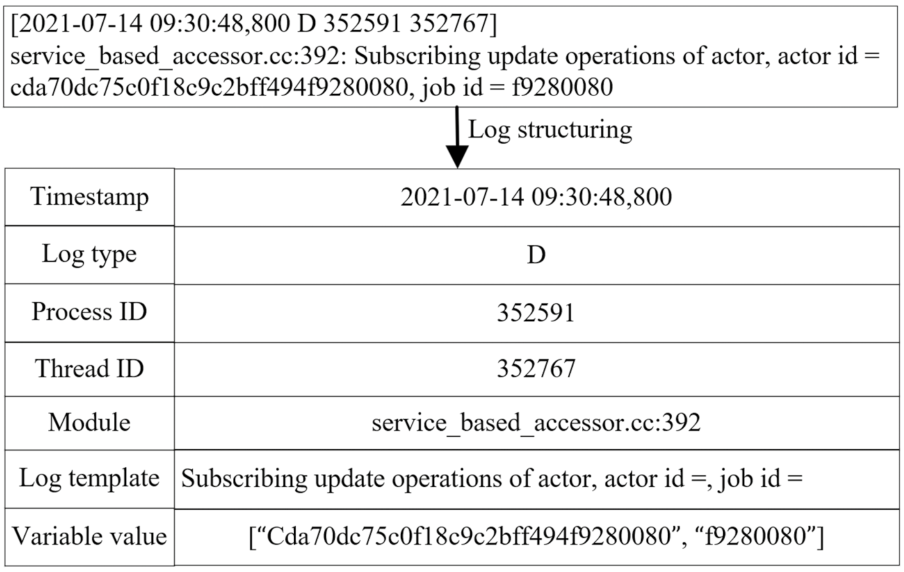


Fig. 2 Knowledge extraction



```
[2021-07-14 09:30:48,800 D 352591 352767]
service_based_accessor.cc:392: Subscribing update operations of actor, actor id =
cda70dc75c0f18c9c2bff494f9280080, job id = f9280080
```

Timestamp	2021-07-14 09:30:48,800
Log type	D
Process ID	352591
Thread ID	352767
Module	service_based_accessor.cc:392
Log template	Subscribing update operations of actor, actor id =, job id =
Variable value	[“Cda70dc75c0f18c9c2bff494f9280080”, “f9280080”]

Fig. 3 Example of log structuring

To facilitate the construction of KDL, it is imperative to establish a general log structure GLS, as defined in Definition 1.

Definition 1 (General Log Structure) Let GLS denote the framework that standardizes the structure of the log provided by $\langle T, LT, V, O\rangle$, where T denotes the timestamp, LT denotes the log template, V denotes the sets of variable values within LT, and O denotes the other information in the log, such as log type and output module.

3.1.2 Element extraction

Element extraction is a critical phase that describes the knowledge extracted from distributed system logs. This phase focuses on identifying specific elements within the general log structure GLS, each playing a pivotal role in the knowledge representation.

The extraction process isolates three primary elements from GLS: log template LT, timestamp T, and variable values V.

1. LT involves the system runtime activities, serving as a guide to locate and interpret logs within distributed systems. This element is crucial for understanding the activity dynamics of the system and detecting anomalies.
2. T represents the temporal information of the log, recording the timing of each event in the system. It is instrumental in establishing timing constraints $Cons^T$ between LTs, thus enabling the understanding of log sequences and timings.

3. V represents the variable values within LTs, denoting variable constraints $Cons^V$ such as equality and inclusion. V is essential for capturing the dynamic changes in logs, which can indicate deviations from normal system activities.

Logs in distributed systems are subject to continuous changes due to system updates, thus it is essential to regularly update the corresponding log templates to improve the expressive capabilities of the knowledge. Additionally, experts may not be familiar with all log templates in each system; thus, a method to manage these templates effectively and reduce reliance on domain-specific knowledge is necessary.

To address these challenges, we introduce the concept of a "Semantic Set" SS, as defined in Definition 2. The semantic set is designed to encapsulate a group of log templates that collectively express a specific system runtime activity.

Definition 2 (Semantic Set) Let SS denote the set of log templates that reveals a common system activity provided by $\langle LT, Cons\rangle$, where LT is the same as defined in Difination 1. Cons denotes the constraints within LTs, $Cons \in \{Cons^T, Cons^V, Cons^L\}$, in which $Cons^T$ denotes the timing constraint, $Cons^V$ denotes the variable constraint, and $Cons^L$ denotes the logic constraint (such as negation, conjunction, or recursion).

SS allows for a more nuanced and flexible representation of system activities, significantly enhancing the expressive capabilities of the extracted knowledge.

3.1.3 Formal modeling

The formal modeling phase is a critical step in the process of knowledge extraction, in which we define the general knowledge description language KDL based on the elements extracted from the general log structure GLS.

The knowledge description language KDL is a general language containing various components crucial for knowledge representation, as defined in Definition 3.

Definition 3 (Knowledge Description Language) Let KDL denote the general knowledge description language provided by $\langle SS, Cons, Ann, Know\rangle$, where SS and Cons are the same as defined in Definition 2, Ann denotes annotations , and Know is the association of SS, Cons, and Ann.

The formal modeling of KDL is implemented using the extended Backus-Naur form (EBNF) (Tietz & Annighoefer, 2022), which is a family of notations, any of which can be used to express context-free grammar, and can be used to make a formal description of a formal language such as a computer programming language.

