From Point-wise to Group-wise: A Fast and Accurate Microservice Trace Anomaly Detection Approach

Zhe Xie
Tsinghua University
China

Changhua Pei*
CNIC, CAS
China

Wanxue Li
eBay Inc.
China

Huai Jiang
eBay Inc.
China

Liangfei Su
eBay Inc.
China

Jianhui Li
CNIC, CAS
China

Gaogang Xie
CNIC, CAS
China

Dan Pei
Tsinghua University
China

ABSTRACT

As Internet applications continue to scale up, microservice architecture has become increasingly popular due to its flexibility and logical structure. Anomaly detection in traces that record inter-microservice invocations is essential for diagnosing system failures. Deep learning-based approaches allow for accurate modeling of structural features (*i.e.*, call paths) and latency features (*i.e.*, call response time), which can determine the anomaly of a particular trace sample. However, the point-wise manner employed by these methods results in substantial system detection overhead and impracticality, given the massive volume of traces (billion-level). Furthermore, the point-wise approach lacks high-level information, as identical sub-structures across multiple traces may be encoded differently. In this paper, we introduce the first Group-wise Trace anomaly detection algorithm, named **GTrace**. This method categorizes the traces into distinct groups based on their shared sub-structure, such as the entire tree or sub-tree structure. A group-wise Variational AutoEncoder (VAE) is then employed to obtain structural representations. Moreover, the innovative "predicting latency with structure" learning paradigm facilitates the association between the grouped structure and the latency distribution within each group. With the group-wise design, representation caching, and batched inference strategies can be implemented, which significantly reduces the burden of detection on the system. Our comprehensive evaluation reveals that GTrace outperforms state-of-the-art methods in both performances (2.64% to 195.45% improvement in AUC metrics and 2.31% to 40.92% improvement in best F-Score) and efficiency (21.9x to 28.2x speedup). We have deployed and assessed the proposed algorithm on eBay's microservices cluster, and our code is available at https://github.com/NetManAIOps/GTrace.git.

CCS CONCEPTS

• **Networks** → **Network services**; • **Computing methodologies** → *Artificial intelligence.*

KEYWORDS

microservice trace, anomaly detection, variational autoencoder

ACM Reference Format:
Zhe Xie, Changhua Pei, Wanxue Li, Huai Jiang, Liangfei Su, Jianhui Li, Gaogang Xie, and Dan Pei. 2023. From Point-wise to Group-wise: A Fast and Accurate Microservice Trace Anomaly Detection Approach. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3611643.3613861

1 INTRODUCTION

Nowadays, with the expansion of Internet applications and the number of users, there is an urgent need for software services to be developed rapidly and scaled dynamically. Microservice architecture, with the advantages of fast development, easy deployment, and scalability, has been widely used in the development of Internet systems [7, 12, 15, 17]. In spite of this, the microservice architecture also poses great challenges for failure diagnosis due to the huge number of candidates and complicated dependencies between them. The first step for failure diagnosis is to accurately and efficiently detect the anomalies. A common way to detect anomalies in large microservices systems is to detect anomalous traces. A microservice trace is a tree structure that consists of a number of spans representing the invocation relationships between the APIs of the microservices, which records the execution process of an external request [12]. Since the traces contain detailed information about the requests such as the invocation paths and the internal latency of the spans, anomaly detection on microservice traces has become a hot topic in industrial companies [19, 29, 30], which is also the focus of this paper.

However, there are a number of challenges in trace anomaly detection in microservices systems:

- Large number of services and complex trace structures. Large microservice systems usually consist of many services, and the number of trace structures may even be much larger [29]. This poses a great challenge for structural modeling.
- Large variation in latency distributions. Due to the flexibility of the microservice system, the same service may have a large difference in their downstream tree structures, resulting in different latency distributions [19].
- In eBay, millions of traces are produced every minute. In order to detect anomalies in this volume of traces, the trace anomaly detection algorithm must be efficient.

*Corresponding author.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613861

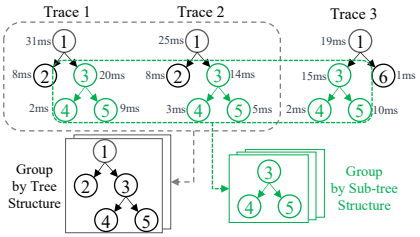


Figure 1: A motivation example of the group-wise model. Traces with the same tree structure can be grouped together. The sub-tree of traces with the same structure can also be grouped together. Grouped items share the same structure but with different latency values.

- A failure tends to result in a large number of anomalous traces with different structures. Trace-level anomaly outputs may not conducive to efficiently helping the operators deal with the failures [14].

Limited by the huge cost of manual labeling, existing trace anomaly detection approaches are mostly unsupervised. Existing trace anomaly detection approaches can be divided into two categories: statistic-based and model-based. **Statistic-based approaches** [9, 12, 17, 21] group a set of trace nodes with the same structural features and model their distributions of latency features using statistical methods. These statistics-based methods run efficiently, but they have difficulties detecting new trace structures. Some coarse-grained grouping strategies will make them less accurate in modeling latency [5, 19]. **Model-based approaches** [19, 25, 27, 29, 30] employ deep learning to model the trace features. The use of deep learning enables accurate modeling of complex trace features. But it significantly increases the detection overhead since a model inference is required for each trace.

Therefore, neither statistic-based approaches nor model-based approaches can satisfy the above-mentioned four challenges (refer to Sec. 2.2). We point out the main culprit of the high detection burden is the point-wise detection manner. Though there are shared structures between different traces, as illustrated in Fig. 1, the point-wise trace detection model has to conduct the complete forwarding process trace-by-trace without reusing. Moreover, the point-wise manner may encode the same structure shard by traces into different representations. This unstable encoding mechanism of trace structure makes it hard to learn high-level structural information and the intrinsic latency correlation, which has been confirmed through statistics: the latency of traces from the same structure or sub-tree can be similar.

In this paper, we propose **GTrace**, a **G**roup-wise **Trace** anomaly detection algorithm. As illustrated in Fig. 1, the main idea behind GTrace is as follows: For a batch of traces, we first categorize the traces into distinct groups based on their shared structure, such as the entire tree or sub-tree structure. By carefully designed grouping strategy (Sec. 3.1), we obtain the underlying structure of these traces. For each group, we use a group-wise VAE (Sec. 3.2) to reassemble the sub-structures together to get the node-wise and graph-wise representations, which are used in the following detection and visualization. On the one hand, the computational overhead can be significantly reduced by encoding sharing (Sec. 3.3). Combined with the dynamic programming strategy and graph merging in batched trace inference (Sec. 3.4), the computational overhead is further reduced. On the other hand, detection accuracy can be enhanced with group-wise VAE, which models trace latency at a higher level by learning the relationship between underlying structure and latency distribution across traces within the group, effectively mitigating the issue of overfitting latency for specific traces commonly found in point-wise models.

