Actionable and Interpretable Fault Localization for Recurring Failures in Online Service Systems

Zeyan Li*
Nengwen Zhao
Mingjie Li
Xianglin Lu
Tsinghua University; BNRist
Beijing, China

Lixin Wang
Dongdong Chang
China Construction Bank
Beijing, China

Xiaohui Nie
Li Cao
Wenchi Zhang
Kaixin Sui
BizSeer
Beijing, China

Yanhua Wang
Xu Du
Guoqiang Duan
China Construction Bank
Beijing, China

Dan Pei†
Tsinghua University; BNRist
Beijing, China

ABSTRACT

Fault localization is challenging in an online service system due to its monitoring data's large volume and variety and complex dependencies across/within its components (e.g., services or databases). Furthermore, engineers require fault localization solutions to be actionable and interpretable, which existing research approaches cannot satisfy. Therefore, the common industry practice is that, for a specific online service system, its experienced engineers focus on localization for recurring failures based on the knowledge accumulated about the system and historical failures. More specifically, 1) they can identify the underlying root causes and take mitigation actions when pinpointing a group of indicative metrics on the faulty component; 2) their diagnosis knowledge is roughly based on how one failure might affect the components in the whole system.

Although the above common practice is actionable and interpretable, it is largely manual, thus slow and sometimes inaccurate. In this paper, we aim to automate this practice through machine learning. That is, we propose an actionable and interpretable fault localization approach, *DéjàVu*, for recurring failures in online service systems. For a specific online service system, *DéjàVu* takes historical failures and dependencies in the system as input and trains a localization model offline; for an incoming failure, the trained model online recommends where the failure occurs (i.e., the faulty components) and which kind of failure occurs (i.e., the indicative group of metrics) (thus actionable), which are further interpreted by both global and local interpretation methods (thus interpretable). Based on the evaluation on 601 failures from three production systems and one open-source benchmark, in less than one second, *DéjàVu* can on average rank the ground truths at 1.66~5.03-*th* among a long candidate list, outperforming baselines by at least 51.51%.

1 INTRODUCTION

Recently, online service systems (e.g., online shopping platform or E-bank) gradually replace traditional software systems and play an indispensable part in our daily life [11, 12, 14, 61]. Though tremendous effort have been devoted to software service maintenance (e.g., various metrics, such as average response time or memory usage, are closely monitored on a 24×7 basis [14]), failures are inevitable in online service systems due to the large scale and complexity, causing huge economic loss and user dissatisfaction [8, 39, 46, 50].

To enable engineers to resolve failures efficiently, fault localization is at the core of software maintenance for online service systems [13, 18, 43]. However, existing approaches mainly focus on unactionable fault levels, e.g., individual metrics [31, 41, 55] or components [22, 40, 50, 63], which, respectively, can be too fine-grained (e.g., it is hard to tell what exactly happened if only *memory usage* is localized) or coarse-grained (e.g., diagnosing faulty components is still challenging). *To be actionable, we aim to inform engineers where the failure occurs (i.e., the faulty component) and which kind of failure occurs (e.g., memory leak).* For convenience, we name the combination of a specific kind of failure and a location as a *failure unit*. More specifically, monitoring metrics serve as the most direct signals to the underlying failures [14], and different kinds of failures (thus the corresponding mitigation actions) can be indicated by different groups of metrics on the faulty components [14, 52]. For example, a combination of high memory usage and high #requests indicates insufficient memory caused by bursting requests rather than memory leak. In summary, we aim to recommend the faulty components and the corresponding indicative metric groups to engineers, i.e., the faulty failure units.

There are four challenges to achieve this goal. First, it is hard to represent failure units uniformly for further analysis because 1) failure units can contain different numbers of metrics and 2) it is hard to select appropriate time-series feature extractors for the various metrics manually. Second, due to the complex dependencies in an online service system, the faulty failure units can cause other metrics on the same or other components abnormal, and it is challenging to model the complex and various failure propagation. Third, since it is unlikely that all failure units have been faulty, it is essential but challenging to generalize to *previously unseen failures* (no failures of the same kinds have occurred at the same locations).