The formalization of KDL employs the extended Backus-Naur form (EBNF), a sophisticated notation known for its capability to express context-free grammars. EBNF's flexibility and precision make it an suitable tool for crafting a formal description of complex languages, including those used in computer programming. The formalization of KDL using EBNF can help to create a structured and general language for knowledge description. The EBNF grammar for KDL is illustrated in Fig. 4.

```
anomaly = [anomalyAnnotation] sequences
anomalyAnnotation = ‘@’ anomalyAnnotaionTag [‘(’ [
    ↪ paramList] ‘)’]*
anomalyAnnotaionTag = ‘Anomaly’ | ‘SemanticSet’ | ‘
    ↪ SimpleAnomaly’
paramList = param [‘,’ param]*
param = type ‘=’ string
type = NAME
NAME = [a-zA-Z_][a-zA-Z0-9]*
string = StringLiteral
StringLiteral = ‘(~[‘‘\\\r\n] | ‘\\’ [btnfr’’\\]) +? ’
sequences = ‘(’ semantics [constraint semantics]* ‘)’+
semantics = template [(‘|’ | ‘ ‖ ’) template]*
template = templateAnnotation* [‘!’] [recursionPrefix?]
    ↪ templateAnnotation* type string
templateAnnotation = ‘@’ templateAnnotaionTag [‘(’ [
    ↪ paramList] ‘)’]*
recursionPrefix = (OctDigits | WildCardNum) ‘{’
    ↪ StringLiteral ‘}’
OctDigits = [0-9]+
WildCardNum = ‘?’ | ‘+’ | ‘*’
patternAnnotaionTag = ‘Template’ | ‘Symptom’ | ‘RootCause’
    ↪  | NAME
constraint = ‘ ≥ ’ | ‘ ≤ ’ | ‘→’ string ‘→’ | ‘<-’ string
    ↪ ‘<-’
WS = [\t\r\n]+ → skip
COMMENT = ‘/*’ .*? ‘*/’ → channel(HIDDEN)
LINE_COMMENT = ‘//’ ~[\r\n]* → channel(HIDDEN)
```

Fig. 4 EBNF grammar of KDL

1. **Knowledge**: The knowledge of “anomaly” is comprised of optional annotations “anomalyAnnotation” and a sequence of log templates “sequences.” The “anomalyAnnotation” includes a specific tag “anomalyAnnotaionTag” and a “paramList” to record the information of variables (optional). The “sequences” is a combination of semantic sets “semantics” and their corresponding constraints “constraint.”
2. **Annotation**: *KDL* provides an annotation functionality to facilitate the presentation of knowledge. Particularly, the annotations added by system experts using KDL are more understandable. This includes knowledge annotations “anomalyAnnotation,” template annotations “templateAnnotation,” and concealable notes “COMMENT” and “LINE_COMMENT.” Furthermore, *KDL* contains the custom operators for performing specific operations, thereby improving its ability to describe complex knowledge.
3. **Constraint**: *KDL* defines three types of constraints:

    (i) Timing constraint denotes the sequential order of objects (such as log templates and semantic sets). “$a =\rangle b$” indicates a precedes b, and “$a\langle= b$” denotes the reverse, where a and b are the objects.

(ii) Variable constraint denotes variable relationships between objects. "–⟩*string*–⟩" indicates that the left object precedes the right, with "string" representing the variable constraint. For example, "left.getAttribute(actorId)=right.getAttribute(actorId)" indicates identical actor IDs in both objects. "⟨–*string*⟨–" represents the opposite arrangement.

(iii) Logic constraint contains non-constraint "!", or-constraint "||" and "|", and recursive constraints "recursionPrefix." The non-constraint indicates the absence of an object, the or-constraint indicates the occurrence of at least one of the objects, and the recursive constraint denotes multiple occurrences of an object under specific recursive conditions.

4. **Semantic set**: The semantic set "semantics" refers to a collection of templates "template" that reflect similar activities. Each template within "semantics" may contain annotations ''TemplateAnnotation", a type specifier "type", the content of the log template "string", and optionally, associated constraints. These elements collectively define the semantic set, providing a structured and detailed representation of log activities.

Figure 5 presents an example of knowledge representation. The anomaly is a kind of request frequent, recorded by *tags = "Request frequent"*; the root cause of the anomaly is that the number of access requests exceeds 100 times within 60 seconds, presented by the sentences *@RootCause (Label = "Access request")*, *getCount(@RootCause > 100)*, and *timeDiff(@RootCause, @RootCause, s) < 60*. Consequently, this anomaly lead to the failure of request connection, recorded by *@SemanticSet (Label = "Connection failed")*. The annotations "*@Anomaly*," "*@RootCause*," and "*@SemanticSet*" serve as contextual markers in the knowledge representation. The custom operators "*getCount*(*@RootCause*)" and "*timeDiff*(*@RootCause*, *@RootCause*, *s*)" describe complex constraints within the knowledge. These operators can help to express the complex relationships, enabling a more comprehensive and flexible representation of the knowledge from various domains. The semantic sets in Fig. 5 are detailed in Fig. 6. The set labeled "*Access request*" includes two log templates that describe web request failures, connected by the logical "||" constraint. Additionally, the set labeled "*Connection failed*" comprises a single log template that characterizes a failed connection request. These semantic sets can help to encapsulate complex system activities within a structured and interpretable format.

3.1.4 Knowledge construction

The knowledge construction phase leverages the experience library, a repository containing detailed records and analyses of anomalies observed during system operations.

```
@Anomaly(tags="Request frequent", getCount(@RootCause) >
   ↪ 100, timeDiff(@RootCause,@RootCause,s) < 60)
@RootCause (Label="Access request")
@SemanticSet (Label="Connection failed")
```

Fig. 5 Example of KDL knowledge

```
@SemanticSet (Label="Access request")
L "Web traffic: http status code: 500" ||
L "Web traffic: http status code: 500, user agent:
    ↪ malicious"
@SemanticSet (Label="Connection failed")
E "Connection refused: Unable to connect to the server
    ↪ serverID"
```

Fig. 6 Example of semantic sets

This library serves as an important resource for the generation of knowledge for anomaly detection. Firstly, anomalies recorded in the experience library are systematically categorized based on their types, which can help to understand the nature and characteristics of different anomalies. Secondly, log templates and their associated constraints are extracted from the experience library. For example, a log template denoted as LT_A will always be followed by another log template denoted as LT_B, within a predefined time frame. This sequence can help to understand and predict the occurrence of anomalies. Thirdly, KDL is employed to generate knowledge that encapsulates these log templates, constraints, and other relevant information, including contextual annotations. This step transforms raw data into structured, interpretable knowledge. Additionally, constraints and annotations in the knowledge can be fine-tuned to optimize the accuracy of knowledge. Finally, the KDL knowledge needs to be validated using historical logs, which can help to ensure the reliability of the knowledge.

3.2 Knowledge parsing

After obtaining the KDL knowledge, KAD compiles them for further knowledge matching. Specifically, each log template in the knowledge is converted into log template vectors containing semantic information, and the KDL knowledge is converted into a directed acrylic graph. Knowledge parsing consists of two phases: template vectorization and knowledge storage.

3.2.1 Template vectorization

For log templates contained in the KDL knowledge, KAD converts them into log template vectors using the BERT model to save their semantic information, as shown in Fig. 7: Specifically, the sentences of each log template are used as inputs for the BERT model, yielding the vector for each word; then, these vectors are passed through a pooling layer; finally, the *log template vector* (denoted by LTV) is created for each log template LT, as defined in Definition 4.