However, an important fact is ignored in the above process. Except for structure features, there are also latency features, which can not be decoupled and reassembled for the existing model-based methods. To address this problem, we exploit the relationship between structural and latency features in traces and propose a novel group-wise VAE model which "predicts latency with structure". At last, to make our model more actionable for failure diagnosis, we develop a novel visualization tool to present the anomalies in visualized graphs. Both the visualization tool and detection algorithm of GTrace have been deployed on eBay's real-world system. Our evaluation of the real-world dataset reveals that GTrace outperforms state-of-the-art methods in both performances (2.64% to 195.45% improvement in AUC metrics and 2.31% to 40.92% improvement in best F-Score) and efficiency (21.9x to 28.2x speedup). The main contributions of this paper are as follows:

- We propose **GTrace**, the first **G**roup-wise **Trace** anomaly detection approach. The novel "predicting latency with structure" learning paradigm facilitates trace grouping, group-wise VAE modeling, and ultimately, inference acceleration. This approach results in a method that is both fast and accurate.
- A batch of caching and reusing techniques (refer to Sec. 3.4), including a dynamic programming strategy, a node&graph caching algorithm, and a merged graph, are proposed to reduce the computational overhead of anomaly detection significantly.
- To provide understandable feedback on detection results, we develop a novel visualization tool for the trace anomalies based on the detection results and graph encodings. GTrace is deployed on eBay's cluster and visualizations are demonstrated through case studies.

2 PRELIMINARY AND MOTIVATION

2.1 Traces and Anomalies

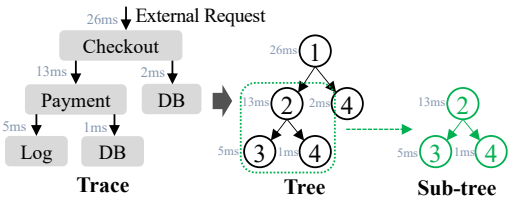


Figure 2: Example of a trace, its tree, and a sub-tree. APIs are mapped to their IDs in tree nodes.

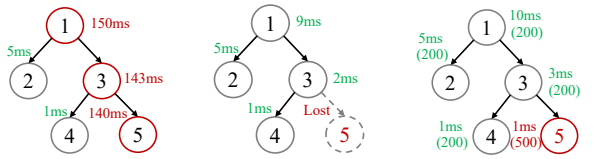
2.1.1 Microservice and API. A microservice system consists of several microservices that communicate with each other by calling the microservice's API. A microservice usually contains multiple APIs

(operations). In this paper, we use a unique numeric ID (API ID) to represent each API in the system.

2.1.2 Trace and Span. A trace records a set of spans representing the invocation relationships between the APIs of the microservices, which are produced by a single external request of a microservices system (see Fig. 2). A span refers to an API (A) invoking another API (B) under this external request, with corresponding API ID, status code, and performance metric such as call latency.

2.1.3 Tree and Sub-tree. Traces can be modeled as trees (see Fig. 2), where each node represents a span (labeled with its API ID), and each edge represents a call between spans. A tree contains *structural features* and *latency features*. Structural features include the structural information in a trace, such as call relationships and status codes. Latency features include the call latency of each span. The tree structure is an abstraction of traces. A tree structure can be shared by multiple traces. For each node (span), itself and its downstream calls form a sub-tree (see Fig. 1).

2.1.4 Node Sequence. By sorting the tree according to certain rules, we can get the node sequence corresponding to the tree [19, 22]. We use DFS (depth-first-search) to obtain the node sequence and sort the child nodes using the call timestamp in DFS.



(a) Latency **(b) Structure (Drop)** **(c) Structure (Status)**

Figure 3: Anomaly Types

2.1.5 Anomaly Types. As previously defined, trace features can be categorized into structural features and latency features. Accordingly, trace anomalies can be classified into **structural anomalies** and **latency anomalies**, following existing work [5, 19]. Latency anomalies are the anomalies of latency features that are usually caused by network latency or service saturation, etc. Latency anomalies can be defined at **node level** or **trace level**, referring to latency anomalies that occur on a span or in an entire trace, respectively. Structural anomalies are trace-level anomalies of structural features, such as missing spans in a trace or an abnormal return status code, which may be caused by network packet loss, bad software updates, etc.

2.2 Limitation of Existing Trace Anomaly Detection

As previous mentions, existing trace anomaly detection can be classified into statistic-based and model-based. Statistics-based approaches have been reported in the literature to have limitations in detection accuracy and scalability for dynamic systems [5, 19, 25].

Therefore, quite a number of recent studies employ model-based approaches. Fig. 4 illustrates the general process in model-based trace anomaly detection. Model-based approaches first encode both structural and latency features from traces into node sequences,

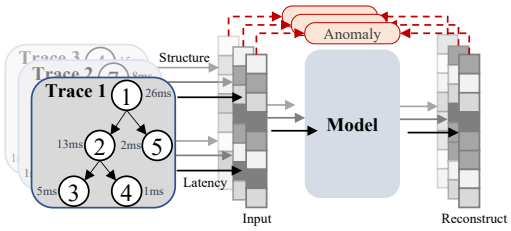


Figure 4: General process of model-based approaches. The anomaly detection is usually point-wise.

graphs, trace vectors, etc, which are then input to a certain model. By comparing the model reconstructions with the inputs, it is possible to determine whether a trace is abnormal or not. However, existing model-based approaches have the following limitations:

- **Limitation 1: Low Scalability**: Existing approaches encode each trace individually, leading to a substantial computational cost. Group strategies may not be applicable in this context because each trace's structural and latency information is combined into a single encoding. As demonstrated in Figure 1, although structures can be shared across different traces, the latency information remains unique and cannot be grouped.
- **Limitation 2: Ignoring High-Level Information**: Current approaches do not take into account the shared high-level information between different traces. Although some traces possess identical sub-trees, their shared sub-trees are encoded differently. This prevents the model from fully leveraging the commonalities between various traces.

2.3 Key Designs of GTrace

To address the limitations in model-based approaches, we propose GTrace featuring three key designs:

- Design 1: We propose the "predict latency with structure" strategy to predict latency distribution using structural features as input. Decoupling latency features from structural features allows for the implementation of a grouping strategy and further supports the design of group-wise models (refer to **Group-wise VAE in Section 3.2**).
- Design 2: We select the most suitable grouping strategy to obtain grouped structures as input, enabling shared encoding between traces. The grouped structure retains high-level information among traces that exhibit similar behaviors, such as trace latency (refer to **Grouping Strategy in Section 3.1**).
- Design 3: Utilizing the group-wise model and an appropriate grouping strategy, we also incorporate dynamic programming with cache and merged graph inference techniques to further enhance the inference efficiency of our proposed model, GTrace (refer to **Inference Acceleration in Section 3.4**).

3 METHODOLOGY

Figure 5 illustrates the training process of GTrace. Firstly, it receives the grouped structure as input through the grouping strategy. Following this step, a Group-wise VAE (Variational Autoencoder) is used for trace modeling. The Group-wise VAE is divided

into two parts: Graph-wise and Node-wise, where each part contains an encoder and a decoder. In the sections below, we will provide a detailed breakdown of each component, including the grouping strategy, group-wise VAE, encoder & decoder, as well as the inference acceleration.