Fourth, lacking either local (i.e., interpreting individual cases [5]) or global (i.e., interpreting general model decisions [24]) interpretability makes it prohibitive for engineers to trust the localization results. Both are important because they interpret localization results from different perspectives.

This paper proposes an actionable and interpretable fault localization approach, *DéjàVu*[1], for recurring failures in online service systems. *Recurring failures* are repeated failures of the same kinds at different locations (e.g., high response time caused by different inefficient SQL queries). Failures may recur due to misunderstanding of root causes, delayed fix deployment, or emergent behaviors caused by high utilization [7]. Fault localization for recurring failures is important due to its large prevalence (e.g., 74.38% in *compA*) in practice (see § 2.1). For recurring failures in a specific online service system, its engineers can summarize the indicative metrics on each class of components to recognize the underlying failure types and direct their mitigation action, according to their domain knowledge and diagnosing experience. Based on these groups of indicative metrics, we define the candidate failure units for recurring failures in the system. Furthermore, to represent the complex dependencies in the system, we connect the failure units that have dependencies between each other into a *failure dependency graph (FDG)* (see § 2.2). Note that a system's FDG is evolving due to deployment and software changes, and engineers can also add new failure units.

When a failure occurs, the monitoring system raises alerts and triggers *DéjàVu*, which takes the latest FDG and the metric values as inputs and recommends suspicious failure units from the candidate failure units on the FDG to engineers (see later in Fig. 5). For challenge 1, *DéjàVu* employs gated recurrent unit-based [15] feature extractors to represent each failure unit as a fixed-width vector (unit-level feature) regardless of its metrics. For challenge 2, we apply graph attention networks [53] on the FDG to consider both the dependencies and the unit-level features. For challenge 3, our model learns from the historical failures of the same kind but at *any locations* to recommend a faulty failure unit by the metrics values and relative structure on the FDG only. For challenge 4, we provide local interpretation by finding the representative historical failures from which the trained model probably learns to make the recommendations for a failure. We also globally interpret the trained model as human-readable rules (e.g., see later in Fig. 15) by mimicking it with decision trees [64]. After engineers get the faulty failure units, they can timely recognize the underlying failure types and take mitigation action to ensure the quality of software services. Finally, the ground-truth failure units manually confirmed by engineers will be saved for future retraining. Unlike existing works [7, 8, 45], we do not aim to find similar historical failures and adopt their ground truths because such methods cannot localize *previously unseen* failures.

We extensively validated *DéjàVu* with four datasets, three of which are based on real-world systems, containing 502 injected failures of 18 types and 99 real-world failures[2]. The results show that *DéjàVu* is effective in localizing faulty failure units. Specifically speaking, the average rank of the ground truths achieves 1.66~5.03 and outperforms baselines by 51.51%~99.41%. The results

also show that the main modules, such as feature extractors and GAT-based aggregation, indeed contribute. Particularly, the results on production systems and real-world failures demonstrate the practical performance. *DéjàVu* is efficient as it costs tens of minutes to train the localization model and less than one second to localize for a failure. Moreover, *DéjàVu* can achieve similar performance on previously unseen failures compared with seen failures. Finally, we demonstrate the effectiveness of our interpretation techniques.

The contributions of this paper are summarized as follows:

- For the first time, we propose an actionable and interpretable fault localization approach, *DéjàVu*, for recurring failures in online service systems. *DéjàVu* offline trains a localization model using historical failures in a given online service system, and online recommends and interprets faulty failure units.
- We propose a novel localization model and two interpretation methods addressing all four challenges.
- We conduct extensive experiments on 601 failures from four systems, including 99 real-world failures and three real-world systems. The results show *DéjàVu*'s effectiveness, efficiency, generalizability, and interpretability. We also share lessons learned from our industrial experience.