Definition 4 (Log template vector) Let LTV denote the log template with its corresponding semantic information provided by $\langle LT, SI\rangle$, where LT is the same as defined in Definition 1, and SI denotes the corresponding semantic information (e.g., the output vector of BERT) of LT.

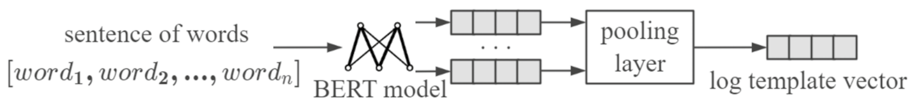


Fig. 7 Vector representation

3.2.2 Knowledge storage

In order to improve the efficiency of subsequent processing, we used the knowledge graphs (Hogan et al., 2021) to store the knowledge, in which each KDL knowledge is transformed into a directed acyclic graph (DAG). This transformation employs a depth-first traversal algorithm, ensuring that the knowledge is stored in an organized and accessible manner. The use of the DAG structure facilitates the efficient representation of the complex relationships and dependencies inherent in the knowledge. This structure not only aids in the quick retrieval and processing of knowledge but also supports the complex analysis required for anomaly detection in distributed systems. The DAG representation of knowledge is defined in Definition 5.

Definition 5 (DAG knowledge) Let G^R denote the knowledge represented as DAGs provided by $\langle N, E\rangle$, where N denotes the set of log template vectors LTVs, as defined in Definition 4,and E denotes the set of constraints among N, $E = \langle LTV_i, Cons, LTV_j\rangle$, $LTV_i \in N$, $LTV_j \in N$, and $LTV_i \neq LTV_i$, and $Cons$ is the same defined in Definition 3.

Algorithm 1 The algorithm of knowledge parsing

```
Input: Knowledge sets $Knows = \{Know_1, Know_2, ...\}$
Output: Log template vectors $LTV$s, Knowledge sets $G^R$
1: initialize $LTV$, $G^R$, $AST$ to empty
2: while knowledge $Know$ in knowledge sets $Knows$ do
3:     $AST \Leftarrow ruleToAST(Know)$
4:     initialize last template $LT_{last}$, forward constraint $Cons_{forward}$, backward
   constraint $Cons_{back}$ to empty
5:     while $childAST$ in $AST$ do
6:         if $LT \in childAST$ then
7:             $LT \Leftarrow getTemplate(childAST)$
8:             $SI \Leftarrow BERT(LT)$
9:             $LTV \Leftarrow add(LT, SI)$
10:            $N \Leftarrow addNode(LTV)$
11:            while $LT_{last}$ is not empty do
12:                if $Cons_{forward} \in childAST$ then
13:                    $Cons \Leftarrow getConstraint(childAST)$
14:                    $E \Leftarrow addEdge(LT_{last}, Cons, LTV)$
15:                else if $Cons_{back} \in childAST$ then
16:                    $Cons \Leftarrow getConstraint(childAST)$
17:                    $E \Leftarrow addEdge(LTV, Cons, LT_{last})$
18:                end if
19:                $LT_{last} = LTV$
20:            end while
21:        end if
22:    end while
23: end while
24: Return $LTV$, $G^R$
```

Algorithm 1 shows the process of knowledge parsing. Each *KDL* knowledge, represented as *Know*, is transformed into an abstract syntax tree *AST* on lines 2–3. Then, the

AST is traversed using a depth-first traversal algorithm. If the child AST *childAST* contains a log template *LT*, the BERT model is employed to extract the corresponding semantic information, producing log template vector *LTV*, which is added to node *N* on lines 5–10. If *childAST* contains a constraint *Cons*, the *childAST* is traversed to identify templates linked to this constraint. These templates, along with the constraint, are subsequently added to the edges *E* of the DAG on lines 11–19. By iterating through the above steps, the knowledge sets G^R and the log template vectors *LTV*s are obtained.

3.3 Log processing

The log processing phase is illustrated in Fig. 8. Firstly, KAD employs log-monitoring tools, such as Elastic Stack, to collect logs from distributed systems. Then, the unstructured or semi-structured logs are transformed into structured logs based on Drain (He et al., 2017), a log parsing tool that is widely used in log-based anomaly detection. Finally, the BERT model (Devlin et al., 2019) is employed to extract semantic information from each structured log, to obtain the *structured log vectors*, as defined in Definition 6. 3.2

Definition 6 (Structured log vectors) Let *SLV* denote the structured log provided by $\langle T, LTV, V, O\rangle$, where T, V, and O are the same defined in Definition 1, and *LTV* is the same defined in Definition 4.

3.4 Knowledge matching

After log processing, the KAD can conduct knowledge matching to select a set of logs that match the identified KDL knowledge through a proposed constraint solver, as these logs can reveal system anomalies.

To improve the efficiency of knowledge matching, a fine-grained distributed task scheduling method is proposed. This method incorporates comprehensive failover and restart mechanisms and can help to effectively manage task allocation and minimize redundant computations. Specifically, we defined the matching of structured template vectors with KDL knowledge as a distributed computation task, as defined in Definition 7.

Definition 7 (Distributed computation task) Let *DCtask* denote the distributed computation task provided by $\langle G^R, SLV, Pro\rangle$, where G^R is the knowledge defined in Definition 5, *SLV* denotes the structured log vectors defined in Definition 6, and *Pro* denotes the progress of the *DCtask* provided by $\langle Comp, Flag, Num, Threshold\rangle$, *Comp* denotes the set of logs that have completed the *DCtask*, *Flag* denotes the computation completion label of the *DCtask*, *Num* denotes the number of *DCtask* fault occurrences, and *Threshold* denotes the the fault occurrence threshold.

During knowledge matching, KAD executes all these distributed computation tasks in parallel. The progress of each distributed computation task is tracked, including completed matched vectors, completion labels, the number of fault occurrences, and the fault occurrence threshold. If a computation task fails, it is reassigned to a new computation node: if the number of fault occurrences exceeds the fault occurrence threshold, the task is discarded; otherwise, the computation task is restarted, and it reads the corresponding knowledge, the set of vectors to be matched, and the progress.