3.1 Grouping Strategy

The purpose of the grouping strategy is to obtain a grouped structure from a set of traces to be used as input to the group-wise model. As previously mentioned, the grouping strategy plays a vital role in both detection accuracy and resource overhead reduction, which is crucial for addressing the aforementioned limitations. A more effective group strategy should fulfill the following requirements:

- It should aim to minimize the training and inference cost as much as possible.
- It should identify the most appropriate granularity, allowing traces within a group to exhibit similar behavior. In our scenario, this means that grouped sub-structures should ideally share the same key performance, *i.e.*, latency distribution.
- The grouping strategy should also take into account the available data for training purposes.

Clearly, if the grouping strategy is too coarse-grained, it may satisfy efficiency requirements but compromise performance. This issue arises because traces classified into the same group do not share the same performance (latency) distribution, making learning patterns within the group difficult. Additionally, an overly coarse-grained approach results in insufficient training samples for the model, hindering convergence. On the other hand, a too-fine-grained strategy cannot achieve a reduction in training and inference computational costs.

Table 1: Empirical Study on Grouping Strategies

Group Strategy	Description	Count	$\overline{\sigma(\log(lat))}$
None	No group	**1**	1.96
API	Group by API ID	413	0.48
STV	Group by stv [19]	51373	0.20
Tree	Group by tree [12]	40370	0.24
Sub-tree	Group by sub-tree	3311	**0.18**

To this end, we empirically study different group strategies with traces collected from an online large-scale microservices system. Following [6], we assume that the latency in traces follows a mixed Log-Normal distribution. In Tab. 1, we demonstrate several candidates of group strategies along with their count and $\overline{\sigma(\log(lat))}$, where σ denotes the standard deviation. The smaller σ implies that there is less deviation in modeling latency with a shared structure. Thus, the group-wise model will be more accurate. The count is used to denote the number of groups under different strategies. A large number of groups will result in an augmented training and inference overhead of the model. In Tab. 1, grouping by sub-tree has the smallest average σ value with a reasonable count, which shows its feasibility in our grouped inference task. Therefore, we design the group-wise VAE model based on sub-tree grouping.

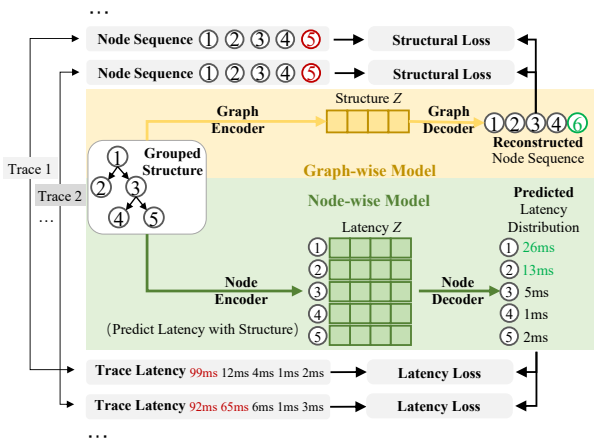


Figure 5: Training process of GTrace. Latency Zs are inferred through the "predict latency with structure" paradigm. Structural anomalies are detected through input node sequence and reconstructed sequence. Latency anomalies are detected through input latency and predicted latency.

3.2 Group-wise VAE

3.2.1 Overview. As depicted in Fig. 5, the proposed group-wise approach employs a Variational Autoencoder (VAE)-based model. VAE is widely used in modeling feature distributions, which infers a latent variable z by the inference model (the encoder) with the input features. The latent variable z is then decoded by the generative model (the decoder) to reconstruct the normal patterns of the features. To realize group-wise trace modeling, the grouped structures are used as model input. Structural features are reconstructed in the form of node sequence (Sec. 2.1.4) through the graph-wise model. Latency features are output by the node-wise model in a novel *"predict latency with structure"* manner.

3.2.2 Model Details. Considering the different nature of structural and latency features, the model is divided into two parts, **graph-wise** and **node-wise**, respectively modeling the structural features of the full graph (a trace tree) and the latency features of a node (root node of a sub-tree).

(1) The **node-wise** model encodes a grouped structure into node-level latency Zs through a node encoder and then decodes them into node-level latency distribution through a decoder. To realize "predict latency with structure", we employ Tree-LSTM (TL) [1] in node encoder and decoder, which is suitable for sub-tree modeling in traces and encoding sharing between sub-trees (see Sec. 3.3).

(2) The **graph-wise** model encodes a grouped structure into a graph-level structure Z through the graph encoder and decodes it into a *node sequence*. Since the node sequence has a fixed order (Sec. 2.1.4), reconstructing the node sequence avoids complicated graph matching in [20]. Structural anomalies can be found by comparing the input node sequences with the decoded sequence. We use another TL to encode the grouped structure and a graph pooling layer as the graph encoder. To decode a node sequence from the graph-level Z, we employ a Multi-Layer Perceptron (MLP) as the decoder. Tree-LSTM is not used here because the input to the decoder is not node-level.

Before the VAE model, we encode the trace features into vectors. Tree edges are encoded as adjacency matrices. Other structural features (including API IDs and status codes) are encoded into a feature matrix and then embedded with an embedding layer. For the latency features, we use the z-score standardization to normalize the latency values according to the API IDs of nodes.

3.3 Tree-LSTM Based Encoder and Decoder

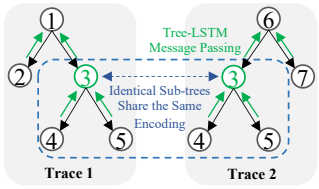


Figure 6: Message passing in Tree-LSTM used in graph encoder, node encoder, and node decoder. The sub-trees with identical structures share the same encoding.

In group-wise VAE, we build the graph encoder, node encoder, and node decoder upon the foundation on Tree-LSTM (TL) [1], which is a special variant of LSTM that can be applied on trees or DAGs. We choose to utilize TL for several reasons:

- TL supports the **sharing of encodings** for identical sub-tree structures, enabling grouping at the sub-tree level. Since messages pass from the leaf nodes (Fig. 6), the encoding of two sub-trees will be the same if they have the same structure.
- Tree-LSTM is highly suitable for sub-tree representation and modeling. This allows efficient and effective capturing of the hierarchical structure within grouped structures. In TL, messages are passed from leaf nodes level-by-level and aggregated to the root node. In this way, node-level latency distributions are predicted through the information passed from the sub-tree nodes.
- The unique message-passing mechanism enables the encoding of different sub-trees to be reused through dynamic programming (Sec. 3.4.1), significantly reducing the computational overhead while maintaining the model's accuracy.

Thus, thanks to TL, sub-trees in the *same group* can share their encoding. Furthermore, the encodings between *different groups* can also be shared during model inference with this unique message-passing mechanism. We will detail this process in Sec. 3.4.

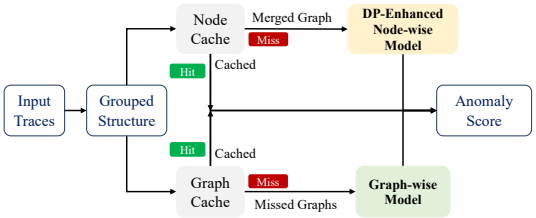


Figure 7: Process of inference acceleration, where DP means dynamic programming.