2 BACKGROUND
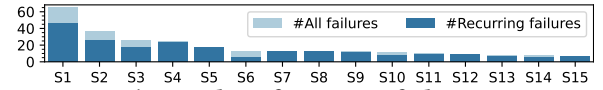
2.1 Recurring Failures

Figure 1: The number of recurring failures at *compA*.

We investigated the failures tickets at *compA*, a large commercial bank with over a hundred million users, to motivate our work on recurring failures. The banking information system at *compA* contains over 300 applications, each of which runs on dozens of servers and contains many components such as database, web server, and load balancer. We collected 576 failure tickets from this system, spanning 12 months, which are triggered by application SLO (service level objective) violation and contain a detailed diagnosis and mitigation process.

We categorize the root causes of these failures into recurring and non-recurring categories by whether they can be repetitive with historical failures (maybe at different locations). The recurring categories are mainly external, hardware, or middleware reasons, such as *bad requests*, *unavailable third-party services*, *failed disks*, *slow SQL queries*, and *missing database indices*. The non-recurring categories are mainly logical reasons, such as *code defects*, *design flaws*, and *data inconsistency* (e.g., incorrect modification on a manually maintained special account list).

In Fig. 1, we present the number of all/recurring failures in the applications with the most failure tickets. On all applications except S6, most failures (at least 2/3) are recurring. In total, there are 74.38% recurring failures. Lots of failures (39.28%) are recurring due to unavailable third-party services. There are also existing studies reporting the large prevalence of recurring failures. For example, Dogga et al. find that recurring categories of root causes cause 94% of the failures at a major SaaS company [20], and Lee and Iyer find that 70% of the failures they studied are recurring [35].

---

[1]A French phrase translating literally to "already seen"
[2]The datasets and implementation can be found in our replication package [1]

In summary, diagnosing recurring failures is important due to the large prevalence. Moreover, the repetitive nature motivates us to diagnose recurring failures by learning from historical failures.

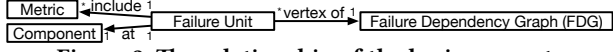## 2.2 Defining Failure Units and FDGs

Figure 2: The relationship of the basic concepts

Table 1: Component classes (CC) and metric groups in *sysA*

CC	Group	Metrics in the metric group
DB, OSB, Service	Requests	#requests, average response time, success rate, process time
DB	Load	ACS, AIOS, AWS, CPUUsedPct, CPUFreePct, CPUPused, {Call, DFParaWrite, Exec, LFParaWrite, LFSync, Login, TPS, SctRead}PerSec, Logic/Physical ReadPerSec, PGAUsedPct, SEQUsedPct, SessConnect, TbsUsedPct, UndoTbsPct, UserCommit
DB	Memory	MEMTotal, MEMUsed, MEMUsedPct, MEMRealUtil
DB	Session	ProcUsedPct, ProcUserUsedPct, Sess{Active, Connect, UsedTemp, UsedUndo, Pct}
DB	State	DbTime, Hang, OnOffState, RowLock, tnspingResultTime
DB	Space	AsmFreeTb, DbFileUsedPct, NewTbs FreeGb/UsedPct, PGAUsedTotal, RedoPerSec, TbsFreeGb, TbsUsedPct, TempTbsPct, Total/Used TbsSize
Docker	CPU	containerCpuUsed
Docker	Memory	containerMemUsed
Docker	Session	containerSessionUsed
Docker	Thread	Thread{Idle, Running, Total, UsedPct}
Docker	Full GC	containerFgc, containerFgct
OS	State	AgentPing, ICMPPing
OS	Disk	Disk{AvgquSz, Await, IoUtil, RdIos, RdKbs, Svctm, WrIos, WrKbs}, FS{MaxAvail, MaxUtil, TotalSpace, UsedPct, UsedSpace}, Free/Total DiskSpace, Free/Total Inodes, Used DiskSpace/Inodes, Used DiskSpace/Inodes Pct
OS	Load	BuffersUsed, CPU{Frequency, IdlePct, IowaitTime, KernelNumber, Number, Pused, SystemTime, UserTime, UtilPc, UtilPct}, NumOfProcesses, NumOfRunningProcesses, ProcessorLoad 1/5/15 Min, System Block/Wait Queue Length, ZombieProcess
OS	Memory	BuffersUsed, CacheUsed, MemoryAvailable, MemoryAvailablePct, MemoryFree, MemoryTotal, MemoryUsed, MemoryUsedPct, PagePi, PagePo, SharedMemory, SwapUsedPct
OS	Network	Incoming/Outgoing NetworkTraffic, Received/Sent ErrorsPackets/Packets/Queue/Total, ssTotal