Algorithm 2 The algorithm of knowledge matching

```
Input: structured log vectors $SLV$, knowledge sets $G^R$
Output: knowledge-matching log sequences $Seq$
1: initialize distributed computation task queue $DCque$ to empty
2: while knowledge $Know$ in $G^R$ do
3:     initialize task $DCtask \Leftarrow generaTask(Know, SLV, Pro)$
4:     $DCque \Leftarrow addTask(DCtask)$
5:     while $length(\text{DCque}) \geq 1$ do
6:         while $DCtask$ in $DCque$ do
7:             if $DCtask.Pro$ is empty then
8:                 initialize $DCtask.Pro.Comp = null$, $DCtask.Pro.Flag = false$, $DCtask.Pro.Num = 0$, $DCtask.Pro.Threshold = threashold$
9:                 initialize constraint $Cons$, log sequence $Seq$ to empty
10:            else
11:                $DCtask.Pro \Leftarrow sendTaskProgress(DCtask)$
12:            end if
13:            run $DCtask$ on distributed computation node $node$
14:            $Cons \Leftarrow generaCons(Know)$
15:            $Seq \Leftarrow constraintSolver(Cons, SLV)$
16:            if $Seq$ is not empty then
17:                $DCtask.Pro.Flag = true$
18:                return $Seq$
19:            else
20:                $DCtask.Pro.Num + +$
21:                if $DCtask.Pro.Num > DCtask.Pro.Threshold$ then
22:                    remove task $DCtask$
23:                else
24:                    insert task $DCtask$ into $DCque$
25:                end if
26:            end if
27:        end while
28:    end while
29: end while
30: Return $Seq$
```

Algorithm 2 details the process of knowledge matching. Firstly, the algorithm initializes a queue for distributed computation tasks, *DCque*, which is set to empty on line 1. For each knowledge *Know* within the knowledge sets G^R, a corresponding distributed computation task *DCtask* is generated and added to *DCque* on lines 2–4. Secondly, each *DCtask* in *DCque* is iterative processed on lines 5–12: if the task is new, initialize its progress indicators *DCtask.Pro* and prepare for constraint resolution; else, the task is retrieved and processed from the current task progress *DCtask.Pro*. Thirdly, the *DCtask* is processed on a distributed computation node *node*, generating constraints *Cons*, and solving these constraints against the structured log vectors *SLV* to obtain log sequences *Seq* on lines 13–24: If *Seq* exists, indicating successful task completion, the algorithm flags *DCtask.Pro.Flag* as true and returns *Seq*; if the task fails, increment the failure count *DCtask.Pro.Num*; if *DCtask.Pro.Num* exceeds the predetermined threshold *DCtask.Pro.Threshold*, the task is removed from the queue; otherwise, the task is

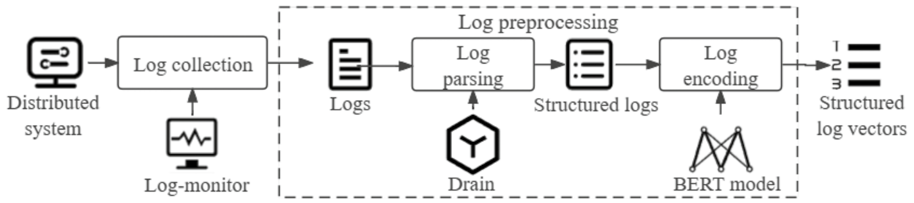


Fig. 8 Log processing

reinserted into *DCque* for reprocessing. Finally, the knowledge-matching log sequences *Seq* is obtained.

4 Evaluation

To evaluate the viability of KDL, and the effectiveness, efficiency, and robustness of KAD, a comparative analysis was conducted, comparing four existing anomaly detection techniques on two distributed system benchmarks.

4.1 Research questions

The following research questions are explored:

- RQ1: Is KDL viable for different distributed systems? The viability is evaluated on two distributed systems—Hadoop and Ray.
- RQ2: How effective is KAD in distributed system anomaly detection? The effectiveness is evaluated on the two distributed systems compared with four baseline techniques.
- RQ3: How efficient is KAD in distributed system anomaly detection? The efficiency is evaluated on the two distributed systems compared with four baseline techniques.
- RQ4: How robust is KAD to the variation of datasets? The robustness is evaluated on the two distributed systems compared with four baseline techniques, in which datasets of the two systems are modified to generate four variants.

4.2 Experimental design

Datasets The evaluation was conducted on two typical distributed systems: Hadoop and Ray. Hadoop is a framework used for processing large data sets across computer clusters, and Ray is a versatile distributed computing framework aiming to provide a general programming interface for distributed systems. The evaluation was performed using two datasets:

- *HDFS* is a public dataset provided by Du et al. (2017). It was generated by running map-reduce tasks on over 200 Amazon EC2 nodes. The dataset consists of 11,175,629 logs and 16,838 labeled anomaly samples, which were provided by domain experts.
- *Ray* is a production dataset obtained through the Ant Group. Due to confidentiality concerns, obtaining production logs from companies is often challenging. However, we were fortunate to obtain a dataset of Ray logs from an industrial environment with the assistance of domain experts at Ant Group. This dataset includes 4,308,687 logs and 215,679 labeled anomaly samples, which were provided by domain experts.

We begin by sorting all datasets in chronological order and then use the method described in (Lou et al., 2010) to generate log sequences. A log sequence was considered as an anomaly if it contained any anomaly logs. These sequences are then shuffled. The data was partitioned into three segments: 60% for model training, 20% for verification, and 20% for testing. Notably, KAD does not necessitate model training; therefore, no training dataset was required. In contrast, techniques that require training had all anomalies removed from their dataset to focus on learning normal patterns and detecting anomalies. The parameters for all techniques were meticulously fine-tuned to ensure optimal performance.

To mitigate the influence of random variations, each technique was executed 30 times. The average results of these repetitions were then reported.

Baseline techniques We use four state-of-the-art anomaly detection techniques as the baselines: *DeepLog*, *LogAnomaly*, *Logsy*, and *CNN*. These techniques follow a similar procedure: first extracting a log event from each log, and then performing anomaly detection on the log event sequence.