3.4 Inference Acceleration with Cache

In GTrace, we propose two techniques, *dynamic programming inference* and *merged graph*, to further reduce the overhead in model inference. The overall process of inference acceleration is depicted in Fig. 7.

3.4.1 Dynamic Programming Inference with Cache. With group-wise modeling, many repeated model inferences have already been avoided. However, the problem of group number inflation and time-consuming model inference still exists. Based on the previously mentioned message-passing strategy in Tree-LSTM, we design dynamic programming (DP)-based inference for the node-wise model. The DP inference works with a *node cache* and a *graph cache*.

The **node cache** stores the node-wise encodings in group-wise VAE. The messages are passed in topological order from the leaves to the root in the TL model. This means we can reuse the node representations with the same sub-tree structure with the DP strategy, *i.e.* use the sub-tree information in the cache to reduce duplicate computations further. To store the enormous amount of node data and the associations between these nodes (to implement DP), we propose a cache named **Trace Cache Tree (TCT)** to dynamically and flexibly store the node representations with reasonable resources. TCT is a tree with nodes managed by an LRU cache, which stores node representations belonging to several traces. Like the general cache, TCT supports *insert* and *query* operations. An example of TCT is depicted in Fig. 8.

Insert operation:

- For each node in a trace, encode its sub-tree and create a new TCT node with this encoding *node_key*.
- Insert the created TCT nodes that are not in the cache to TCT in an LRU manner.
- Add edges between the created TCT nodes according to their relationship in the original traces.

Query operation returns a subgraph G_i' for a trace G_i, which includes the missed nodes V_{miss} and data nodes V_{data}:

- Encode sub-trees for each node in G_i and their *node_key*. Add the missed nodes to $V(G_i')$.
- Add the children of missed nodes to $V(G_i')$ as data nodes V_{data}, whose encodings are needed to be loaded from the cache to the subgraph G_i' before dynamic programming inference.
- Add edges between the nodes in G_i' according to their edges in TCT.

We use **graph cache** to store the graph-level structural encodings (*e.g.*, structure Z). The graph cache is a standard LRU cache, which uses the tree structure of a whole trace as the key (*graph_key*).

3.4.2 Batched Inference with Merged Graph. DP achieves acceleration for a single group through encoding sharing. In fact, we can merge a batch of groups together as a merged graph for batched inference, which can take full advantage of the parallel performance of hardware. In the group-wise model, we propose the merged graph further to accelerate the node-wise model inference for batch traces. A merged graph G_{merge} is essentially a graph containing all the missed nodes and data nodes in **a batch of traces** (see Fig. 8), which is also a subgraph of TCT. The results of performing node-wise model inference on the merged graph are equivalent to the results on the original traces. Therefore, we only need to perform

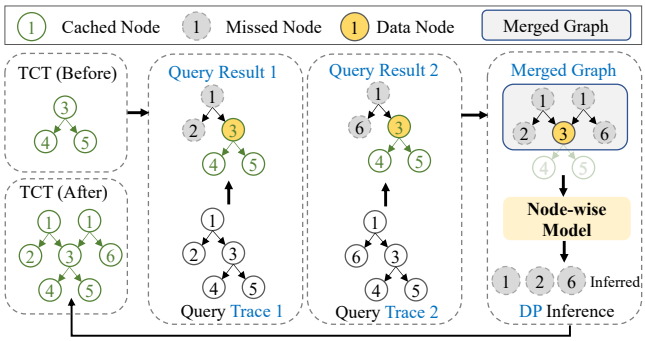


Figure 8: Example of Trace Cache Tree (TCT) and merged graph. Data nodes are cached nodes, but they contain encodings that are used in dynamic programming.

model inference on this merged graph with much less computational effort, which enables further speed improvement in model inference. With these two LRU caches and the merged graph, the group-wise representations can be effectively inferred with reasonable memory usage without any changes to the model in Fig. 5.

3.5 Loss Function and Anomaly Score

3.5.1 Loss Function. The Evidence Lower-Bound (ELBO) $\mathcal{L}_{\phi,\theta}(G)$ is a widely used optimization objective in the VAE model [16]. It is also used as the optimization objective in GTrace. Its ELBO is derived as follows:

$$\begin{aligned}\log p_\theta(G) &\geq \mathcal{L}_{\phi,\theta}(G) \\ &= \mathbb{E}_{Z\sim q_\phi(Z|G)}[\log p_\theta(G|Z)] - \mathrm{KL}(q_\phi(Z|G)||p(Z))\end{aligned} \tag{1}$$

where $p_\theta(Z) = \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ is the prior distributions of the latent variables Z. During training, sampling and backpropagation of the latent variables are performed using the reparameterization trick. The parameters θ and ϕ are optimized by maximizing the optimization objective $\mathcal{L}_{\phi,\theta}(G)$ defined in Eq. (1). The first term of $\mathcal{L}_{\phi,\theta}(G)$ contains the structural and latency loss in Fig. 5, and the second term is the KL divergence term of the latent variables in VAE.

3.5.2 Anomaly Score. The group-wise VAE models the normal patterns of trace structure and latency features, with the input structural features. The *anomaly score* reflects the degree of the anomaly of the test sample [4]. It can be obtained by calculating the probability of a trace $\boldsymbol{G}$ with the features under the distribution in its normal pattern (with group-wise VAE). A common practice of calculating $p(\boldsymbol{G})$ is Monte Carlo importance sampling [23]. An approximate estimate of the likelihood value can be computed with:

$$p_\theta(G) \approx \frac{1}{n_Z}\sum_{i=1}^{n_Z}\frac{p_\theta(G|Z^{(i)})p_\theta(Z^{(i)})}{q_\phi(Z^{(i)}|G)} \tag{2}$$

where n_z is the number of sampled latent variables, and $Z^{(i)}$ denote the i-th sample of Z. In GTrace, we use the negative log-likelihood (NLL) $-\log p_\theta(G)$ as the anomaly score. The test sample is considered anomalous when the NLL value is significantly higher than the normal NLL value.

In general, latency anomalies occur only in some of the trace nodes. In order to provide more accurate detection results, we detect node-level latency anomalies. We calculate the likelihood $L_{T,k,\theta}$ for the latency T_k of node k and the trace-level structural likelihood $L_{S,\theta}$ separately as follows:

$$p_\theta(X) \approx \frac{1}{n_Z}\sum_{i=1}^{n_Z}\frac{p_\theta(X|Z_S^{(i)})p_\theta(Z_S^{(i)})}{q_\phi(Z_S^{(i)}|X,E)} \stackrel{\text{def}}{=} L_{S,\theta} \tag{3}$$

$$p_\theta(T_k) \approx \frac{1}{n_Z}\sum_{i=1}^{n_Z}\frac{p_\theta(T_k|Z_T^{(i)})p_\theta(Z_T^{(i)})}{q_\phi(Z_T^{(i)}|X,E)} \stackrel{\text{def}}{=} L_{T,k,\theta} \tag{4}$$

where Z_S and Z_T denotes the structure Z and latency Z respectively. X denotes the node sequence and E denotes the adjacency matrix of the tree.