* #components in OSB (Oracle Service Bus)/Service/Docker/DB/OS: 2/8/8/3/6.

This section describes how *DéjàVu* defines failure units and FDGs to enable actionable fault localization. As introduced in § 1, the experienced engineers of a specific online service system can define the candidate failure units by summarizing the indicative metric groups on different component classes (see Fig. 2). For example, in Table 1, we present the component classes and metric groups for $\mathcal{A}$ (see § 5.1). *Each group of such indicative metrics at a corresponding component is a candidate **failure unit** for recurring failures in the system*. For example, the metrics in *Requests* (i.e., *#requests*/*average response time*/...) at DB1 form a failure unit, *DB1 Requests*. In this paper, we focus on metrics, as logs are of huge volume and various types and traces contain little information on low-level performance issues. For convenience, we name the failure units defined by the same group of metrics on the same component class, which contain the same metrics at different locations, as a *failure class*.

To model the complex and various failure propagation, we represent the dependencies in an online service system with an *FDG*.

Definition 1. An **FDG** (failure dependency graph) is an undirected graph, $G = (V, E)$. V is the set of all the defined candidate failure units of an online service system. An edge (v_i, v_j) exists in the edge set, E, *iff*. the failure unit v_i depends on v_j or vice versa.

For example, the FDG of *sysA* (§ 5.1) is shown in Fig. 3, which is complex considering that *sysA* comprises only 27 components (see Table 1). In Fig. 3, a failure caused by CPU exhaustion on *Docker6* propagates to *Service6 Requests*, since *Service6* is deployed on *Docker6*. It further propagates to *Service1 Requests*, *Service2 Requests*, and *OSB1 Requests*, since *Service1* and *Service2* rely on *Service6* and *OSB1* relies on these two services.

We construct FDGs automatically, which is necessary because FDGs are complex and dynamic (e.g., in microservice systems, pods are dynamically created and deleted), based on the call and deployment component relationships. For example, in *sysA*, we collect the call relationships among OSBs, services, and databases through tracing tools (the leftmost graph in Fig. 4) and collect deployment relationships among services, containers, and servers (the others in Fig. 4) through configuration management database. By combining the component relationships and failure units, we automatically construct an FDG. For example, as *Service1* is deployed on *Docker1* (see Fig. 4), *Service1 Requests* is connected to *Docker1 CPU/Memory/...* (see Fig. 3). Besides, engineers can manually add or remove edges on the FDG according to their domain knowledge and existing diagnosis knowledge. For example, when two services depend on a stateful third-party service, which cannot be captured by tracing, engineers can manually connect them on the FDGs.

In this paper, we group metrics roughly by their general categories (e.g., CPU-related or network-related) for most datasets and include no prior knowledge in the construction of FDGs, since it is already enough for the failures we collected (see § 5). In practice, engineers can carefully tune the specifications of failure units and FDGs to make the localization results more helpful, which is out of the topic of this work.