- *DeepLog* (Du et al., 2017): This is the first work to employ long short-term memory (LSTM) (LeCun et al., 2015) for log anomaly detection. It is also the first work to detect anomalies in a forecasting-based fashion, which is widely used in many follow-up studies.
- *LogAnomaly* (Meng et al., 2019): To further consider the semantic information of logs, LogAnomaly is proposed. Specifically, they proposed template2Vec to represent the vector of words in log events by considering the synonyms and antonyms therein. Similarly, LogAnomaly adopts forecasting-based anomaly detection with an LSTM model.
- *Logsy* (Nedelkoski et al., 2020): This is the first work utilizing the Transformer to detect anomalies in logs. Specifically, Logsy is a classification-based method to learn log representations in a way to better distinguish between normal data from the system and abnormal samples from auxiliary log datasets. The auxiliary datasets help learn a better representation of the normal data while regularizing against overfitting.
- *CNN* (Lu et al., 2018): This is the first work to explore the effectiveness of convolutional neural networks (CNNs) in this field. This technique involved constructing log event sequences via identifier-based partitioning and introducing logkey2vec for embedding, which facilitates the convolution calculations needed for a CNN. They applied various convolutional layers, concatenating their outputs for input into a fully connected layer to generate predictions.

Metrics We use the commonly used *precision*, *recall*, and *F1-score* to evaluate the effectiveness of anomaly detection. Let *TP* and *FN* denote the number of anomalous samples that are correctly and incorrectly predicted, respectively, and *TN* and *FP* denote the number of normal samples that are correctly and incorrectly predicted, respectively. The metrics can be calculated as Eqs. (1)–(3).

- *Precision* measures the percentage of anomalous data out of all data identified as anomalies.

$$precision = \frac{TP}{TP + FP} \tag{1}$$

- *Recall* measures the percentage of the anomalous data that are correctly identified as anomalies.

$$recall = \frac{TP}{TP + FN} \tag{2}$$

- *F1-score* is the harmonic mean of precision and recall.

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} \quad (3)$$

4.3 Results and Analysis

4.3.1 RQ1: Viability

Table 1 shows the KDL knowledge in datasets. Table 1 presents an overview of the KDL knowledge extracted from two datasets—Hadoop and Ray, including the dataset, knowledge ID, anomaly description, knowledge annotation, anomaly root cause, and symptom. Note that each entry in the table represents a knowledge.

Table 1 details a total of 28 instances of KDL knowledge. Among them, ten are associated with the Hadoop dataset, while the remaining 18 are associated with the Ray dataset. Fifty percent of the knowledge reveals functional anomalies in distributed systems. For example, entry 2 describes anomalies occurring during system interactions; entries 4 and 14 describe anomalies related to logical design and coding within the system's internal architecture; and entries 6–9, 21–22, and 24–28 describe anomalies related to system configuration. The other 50% of the knowledge describes non-functional anomalies. For example, entries 5, 11–12, 15–16, 20, and 23 describe performance-related anomalies, and entries 1, 3, 10, 13, and 17–19 describe communication anomalies within the systems.

Except for entries 5 and 28, all the other knowledge employs semantic sets for their descriptions. For example, entry 1 utilizes the semantic sets "HeartbeatLost" and "TaskFail" to detail the root cause and symptom of the anomaly "KilledTaskRun," respectively.

In summary, KDL serves to define functional and non-functional anomalies in distributed systems. It effectively enhances the expressiveness of the knowledge, reducing the dependency on domain-specific expertise, primarily through the semantic sets.

4.3.2 RQ2: Effectiveness

Table 2 shows the effectiveness evaluation results with the two datasets, in which values in bold indicate the best score for precision, recall, and F1-score. KAD demonstrates high precision (1 in HDFS, 1 in Ray), recall (1 in HDFS, 0.9 in Ray), and F1-score (1 in HDFS, 0.95 in Ray) across all datasets. KAD has a number of knowledge available for anomaly detection, contributing to its high precision (1 on two datasets). However, the recall of KAD on Ray (0.9) is slightly lower compared to HDFS (1) due to an anomaly ("raylet timeout due to insufficient node CPU resources") that is not covered by the existing knowledge in Table 1.

KAD outperforms four other log-based anomaly detection techniques (DeepLog, LogAnomaly, Logsy, and CNN). On average, KAD achieves an improvement of 4.44% in precision, 8.4% in recall, and 6.38% in F1-score in HDFS, and an average improvement of 63.27% in precision, 80% in recall, and 78.4% in F1-score in Ray. This is because that KAD can detect not only functional anomalies but also non-functional anomalies that are rarely identified by other baseline techniques. For example, anomalies represented by entry 3 in HDFS and entries 12 and 13 in Ray, which go undetected by baseline methods, are detected by KAD.

Furthermore, we observe that the industrial dataset Ray poses more challenges to the four baseline techniques. This is because the logs and anomalies in Ray are more complex compared to the public dataset HDFS. However, KAD leverages domain knowledge

Table 1 KDL knowledge in datasets

Dataset	ID	Anomaly description	Anomaly	Root cause	Symptom
HDFS	1	Task fail due to heartbeat lost	tags=TaskFailDueHBLost	label=HeartbeatLost	label=TaskFail
	2	A killed task continued to be running in both JobTracker and TaskTracker	tags=KilledTaskRun	label=TaskRun	label=TaskKillFail
	3	Ask more than one node to replicate the same block to a single node simultaneously	tags=ReplicateRepeat, $getCount(label = BlockReplicate) > 1$	label=BlockReplicate	label=ReplicateFail
	4	Write a block already existed	tags=WriteBlockExit	label=BlockExit	label=WriteFail
	5	Task JVM hang	tags=JVMHang	Task JVM for taskID=$taskID$ is unresponsive	Task $taskID$ failed with exit code 137
	6	Swap a JVM, but mark it as unknown	tags=SwapUnknownJVM	label=MarkJVMUnknown	label=JVMSwapFail
	7	Swap a JVM, and delete it immediately	tags=SwapDeleteJVM	label=DeleteJVM	label=JVMSwapFail
	8	Try to delete a data block when it is opened by a client	tags=DeleteOpenBlock	label=BlockOpen	label=DeleteBlockFail
	9	JVM inconsistent state	tags=JVMInconsist	label=JVMInconsist	label=JVMFail
	10	The pollForTaskWithClosed-Job call from a Jobtracker to a task tracker times out when a job completes	tags=TrackCallTimeOut	label=TrackCallTimeOut	label=JobCallFail
Ray	11	Actor OOM	tags=ActorOOM	label=ActorOOM	label=jobFail
	12	L1FO causes a job fail	tags=L1FOWorkerFail, timespan=90 s	label = L1FOStart => label = WorkerFail	label=JobFail
	13	Taint manager eviction deletes pod, resulting in node remove	tags=NodeRemoveDuePodDelete, timespan=100 s	Marking for deletion Pod *podName* || Marking for deletion Pod *podName*:*hostName*	label=NodeRemove
	14	Task fail causes Ray check fail	tags=RayCheckFail, node=*nodeID*	label=TaskFail	label=RayCheckFail
	15	OOM killer causes worker fail	tags=WorkerFailDueOOM	label=OOMKiller	label=WorkerFail
	16	JVM crash causes worker fail	tags=WorkerFailDueJVM	label=HsErrLog	label=WorkerFail