4 EVALUATION

In this section, we focus on the evaluation to answer the following research questions:

RQ1: How does GTrace perform on accurately detect anomalies?

RQ2: Do "predicting latency with structure" and "splitting node-wise and graph-wise models" prove to be effective in accurately detecting trace anomalies?

RQ3: How does the group-wise model work in modeling the latency?

RQ4: Does GTrace really time efficient? How much do the cache and merged graph accelerate anomaly detection?

4.1 Datasets

To comprehensively evaluate the performance of the proposed algorithm, 2 datasets are collected and produced, including one **real-world** dataset collected from eBay's microservices system and one dataset collected from a **testbed** with fault injection. Basic information about these 2 datasets is shown in Tab. 2.

Table 2: Basic Information about Datasets

Dataset	#Traces	P99 Latency	P99 #Spans	P99 Depth
$\mathcal{A}$	125k	7580ms	90	10
$\mathcal{B}$	140k	263ms	96	4

4.1.1 Real-world Dataset from eBay. To evaluate the performance of the proposed GTrace algorithm for detecting real anomalies in a real microservice system, the dataset $\mathcal{A}$, which contains real anomalies, is collected from eBay's microservices system. The system contains a total of 314 microservices and 1487 APIs. The dataset contains 5 hours of trace data at the normal time and 2 hours of test data at the time of failure. We manually label the traces during the failure, which include 253 traces with structure anomalies and 1417 traces with latency anomalies.

4.1.2 Dataset from Testbed with Fault Injection. Dataset $\mathcal{B}$ is collected from a testbed built based on the open-source microservices system *microservice-demo*[1]. We use Kubernetes to manage the microservices system, which contains 11 microservices and 65 APIs.

[1]https://github.com/GoogleCloudPlatform/microservices-demo.git

In order to obtain anomalous trace data, several services are randomly selected and injected into different types of faults. The types of injected faults include network latency, network packet loss, and CPU stress of pods.

4.2 Baselines

In order to comprehensively evaluate the anomaly detection performance of the model, the proposed model is compared with various existing baseline models that can be used for trace anomaly detection.[2]

(1) CFG [21]. CFG models the latency distribution of each node and its children in the program control flow and uses it to determine anomalies in program execution.

(2) FSA [9]. Use FSA (finite-state automaton) to model for method invocation in the program execution flow. When the automaton cannot accept a program execution flow or its latency exceeds the threshold, this execution flow is considered to be anomalous.

(3) TraceAnomaly (TA) [19]. TA takes the preceding call path of each span as one dimension of its input service trace vector (stv) and encodes the span latency as its feature. TraceAnomaly models the service trace vector with VAE.

(4) LSTM [22]. The multimodal LSTM model encodes the traces into sequences of vectors according to the order of spans in the traces and reconstructs them with an LSTM model.

(5) CRISP [30]. CRISP computes the critical paths for traces and presents them in the form of critical calling context trees (CCCT). Anomalies are detected based on TraceAnomaly [19] with the service critical path vectors (SCPV), which are obtained with the CCCT.

(6) TraceCRL [29]. TraceCRL employs graph contrastive learning in modeling trace graphs and detects anomalies with classifiers (*e.g.* OC-SVM).

4.3 Experiment Settings

4.3.1 Evaluation Metrics. To evaluate the accuracy of anomaly detection, following the common practice in anomaly detection [11, 18], we choose the best F-Score and AUC as the accuracy evaluation metrics. These metrics are both widely used threshold-free metrics, which do not require threshold selection.

4.3.2 Experimental Details. We use PyTorch and the DGL to implement the GTrace model. The training batch size is set to 128, the learning rate is set to 0.001, the weight decay rate is set to 0.01, and the AdamW optimizer is used. The cache size is set to 2^{18} for both caches, to ensure that the cached content does not cause overflows of the GPU memory. During model training, we do not use inference acceleration techniques in order to ensure the number of training samples and the distribution of samples with different structures during training. In terms of baseline model implementation, for TraceAnomaly and CRISP, we use the open-source code repository[3][4] of the original papers and modify some parts of them for data loading and evaluation. For CFG, FSA, LSTM, and TraceCRL, we implement the algorithms based on the algorithm descriptions and model settings in the original paper with Python. Some of the baselines are modified to detect node-level latency anomalies, in which we use the latency reconstruction results of each node are used for the anomaly scores. We add classifiers after the node embeddings to detect node-level anomalies in TraceCRL. In order to perform detection for trace-level latency anomalies in GTrace, we take the average value of node latency anomaly scores. We use RTX 3060 GPU in model inference for all deep learning models, which are accelerated by CUDA.

4.4 RQ1: Baseline Comparison

The comparison of the model GTrace proposed in this paper with the baseline model is shown in Tab. 3. Our proposed GTrace model outperforms the other models in all evaluation indexes of the four datasets. Among all the evaluation metrics, GTrace leads the most in the structure-related metrics. Compared to the baseline models, GTrace models the overall structure of the traces by separating latency and structural features in a graph-based approach. This modeling approach helps fully learn the structural features of the traces, which results in more accurate structural anomaly detection.

GTrace also shows advantages in latency anomaly detection, especially in node-level metrics. Compared with the statistic-based methods like CFG and FSA, GTrace utilizes group-wise VAE to model node latency distribution, which enables more accurate latency anomaly modeling for different trace structures. Compared with model-based methods like TraceAnomaly, LSTM, CRISP, and TraceCRL, we innovatively propose the latency modeling method of "predicting latency with structure". This essentially avoids the dependence of the model on the input of latency features, which further helps mitigate the overfitting of latency features. It is worth noting that TraceCRL also uses a GNN-based model to obtain graph-level representations. However, TraceCRL can hardly be used for node-level anomaly detection due to its lack of targeted design for node-level representation learning.

4.5 RQ2: Effectiveness of Group-wise VAE

In this paper, we propose to use the "predicting latency with structure" approach with group-wise VAE to model the latency in traces. To investigate the effectiveness of the group-wise modeling approach proposed in this paper, we perform ablation studies on the VAE model in GTrace. The results are shown in Tab. 4. The first ablation study (+ input $\boldsymbol{T}$) add latency features $\boldsymbol{T}$ to the model input. It can be found that GTrace performs better in most of the evaluation metrics even without the latency input. This shows that the proposed group-wise VAE model can infer the latency information with the structural input and "predicting latency with structure" effectively mitigate the overfitting of latency features. The second one (- split $\boldsymbol{Z}$) studies the effectiveness of the "splitting node-wise and graph-wise model" design in the group-wise VAE. In this ablation study, we remove the node-wise model and only employ a graph-wise model with a single $\boldsymbol{Z}$ as the VAE structure. The results suggest that the splitting of node-wise and graph-wise models effectively enhances the learning ability of the VAE model for trace features.

[2] DeepTraLog [28] and TraceLingo [27] are not compared since they require extra data or labels. TraceVAE [25] is not compared since it does not support node-level latency detection.