2.3 Problem Statement

In this work, we target recommending the faulty failure unit given the latest FDG and the corresponding metric values when a failure occurs. The faulty failure unit pinpoints where the failure occurs (the component) and which kind of failure occurs (indicated by the metrics), and thus, helps engineers take the right mitigation actions rapidly. Finding the most helpful specifications of failure units or FDGs and failure discovery are out of the scope of this paper.

3 DESIGN OF *DÉJÀVU* MODEL

In this section, we introduce the design of *DéjàVu* model. As shown in Fig. 5, we train a model offline with historical failures and FDGs of a given online service system. The trained model recommends faulty failure units online when a failure occurs, given the metric values and the latest FDG.

3.1 Overview

As shown in Fig. 6, the *DéjàVu* model takes the metric values and the FDG around the failure time as inputs and outputs a suspicious score for each failure unit with a binary classifier. Note that different failures could have different corresponding FDGs.

More specifically, first, to solve the challenge of representing failure units uniformly, we introduce a feature extractor module that can grasp temporal information of metrics and correlations among metrics (§ 3.2). A feature extractor is trained to map the metrics of
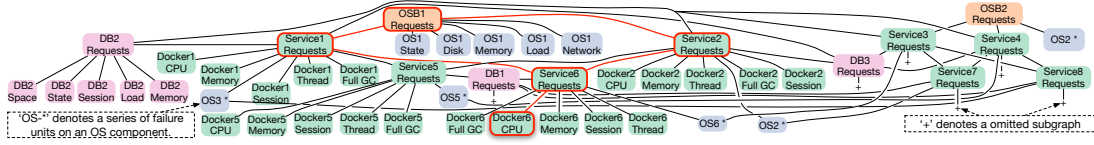
Figure 3: The FDG of _sysA_. There are 102 vertices and 109 edges in total, some of which are omitted due to the space limit.
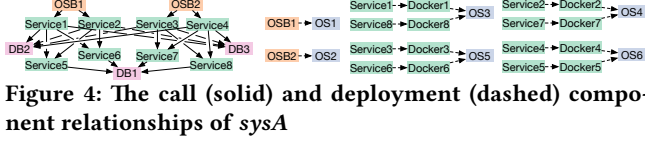

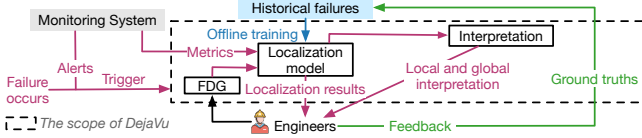**Figure 4: The call (solid) and deployment (dashed) component relationships of _sysA_**

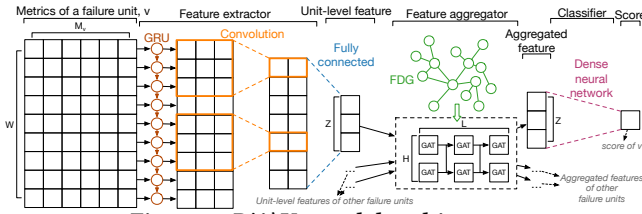**Figure 5: Workflow of _DéjàVu_**

Figure 6: _DéjàVu_ model architecture

any failure unit of the same failure class into a fixed-width vector (unit-level feature). We train a failure extractor for each failure class, since failure units of different failure classes contain different metrics. Second, to enable modeling failure propagation, we employ a feature aggregator to encode the structural information on the FDG into aggregated features (§ 3.3). The feature aggregator utilizes attention mechanism to pay more attention on related failure units. Finally, we introduce a classifier to score each failure unit based on its aggregated feature (§ 3.4). For generalizability, the failure units in the same failure class share a feature extractor, and all failure units share a feature aggregator and classifier. For different failure classes, their feature extractors are all the same except the input size. In this way, the score of a failure unit (e.g., _Docker6 CPU_ in Fig. 3) is majorly determined by its metric values and the metric values of its related failure units (e.g., _Service6 Requests_, _Service2 Requests_, _OSB1 Requests_), even if no historical failure has the same faulty failure unit.