Table 1 (continued)

Dataset	ID	Anomaly description	Anomaly	Root cause	Symptom
	17	Actor died causing driver fail	tags=DriverFailDueActor	label=ActorDied	Driver of Job *jobID* failed. The actor *actorID* died unexpectedly before finishing
	18	Actor died causing task fail	tags=TaskFailDueActor	label=ActorDied	label=TaskFail
	19	Actor died due to node remove	tags=ActorfaildDueNodeRmove	label=NodeRemove	ActorDied
	20	Cluster resource shortage	tags=ResourcePending	label=AllocateResourceFail	label=JobFail
	21	Resource misconfiguration	tags=ResoureErrorOOM	userspace oom killer sending SIGTERM to process	label=JobFail
	22	TensorFlow configuration failure	tags=TFConfiguration	tensorflow. python. framework. errors. NotFoundError	JobFail
	23	Insufficient cluster scheduling resources	tags=ResourceError	label=AllocateResourceFail	label=JobFail
	24	ALPS configuration failure	tags=ALPSError	label =BulidExpError	label=JobFail
	25	System anomaly caused by py file configuration failure	tags=pyError	label =pyConfError	label=JobFail
	26	Configuration data inconsistency	tags=SampleConfError	label=SampleConfError	label=JobFail
	27	Inconsistent data fields	tags=SampleError	label=SampleError	label=LogException
	28	Dirty data in the sample stream	tags=DirtyData	tensorflow. python. framework. errors. InvalidArgumentError	Exception: class LogException: log-store without index config

Table 2 Effectiveness of different techniques in different datasets

Methods	HDFS			Ray		
	Precision	Recall	F1-score	Precision	Recall	F1-score
DeepLog	0.96	0.93	0.94	0.64	0.33	0.44
LogAnomaly	0.97	0.9	0.95	0.45	0.67	0.54
Logsy	0.95	0.87	0.9	0.62	0.5	0.55
CNN	0.95	0.99	0.97	0.74	0.5	0.6
KAD	**1**	**1**	**1**	**1**	**0.9**	**0.95**

to detect these anomalies, which enhances effectiveness and interpretability while dealing with various complex anomalies in Ray.

In summary, KAD is observed to be highly effective in detecting anomalies within different datasets compared with the four baselines.

4.3.3 RQ3: Efficiency

Table 3 presents the training and testing times for different techniques in the two datasets, in which values in bold indicate the minimum time required to train and test.

A notable observation is that KAD, unlike the other baseline techniques, necessitates no time for the training phase. This is because KAD's design philosophy eschews the model training process. The omission of the model training phase can help to reduce the need for substantial computational resources and time investment. Consequently, KAD effectively reduces the time and resource overhead associated with model training compared with the four baseline techniques. However, the construction of knowledge within KAD incurs some time overhead which depends on the expert's familiarity with the system. Specifically, the overhead is mainly spent on knowledge extraction and representation. For experts who are familiar with the system, their prior knowledge helps them to identify anomalous behaviors, thus significantly reducing the time of the knowledge construction process; otherwise, they have to take some time to comprehend the system's behavior and logs.

Considering the testing process, KAD is faster than the four baseline techniques, as it used log processing (Section 3.3) and the distributed computation methods during knowledge matching (Section 3.4). Specifically, KAD exhibits an on-average 70.73% decrease in testing time compared to the four baseline techniques for the HDFS dataset and an on-average 74.42% decrease in testing time compared to the four baseline techniques on the Ray dataset. This is because KAD employs the matching task to multiple compute nodes

Table 3 Training and testing times for different techniques

Methods	HDFS		Ray	
	Training (min)	Testing (s)	Training (min)	Testing (s)
DeepLog	37.8	32	45.6	47
LogAnomaly	42.3	44	51.2	55
Logsy	25.6	25	36.7	37
CNN	32.6	22	40.1	33
KAD	**0**	**9**	**0**	**11**

in parallel, which can increase the speed of testing. Thus, once the knowledge is obtained, KAD can detect anomalies in a shorter time compared to the four baseline techniques.

4.3.4 RQ4: Robustness

To assess the robustness of the selected methods, i.e., their effectiveness in the variation of datasets, additional logs are introduced into the original test set. These additions were made at proportions of 5% and 10%, resulting in the creation of four new test sets. We follow (Zhang et al., 2019) to synthesize new logs. Specifically, given a randomly sampled log event sequence in the testing data, we apply one of the following four noise injection strategies: randomly injecting a few pseudo log events (generated by trivial word addition/removal or synonym replacement) or deleting/shuffling/duplicating a few existing log events in the sequence.

Table 4 presents the effectiveness of the different techniques with the variation of datasets, in which values in bold indicate the best score for precision, recall, and F1-score. It can be observed that KAD achieves higher precision, recall, and F1-score than the other four techniques, across all four datasets. Notably, KAD has relatively stable performance across different datasets (with precision still remaining 1, recall ranging between 0.86 and 0.96, and F1-score ranging between 0.92 and 0.98). This is because the semantic sets in KAD can help to capture and adapt to the changes in logs.

In summary, KAD demonstrates higher effectiveness compared with the other baseline techniques when applied to the variation of datasets in distributed systems.

4.4 Threats to validity

One of the threats to the experiment is the accuracy of the knowledge. Indeed, incorrect knowledge may lead to the mismatch between the logs and the knowledge pieces written by KDL, and hence lead to wrong anomaly detection. To reduce the threat, we involved domain experts for their insights and feedback. In addition, we employed a validation process using verification datasets. These datasets served to validate the accuracy and reliability of the knowledge, thereby enhancing the reliability of our experimental results.