[3] https://github.com/NetManAIOps/TraceAnomaly.git

[4] https://github.com/uber-research/CRISP

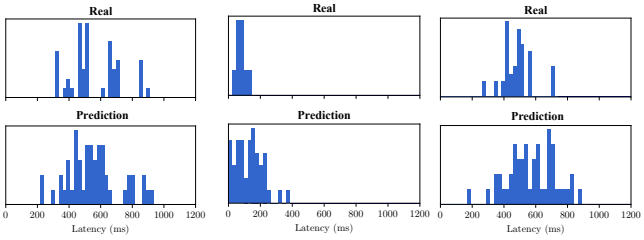
Table 3: Evaluation of accuracy, where "A" denotes AUC and "F" denotes F-Score. The best results are marked in bold and the second best results are underlined.

Dataset	Model	Node Latency				Trace Latency				Trace Structure			
		A	⇑A	F	⇑F	A	⇑A	F	⇑F	A	⇑A	F	⇑F
$\mathcal{A}$	CFG	0.795	9.1%	0.757	14.8%	0.880	6.3%	0.803	11.6%	0.192	340.1%	0.314	174.2%
	FSA	0.277	213.0%	0.446	94.8%	0.303	208.6%	0.472	89.8%	0.050	1590.0%	0.105	720.0%
	TA	0.250	246.8%	0.305	184.9%	0.337	177.5%	0.504	77.7%	0.286	195.5%	0.611	40.9%
	LSTM	0.052	1567.3%	0.121	618.2%	0.398	134.9%	0.394	127.4%	0.163	418.4%	0.254	239.0%
	CRISP	0.183	373.8%	0.278	212.6%	0.294	218.0%	0.318	181.8%	0.011	7581.8%	0.054	1494.4%
	TraceCRL	0.022	3840.9%	0.077	1028.6%	0.089	950.6%	0.277	237.6%	0.065	1200.0%	0.221	289.6%
	GTrace	**0.867**	-	**0.869**	-	**0.935**	-	**0.896**	-	**0.845**	-	**0.861**	-
$\mathcal{B}$	CFG	0.698	12.2%	0.663	10.4%	0.671	16.1%	0.717	5.0%	0.206	306.3%	0.501	60.7%
	FSA	0.392	99.7%	0.616	18.8%	0.384	102.9%	0.569	32.3%	0.124	575.0%	0.221	264.3%
	TA	0.275	184.7%	0.446	64.1%	0.337	131.2%	0.504	49.4%	0.286	192.7%	0.611	31.8%
	LSTM	0.147	432.7%	0.244	200.0%	0.759	2.6%	0.736	2.3%	0.123	580.5%	0.342	135.4%
	CRISP	0.143	447.6%	0.261	180.5%	0.336	131.9%	0.482	56.2%	0.295	183.7%	0.611	31.8%
	TraceCRL	0.023	3304.4%	0.072	916.7%	0.437	78.3%	0.552	36.4%	0.072	1062.5%	0.227	254.6%
	GTrace	**0.783**	-	**0.732**	-	**0.779**	-	**0.753**	-	**0.837**	-	**0.805**	-

Table 4: Ablation Study of GTrace

Data	Model	Node Lat.		Trace Lat.		Trace Struct.	
		A	F	A	F	A	F
$\mathcal{A}$	+ input T	0.439	0.407	0.498	0.453	0.795	0.850
	- split Z	0.208	0.290	0.297	0.359	0.277	0.424
	GTrace	**0.867**	**0.869**	**0.935**	**0.896**	**0.845**	**0.861**
$\mathcal{B}$	+ input T	0.668	0.659	0.746	0.750	0.559	**0.830**
	- split Z	0.551	0.601	0.644	0.684	0.219	0.385
	GTrace	**0.783**	**0.732**	**0.779**	**0.753**	**0.837**	0.805

4.6 RQ3: Effectiveness of Latency Modeling in GTrace



(a) Structure 1 **(b) Structure 2** **(c) Structure 3**

Figure 9: Real and reconstructed latency distributions for the same API in different sub-trees. "Real" shows the real latency distribution for the API in the current tree structure. "Prediction" shows the predicted latency distribution for the API in the current tree structure.

To further investigate the accuracy of the node-wise model in "prediction latency with structure", we visualize the prediction results of VAE in GTrace. Fig. 9 shows the real and predicted latency distributions for a certain API in different sub-trees. It can be found that the predicted latency of GTrace is different for different sub-trees, which also fits the real distribution well. This suggests that the group-wise VAE effectively models the latency propagation from the sub-tree structures.

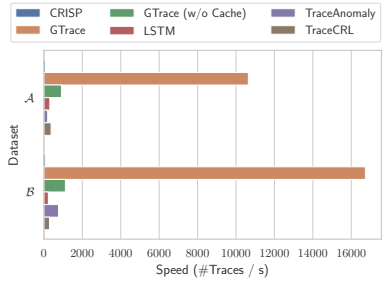
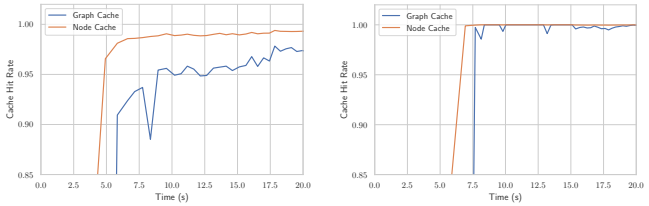


Figure 10: Speed Comparison on $\mathcal{A}$ and $\mathcal{B}$

4.7 RQ4: Evaluation of Time Efficiency



(a) Dataset $\mathcal{A}$ **(b) Dataset $\mathcal{B}$**

Figure 11: Cache Hit Rate on $\mathcal{A}$ and $\mathcal{B}$

To evaluate the time efficiency of GTrace, we implement and deploy the detection pipeline in Fig. 7. We also implement other

Table 5: Time consumption breakdown for 30,000 traces in $\mathcal{A}$. Note that graph building and anomaly detection are executed in parallel in different threads. "Other" period denotes other necessary codes in the program (e.g. logger, garbage collection. data format conversion, and some other functional logic codes).

Period	GTrace		w/o Cache	
	Time (s)	Percent	Time (s)	Percent
Graph Build (P.)*	0.144	4.7%	0.140	0.4%
Cache R & W	0.419	13.7%	0.000	0.0%
Model Inference	0.937	30.6%	20.110	56.2%
Other*	1.704	55.7%	15.713	43.8%
Total	3.060	100%	35.823	100%

DL-based algorithms in this pipeline for evaluation. To ensure fairness, the same graph building code is used for all methods. We set a cold start time of 10s for all methods and calculated the mean value of their detection speed, as shown in Fig. 10. It can be found that GTrace runs significantly faster than the other model-based baselines, achieving a 28.2x speedup in $\mathcal{A}$ and a 21.9x speedup in $\mathcal{B}$. This suggests that the group-wise model effectively improves time efficiency compared to other deep learning models.