Our model is trained by minimizing a revised loss function (§ 3.5). Furthermore, we employ class balancing for training, since the frequency of each failure class in training data varies (§ 3.6).

3.2 Feature Extractor

Each metric of a failure unit has temporal information, and there are correlations among the metrics. To better extract meaningful features, we use a three-stage feature extraction method. Specifically speaking, the first stage learns temporal information, while the latter two learn higher-level features.

In the first stage, the temporal feature of a failure unit is extracted using gated recurrent unit (GRU)[15] recurrent neural networks. GRU is a recurrent neural network with a gating mechanism but

has simpler architecture and fewer parameters than LSTM. The input metrics of a failure unit are encoded as a numerical matrix of dimension $W \times M_v$, where W is the length of time window we slice at the failure time and M_v is the number of metrics of the failure unit v. In this paper, we empirically set W as 20 minutes to capture the near history of the metrics.

In the second stage, we apply 1-D convolution neural network (1-D CNN) [23] and GELU (Gaussian error linear unit) activation function [28] on the temporal feature matrix obtained at the first stage. With the application of 1-D CNN, we get multiple feature maps extracting the correlations among different time points and metrics. GELU provides nonlinearity by preserving linearity in the positive activations and suppressing the negative activations while relieving vanishing gradients problem [28].

In the third stage, a fully connected layer is applied to learn the relations among different feature maps obtained in the second stage and output a numeric _unit-level feature_ vector of dimension Z.

3.3 Feature Aggregator

The second module, feature aggregator, is responsible for aggregating the unit-level features of related failure units of a failure unit v and the structure information into one _aggregated feature_, $\hat{f}^{(v)}$, based on the FDG, G. For generalizability, the feature aggregator should be structure-independent. Thus, graph convolutional network (GCN) [33] is unsuitable. Since the relationship between any pair of failure units varies, it is also inappropriate to simply average features of connected failure units together as GraphSAGE [26] does. Therefore, for each failure unit, we dynamically calculate the aggregation weights of its related failure units based on their unit-level features through graph attention networks (GAT)[53].

However, a single GAT aggregates features of a failure unit's neighbors only, while failures propagate along the FDG for more than one hop. Therefore, we stack multiple GAT together sequentially, and the outputs of the last GAT are taken as the input of the next GAT. In this way, we aggregate features of farther related failure units and thus, model multi-hop failure propagation. Though multiple-layer graph neural network could suffer from over-smoothing, we apply residual connections to relieve the problem [36]. On the other hand, we also introduce multi-head attention to improve the capacity and stabilize the training process [53]. Specifically speaking, we apply multiple GATs in parallel and concatenate their outputs together. We denote the number of sequentially stacked GATs as L and the number of heads as H. We set $H = 4$ and $L = 8$ by default and discuss their impact in § 5.3. For generalizability, the feature aggregator is shared by all failure units regardless of their classes. In summary, the feature aggregator can be formalized as follows.

$$\{\hat{f}^{(l+1,v)}|v \in V\} = \underbrace{[\text{GAT}(\{\hat{f}^{(l,v)}|v \in V\}); ...; \text{GAT}(\{\hat{f}^{(l,v)}|v \in V\})]}_{H \text{ GATs in total}}$$

where $l \in \{0, 1, ..., L - 1\}$, $\hat{f}^{(0,v)} := f^{(v)}$, $\hat{f}^{(v)} := \hat{f}^{(L,v)}$, and $GAT(\cdot)$ calculates aggregated features for all failure units in the FDG.