Another potential threat is the time required for constructing knowledge. Unlike the baseline techniques, KAD necessitates a certain degree of domain knowledge. However,

Table 4 Effectiveness of different techniques in the variation of datasets

Injection ratio	Methods	HDFS			Ray		
		Precision	Recall	F1-score	Precision	Recall	F1-score
5%	DeepLog	0.67	0.59	0.63	0.28	0.19	0.23
	LogAnomaly	0.71	0.62	0.66	0.31	0.21	0.25
	Logsy	0.58	0.47	0.52	0.19	0.12	0.15
	CNN	0.73	0.65	0.69	0.34	0.29	0.31
	KAD	**1**	**0.96**	**0.98**	**1**	**0.89**	**0.94**
10%	DeepLog	0.54	0.46	0.50	0.14	0.09	0.11
	LogAnomaly	0.6	0.51	0.55	0.19	0.13	0.15
	Logsy	0.39	0.24	0.30	0.08	0.05	0.06
	CNN	0.61	0.49	0.54	0.21	0.19	0.20
	KAD	**1**	**0.91**	**0.95**	**1**	**0.86**	**0.92**

the accuracy and reliability of the knowledge enhance the effectiveness and robustness of KAD's anomaly detection capabilities. This is crucial, especially in the context of industrial applications where accuracy is important. Despite the time required for knowledge construction, KAD demonstrates notable efficiency in the testing phase, achieving a precision rate of 100% in anomaly detection. The results highlights the value of the time and effort spent on knowledge construction.

The validity of experimental results may be threatened by the benchmarks. We only use datasets collected from HDFS and Ray systems. Although HDFS is a typical open-source project and Ray is a large-scale, real-world distributed system, the number of experimental systems is still limited.

Finally, the log dataset might have data imbalance issues, such as the number of normal logs is greater than the number of anomaly logs. This imbalance in the log dataset may lead to a trade-off between precision and recall, in which the model may reduce precision to improve recall excessively, or vice versa.

5 Related work

A significant amount of methods for log anomaly detection has been published in academia (Fu et al., 2023; Huang et al., 2023; Qi et al., 2023; Ma et al., 2023; Chen et al., 2023). The existing approaches are mainly divided into three categories: supervised learning methods, unsupervised learning methods, and deep learning methods.

Many supervised learning methods are applied to log-based anomaly detection. For example, Zhang and Sivasubramaniam (2007) applied a support vector machine (SVM) to detect anomalies using event logs. Farshchi et al. (2015) proposed a regression-based method to detect anomalies using logs in cloud systems. Breier and Branišová (2015) summarized some classical supervised classification models that are applied to log-based anomaly detection. However, obtaining system-specific labeled logs is costly and often practically infeasible.

Apart from supervised learning approaches, many unsupervised learning approaches have been proposed. For example, Lou et al. (2010) proposed invariant mining (IM) to mine the linear relationships among log events from log event count vectors. Those log sequences that violate the invariant relationship are considered as anomalies. Xu et al. (2009) constructed normal space and abnormal space of log event count matrix using principal component analysis to detect anomalies. He et al. (2018) designed clustering-based methods to identify problems of online service systems. Unsupervised learning approaches have the advantage that they do not require manual labels in the training set. However, they are not robust to variation logs, which significantly restricts their applicability in real-world practice.

The recent rise of deep learning methods has given a new solution for log-based anomaly detection. Du et al. (2017) used LSTM to forecast the next log event and then compare it with the current ground truth to detect anomalies. Vinayakumar et al. (2017) trained a stacked LSTM to model the operation logs of normal and anomaly events. However, their input of neural networks is the one-hot vector of log events. Thus it cannot cope with evolving log data, especially in the scenario when new log events appear.

A few studies have leveraged NLP techniques to analyze log data based on the idea that log is actually a natural language sequence. Zhang et al. (2016) proposed to use the LSTM model and TF-IDF weight to predict the anomaly logs. Bertero et al. (2017) used word2vec and traditional classifiers, like SVM and random forest, to check whether a log event is an anomaly or not. Similarly, Zhang et al. (2019) and Meng et al. (2019) incorporate pre-trained word vectors for learning a sequence of logs where they train an attention-based Bi-LSTM model. However, these approaches only focused on the granularity of log events rather than log sequences. They ignored the contextual information in log sequences.

Different from all the above methods, we focus on the formalization of knowledge to detect anomalies in distributed systems. Through the anomaly detection framework KAD, the interoperability of knowledge between different distributed systems is achieved, which further improves the performance and robustness of anomaly detection.

6 Conclusion

In this paper, we have proposed KAD, a knowledge formalization-based anomaly detection approach for distributed systems. KAD introduces a general knowledge description language—KDL, to describe complex anomaly knowledge in various distributed systems. KDL involves the semantic set to improve the expressive capabilities of knowledge. Based on KDL, KAD utilizes the BERT model to further improve the robustness of anomaly detection. In addition, KAD employs a constraint solver and a distributed scheduling computation method, enhancing the efficiency of anomaly detection. The evaluation of KAD on two benchmark distributed systems has shown its high effectiveness in anomaly detection compared to four baseline techniques. Its capability to detect anomalies across variant datasets also demonstrates its robustness.

In the future, we will evaluate KAD on a broader range of distributed systems. Furthermore, we will improve the intellectualization of KAD to generate KDL knowledge more time-saving in different distributed systems.

Author contribution Xinjie Wei initiated the project, wrote the manuscript, and conducted the experiment; Chang-ai Sun proposed the main idea, discussed the settings of the experiment, and made a revision of the paper; Xiao-Yi Zhang discussed the settings of the experiment and made a revision of the paper.

Funding This article is supported by the National Natural Science Foundation of China under Grant Nos. 62272037, 61872039, and 62302035, and CCF-Ant Research Fund.

Availability of data and materials HDFS is a public data set and can be obtained from (Du et al., 2017). Ray is a privacy data set that involves the privacy of a partner company (the Ant Group) and cannot be made public for the time being.

Code availability The code involves the privacy of a partner company (the Ant Group) and cannot be made public for the time being.

Declarations

Ethics approval This article does not contain any studies with human participants or animals performed by any of the authors.

Consent to participate Not applicable

Consent for publication The results/data/figures in this manuscript have not been published elsewhere, nor are they under consideration (from you or one of your contributing authors) by another publisher.

Competing interest The authors declare no competing interests.

References

Ali, A., Ali, A., Abaluof, H., et al. (2023). Moisture detection in tree trunks in semiarid lands using low-cost non-invasive capacitive sensors with statistical based anomaly detection approach. *Sensors, 23*(4), 21–31.