To further investigate the effectiveness of cache and merged graph, we remove the cache and merged graph modules and compare its speed with the original GTrace model, shown as GTrace (w/o Cache) in Fig. 10 and Tab. 5. It can be found that the cache and merged graphs utilize very little time (Cache R & W) in exchange for a significant reduction in inference time. Fig. 11 also shows the cache hit rate in GTrace with the detected traces. It demonstrates that the hit rates of both the node cache and the graph cache get higher and eventually stabilize at over 95%, as the number of detected traces increases. This suggests that GTrace effectively takes advantage of the cache and merged graph and significantly reduces the running overhead in trace anomaly detection.

5 VISUALIZATION TOOL

5.1 Motivation and Design

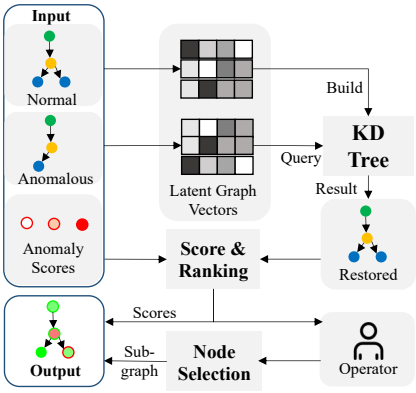


Figure 12: Overall Process of Visualization Tool

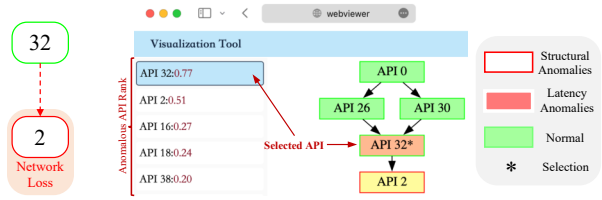
In large microservices systems, each failure can generate numerous anomalous traces with varying structures. Current anomaly detection algorithms typically present these traces individually, complicating operators' ability to quickly understand the anomalies. As a result, we believe it's crucial to provide operators with a **summary of detected anomalies**. To achieve this, we develop a visualization tool and deploy it on eBay's microservices cluster. The main features of the visualization tool are:

- Automatically reconstructs missing nodes and provides visualization for both latency and structural anomalies.
- Displays upstream and downstream APIs of the selected API, facilitating an easy understanding of fault propagation paths.

The overall process of the visualization tool is illustrated in Fig. 12. Displaying structural anomalies is challenging due to missing nodes in anomalous traces. We employ a KD-Tree constructed from structure Z of normal traces and queried by anomalous trace vectors to restore normal patterns. APIs from all restored anomalous traces within a specific time range are then ranked using the JI-score (following [17]) and displayed as a list for operators. Operators can select an API from the list, and a subgraph containing all APIs within a distance of k from the chosen API will be shown, presenting a failure summary instead of a single trace. More information about this visualization tool can be found in the source code. To illustrate the efficacy of the visualization tool on failure cases, we deploy GTrace on eBay's microservices cluster and present two case studies to demonstrate the visualization results.

5.2 Case 1: Structural Anomalies

(a) Example traces. Some nodes are lost in abnormal traces due to the failure.



(b) Failure **(c) Output of Visualization Tool**

Figure 13: Visualization on Case 1

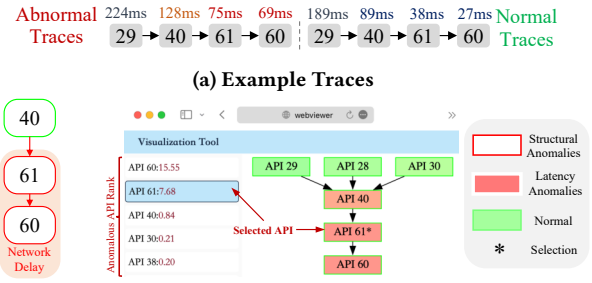
Fig. 13 shows the output of the visualization in a failure which is caused by network packet loss between "API 32" and "API 2" (Fig. 13c). In this case, "API 2" are lost in abnormal traces due to the failure[5]. Therefore, if the anomalous traces are output one by one, operators may *not* be able to notice the anomalies in API 2.

However, as depicted in Fig. 13c, the visualization tool successfully reconstructs API 2. The operators can obtain this as follows: First, select an API from the abnormal ranking in Fig. 13c. In this

[5]Some tracing tools are able to record the "caller" span, with which there will be an error span of API 2 from the server side. But in this paper, we consider all spans as the "callee" span. In this case, all of the spans in the sub-tree of API 2 are lost.

case, "API 32" is selected and the visualization tool shows Fig. 13c, where the red border of API 2 indicates the structural anomalies in this failure. Moreover, we can find that the loss of API 2 causes latency anomalies in API 32. Therefore, even if "API 2" is missing from the traces, operators can still find the anomaly in "API 2" and understand the anomaly propagation through the visualization tool.

5.3 Case 2: Latency Anomalies



(a) Example Traces

(b) Failure (c) Output of Visualization Tool

Figure 14: Visualization on Case 2

In Fig. 14, we demonstrate a failure case with latency anomalies, where the API 60 and API 61 are both experiencing network delay. Similarly, both API 60 and API 61 are filled with red in the visualization tool (Fig. 14c), which indicates that the proposed visualization tool successfully shows the anomalies in this case. API 40 is also filled with light red color, which indicates that its downstream anomalies cause its own latency anomalies with a weaker degree. In this case, the visualization tool avoids the need for operators to manually examine each trace and instead displays the anomalous API along with the fault propagation.

6 THREATS TO VALIDITY

The threat to the internal validity mainly lying in this paper is the implementation of the anomaly detection systems. We implemented LSTM, CFG, FSA, and TraceCRL by ourselves as there are no publicly available implementations. We implement these algorithms following their papers.

The threat to external validity mainly lies in this paper in the evaluation datasets. Our experimental study is based on two datasets, but the results might not represent the performance in other systems. To reduce this threat, we collect these datasets from several different microservices systems or different business components. In addition, the time efficiency of the algorithm is also highly dependent on the CPU model and architecture on which the detection pods run. In the future, we will evaluate the performance of GTrace on more microservices systems.

7 RELATED WORKS

In microservice systems, the objects of anomaly detection mainly include time series [24, 26], logs [8, 13] and traces [2, 19]. Compared with them, traces have flexible graph structures, thus bringing more challenges to anomaly detection. Numerous statistics-based approaches [2, 9, 10, 17, 21] derive associations between latencies and structures by employing grouping statistical methods. In recent years, many studies have leveraged model-based frameworks in conjunction with deep learning to achieve more accurate trace modeling. TraceAnomaly [19] encodes latency features into a dimension of the stv (Service Trace Vector) and employs a VAE model to reconstruct it. In CRISP [30], the critical path of a trace is computed and presented in the form of critical calling context trees (CCCT). CRISP detects the anomalies based on TraceAnomaly [19] with the stv of CCCT. The multimodal LSTM [22] and MSP [3] convert the trace to a sequence and use the LSTM and the attention model, respectively, to capture the sequential pattern. Recently, GNN-based methods have been used. TraceLingo [27] employs the Tree-LSTM model to capture the dependencies, but it is still a point-wise model, which is very different from the proposed GTrace. DeepTraLog [28] proposed to combine traces and logs for anomaly detection with GNN. TraceCRL [29] introduces graph contrastive learning to trace anomaly detection and sampling. TraceVAE [25] further introduces graph VAE in trace modeling. These model-based approaches have better learning ability for trace structures but tend to have a larger runtime overhead. Therefore, we present the first group-wise trace anomaly detection algorithm GTrace, which leverages group-wise modeling and inference acceleration to significantly improve time efficiency while attaining accurate detection through deep learning.

8 CONCLUSION

In failure diagnosis of large microservices systems, the anomaly detection algorithm needs not only accuracy but also time efficiency. This paper presents GTrace, the first group-wise trace anomaly detection method. We first conduct an empirical study to select the sub-tree as the grouping strategy. Based on this grouping strategy, we propose a novel group-wise VAE model, which takes group-wise structural features as input, and models latency in a novel "predicting latency with structure" way. We also propose a tree-structured node-level cache and a graph cache, combined with dynamic programming and merged graph strategy, to enable accurate and efficient anomaly detection with a reasonable resource. We evaluate GTrace on two datasets, including a real-world dataset, and the results show that the proposed group-wise anomaly detection approach is effective in both accuracy (2.64%-195.45% AUC improvement) and time efficiency (21.9x to 28.2x speedup). Besides, we present a visualization tool to show a summary of detected trace anomalies in the form of a graph. We deployed GTrace on eBay's microservices system and analyzed the effectiveness of the visualization tool.

ACKNOWLEDGMENTS

We thank Haowen Xu and Zeyan Li for their helpful suggestions. This work is supported by the National Key Research and Development Program of China (No.2019YFE0105500), the Research Council of Norway (No.309494), the State Key Program of National Natural Science of China (No.62072264), the National Natural Science Foundation of China (No.62202445), and the Beijing National Research Center for Information Science and Technology (BNRist) key projects.

REFERENCES

[1] Mahtab Ahmed, Muhammad Rifayat Samee, and Robert E. Mercer. 2019. Improving Tree-LSTM with Tree Attention. In *13th IEEE International Conference on Semantic Computing, ICSC 2019, Newport Beach, CA, USA, January 30 - February 1, 2019*. IEEE, 247–254.

[2] Liang Bao, Qian Li, Peiyao Lu, Jie Lu, Tongxiao Ruan, and Ke Zhang. 2018. Execution Anomaly Detection in Large-Scale Systems through Console Log Analysis. *Journal of Systems and Software* 143 (Sept. 2018), 172–186.

[3] Jasmin Bogatinovski, Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. 2020. Self-Supervised Anomaly Detection from Distributed Traces. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, Leicester, UK, 342–347.

[4] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *Comput. Surveys* 41, 3 (July 2009), 1–58.

[5] Jian Chen, Fagui Liu, Jun Jiang, Guoxiang Zhong, Dishi Xu, Zhuanglun Tan, and Shangsong Shi. 2023. TraceGra: A trace-based anomaly detection for microservice using graph deep learning. *Computer Communications* 204 (2023), 109–117.

[6] David M Ciemiewicz. 2001. What Do You mean?-Revisiting Statistics for Web Response Time Measurements. In *Int. CMG Conference*. 385–396.

[7] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*, Manuel Mazzara and Bertrand Meyer (Eds.). Springer International Publishing, Cham, 195–216.

[8] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 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*. ACM, Dallas Texas USA, 1285–1298.

[9] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *2009 Ninth IEEE International Conference on Data Mining*. IEEE, Miami Beach, FL, USA, 149–158.

[10] Debin Gao, Michael K. Reiter, and Dawn Song. 2004. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security - CCS '04*. ACM Press, Washington DC, USA, 318.

[11] Astha Garg, Wenyu Zhang, Jules Samaran, Ramasamy Savitha, and Chuan-Sheng Foo. 2021. An Evaluation of Anomaly Detection and Diagnosis in Multivariate Time Series. *IEEE Transactions on Neural Networks and Learning Systems* (2021), 1–10.

[12] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 1387–1397.

[13] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. 2016. Experience Report: System Log Analysis for Anomaly Detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Ottawa, ON, Canada, 207–218.

[14] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*. 76–91.

[15] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonca, James Lewis, and Stefan Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (May 2018), 24–35.

[16] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[17] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, Zhekang Chen, Wenchi Zhang, Xiaohui Nie, Kaixin Sui, and Dan Pei. 2021. Practical Root Cause Localization for Microservice Systems via Trace Analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, Tokyo, Japan, 1–10.

[18] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2012. Isolation-Based Anomaly Detection. *ACM Transactions on Knowledge Discovery from Data* 6, 1 (March 2012), 1–39.

[19] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Coimbra, Portugal, 48–58.

[20] Joshua Mitton, Hans M. Senn, Klaas Wynne, and Roderick Murray-Smith. 2021. A Graph VAE and Graph Transformer Approach to Generating Molecular Graphs. *arXiv:2104.04345 [cs]* (April 2021). arXiv:2104.04345 [cs]

[21] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B. Dasgupta, and Subhrajit Bhattacharya. 2016. Anomaly Detection Using Program Control Flow Graph Mining From Execution Logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, San Francisco California USA, 215–224.

[22] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. 2019. Anomaly Detection from System Tracing Data Using Multimodal Deep Learning. In *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama (Eds.). IEEE, 179–186.

[23] Christian P Robert, George Casella, and George Casella. 1999. *Monte Carlo statistical methods*. Vol. 2. Springer.

[24] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. 2019. Robust Anomaly Detection for Multivariate Time Series through Stochastic Recurrent Neural Network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, Anchorage AK USA, 2828–2837.

[25] Zhe Xie, Haowen Xu, Wenxiao Chen, Wanxue Li, Huai Jiang, Liangfei Su, Hanzhang Wang, and Dan Pei. 2023. Unsupervised Anomaly Detection on Microservice Traces through Graph VAE. In *Proceedings of the ACM Web Conference 2023*. 2874–2884.

[26] Haowen Xu, Yang Feng, Jie Chen, Zhaogang Wang, Honglin Qiao, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li, Ying Liu, Youjian Zhao, and Dan Pei. 2018. Unsupervised Anomaly Detection via Variational Auto-Encoder for Seasonal KPIs in Web Applications. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*. ACM Press, Lyon, France, 187–196.

[27] Yong Xu, Yaokang Zhu, Bo Qiao, Hongshu Che, Pu Zhao, Xu Zhang, Ze Li, Yingnong Dang, and Qingwei Lin. 2021. TraceLingo: Trace representation and learning for performance issue diagnosis in cloud services. In *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*. IEEE, 37–40.

[28] 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 *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. IEEE, 623–634.

[29] Chenxi Zhang, Xin Peng, Tong Zhou, Chaofeng Sha, Zhenghui Yan, Yiru Chen, and Hong Yang. 2022. TraceCRL: contrastive representation learning for microservice trace analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1221–1232.

[30] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 655–672.

Received 2023-05-18; accepted 2023-07-31