Apache Hadoop. (2023). Apache Hadoop Home. http://hadoop.apache.org/

Apache Spark. (2023). What is Apache Spark? http://spark.apache.org/

Bertero, C., Roy, M., Sauvanaud, C., et al. (2017). Experience report: Log mining using natural language processing and application to anomaly detection. In: *Proceedings of the 28th IEEE International Symposium on Software Reliability Engineering*, pp 351–360.

Breier, J., & Branišová, J. (2015). Anomaly detection from log files using data mining techniques. In: *Proceedings of the 2015 Information Science and Applications*, pp 449–457.

Chen, L., Dang, Q., Chen, M., et al. (2023). BertHTLG: Graph-based microservice anomaly detection through sentence-Bert enhancement. In: *Proceedings of the 2023 International Conference on Web Information Systems and Applications*, pp 427–439.

Devlin, J., Chang, M. W., Lee, K., et al. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of the 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pp 4171–4186.

Du, M., Li, F., Zheng, G., et al. (2017). DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp 1285–1298.

Farshchi, M., Schneider, J. G., Weber, I., et al. (2015). Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis. In: *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering*, pp 24–34.

Fu, Y., Yan, M., Xu, Z., et al. (2023). An empirical study of the impact of log parsers on the performance of log-based anomaly detection. *Empirical Software Engineering, 28*(1), 1–39.

Gómez, Á. L. P., Maimó, L. F., Celdrán, A. H., et al. (2023). SUSAN: A deep learning based anomaly detection framework for sustainable industry. *Sustainable Computing: Informatics and Systems, 37*(3), 834–842.

Haoming, L., & Yuguo, L. (2020). LogSpy: System log anomaly detection for distributed systems. In: *Proceedings of the 2020 International Conference on Artificial Intelligence and Computer Engineering*, pp 347–352.

He, P., Zhu, J., Zheng, Z., et al. (2017). Drain: An online log parsing approach with fixed depth tree. In: *Proceedings of the 2017 IEEE International Conference on Web Services*, pp 33–40.

He, S., Lin, Q., Lou, J. G., et al. (2018). Identifying impactful service system problems via log analysis. In: *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp 60–70.

Hidayati, J., Vamelia, R., Hammami, J., et al. (2023). Transparent distribution system design of halal beef supply chain. *Uncertain Supply Chain Management, 11*(1), 31–40.

Hogan, A., Blomqvist, E., Cochez, M., et al. (2021). Knowledge graphs. *ACM Computing Surveys, 54*(4), 1–37.

Hristov, M., Nenova, M., Iliev, G., et al. (2021). Integration of Splunk enterprise SIEM for DDoS attack detection in IoT. In: *Proceedings of the 20th IEEE International Symposium on Network Computing and Applications*, pp 1–5.

Huang, S., Liu, Y., Fung, C., et al. (2023). Improving log-based anomaly detection by pre-training hierarchical transformers. *IEEE Transactions on Computers, 72*(9), 2656–2667.

IBM. (2023). Ariel Query Language Guide. https://www.ibm.com/docs/en/SS42VS_7.4/pdf/b_qradar_aql.pdf

Le, V. H., & Zhang, H. (2022). Log-based anomaly detection with deep learning: How far are we? In: *Proceedings of the 44th international conference on software engineering*, pp 1356–1367.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521*(7553), 436–444.

Liang, E., Nishihara, R., Mika, S., et al. (2023). Ray. https://github.com/ray-project/ray

Lou, J. G., Fu, Q., Yang, S., et al. (2010). Mining invariants from console logs for system problem detection. In: *Proceedings of the 2010 USENIX Annual Technical Conference*, pp 24–37.

Lu, S., Wei, X., Li, Y., et al. (2018). Detecting anomaly in big data system logs using convolutional neural network. In: *Proceedings of the 16th IEEE Intlernational Conference on Dependable*, Autonomic and Secure Computing, pp 151–158.

Ma, X., Keung, J., He, P., et al. (2023). A semi-supervised approach for industrial anomaly detection via self-adaptive clustering. *IEEE Transactions on Industrial Informatics, 6*(2), 1–12.

Majeed, A., ur Rasool R, Ahmad F, et al. (2019). Near-miss situation based visual analysis of SIEM rules for real time network security monitoring. *Journal of Ambient Intelligence and Humanized Computing, 10*(7), 1509–1526.

Meng, W., Liu, Y., Zhu, Y., et al. (2019). LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In: *Proceedings of the 2019 International Joint Conference on Artificial Intelligence*, pp 4739–4745.

Moritz, P., Nishihara, R., Wang, S., et al. (2018). Ray: A distributed framework for emerging AI applications. In: *Proceedings of the 13th Operating Systems Design and Implementation*, pp 561–577.

Nedelkoski, S., Bogatinovski, J., Acker, A., et al. (2020). Self-attentive classification-based anomaly detection in unstructured logs. In: *Proceedings of the 2020 IEEE International Conference on Data Mining*, pp 1196–1201.

Qi, J., Luan, Z., Huang, S., et al. (2023). LogEncoder: Log-based contrastive representation learning for anomaly detection. *IEEE Transactions on Network and Service Management, 20*(2), 1378–1391.

Splunk Enterprise. (2023). Search Tutorial-Use the search language. https://docs.splunk.com/Documentation/Splunk/9.1.1/SearchTutorial/Usethesearchlanguage

Tietz, V., & Annighoefer, B. (2022). A formally defined and formally provable EBNF-based constraint language for use in qualifiable software. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pp 862–871.

Vinayakumar, R., Soman, K., & Poornachandran, P. (2017). Long short-term memory based operation log anomaly detection. In: *Proceedings of the 2017 International Conference on Advances in Computing, Communications and Informatics*, pp 236–242.

Xu, W., Huang, L., Fox, A., et al. (2009). Detecting large-scale system problems by mining console logs. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pp 117–132.

Zhang, K., Xu, J., Min, M. R., et al. (2016). Automated it system failure prediction: A deep learning approach. In: *Proceedings of the 2016 IEEE International Conference on Big Data*, pp 1291–1300.

Zhang, X., Xu, Y., Lin, Q., et al. (2019). Robust log-based anomaly detection on unstable log data. In: *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp 807–817.

Zhang, Y., & Sivasubramaniam, A. (2007). Failure prediction in IBM BlueGene/L event logs. In: *Proceedings of the 7th International Conference on Data Mining*, pp 583–588.