3.4 Classifier

The final module, classifier, assigns a suspicious score, $s(v)$, for each failure unit given its aggregated feature, $\hat{f}^{(v)}$. Specifically speaking, we simply employ a two-layer dense neural network since it is already enough to achieve satisfactory performance. To restrict the output value in $[0, 1]$, we use sigmoid function σ as the output activation. In summary, the output suspicious score for a failure unit v (denoted as $s(v)$) can be formulated as follows,

$$s(v) = (\sigma \circ \text{Dense} \circ \text{GELU} \circ \text{Dense})(\hat{f}^{(v)}) \qquad (1)$$

We do not need thresholds on the suspicious scores to obtain binary classification for failure units. Instead, we only rank all failure units by their suspicious scores in descending order, based on which engineers can check the suspicious failure units one by one.

3.5 Loss Function

For each failure unit v of a failure T, we expect the suspicious score $s_T(v) \in [0, 1]$ to be as close as possible to its ground truth label $r_T(v) \in \{0, 1\}$. To measure the difference between $s_T(v)$ and $r_T(v)$, we use the widely-used binary cross-entropy [23]:

$$BCE(r_T(v), s_T(v)) = r_T(v) \cdot \log(s_T(v)) + (1 - r_T(v)) \cdot \log(1 - s_T(v))$$

Since most failure units are not faulty, their losses are dominant compared with the faulty ones. As a result, the model could fall back to score every failure unit equally. To solve this problem, we assign different weights to faulty and normal failure units. Specifically speaking, the weights of normal and faulty failure units are 1 and N (the number of failure units) respectively. In summary,

$$\mathcal{L}_s = \frac{1}{N_H} \sum_{T \in \{T_1, \cdots, T_{N_H}\}} \left[\frac{\sum_{v \in V} w_v^{(T)} BCE(r_T(v), s_T(v))}{\sum_{v \in V} w_v^{(T)}} \right] \qquad (2)$$

where V is the set of failure units, $w_v^{(T)} = r_T(v) \cdot |V| + (1 - r_T(v)) \cdot 1$, and $\{T_1, \cdots, T_{N_H}\}$ is the set of training failures. By minimizing $\mathcal{L}_s$ over all historical failures, we train the *DéjàVu* model.

3.6 Class Balancing

In practice, the number of failures in different classes varies. Thus, to prevent the *DéjàVu* model from neglecting the minority classes in the training process, we upsample historical failures of such classes. Suppose there are C failure classes and the number of failures of the i-th class is $N_H^{(i)}$, we sample a failure with probability $1/(C \cdot N_H^{(i)})$ for each training step. In this way, the failures of each class are sampled with the same probability (i.e., $1/C$).

4 INTERPRETATION METHODS

In this section, we propose two interpretation methods. As shown in Fig. 5, the interpretation is fed to engineers together with the localization results to help them understand and trust the results.

4.1 Global Interpretation

As shown in Fig. 7, we interpret the trained model globally by converting it into decision trees [64], which are rule-based models. More specifically, the training inputs for decision trees are
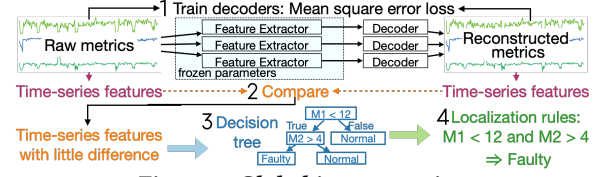

Figure 7: Global interpretation

time-series features of failure units' metrics and targets are the suspicious scores by *DéjàVu*. We do not use existing interpretation methods for two reasons. On the one hand, though many interpretation methods [6, 19, 60] are targeted at interpreting deep-learning models by understanding their inner mechanism, engineers are mainly concerned about the relationship between raw data and root causes [47]. On the other hand, the inputs and targets of our problem is non-standard and requires extra efforts.

First, we select 57 easy-to-understand time-series feature extractors (e.g., `range_count`, `variance`, `peak`) from `tsfresh`[3]. In total, these feature extractors extract 191 time-series features for each metric (see our replication package [1]), each of which represents an understandable property of a metric, such as level, variance, tailedness or complexity.

However, the number of time-series features for a failure unit (i.e., #metrics × 191) can be extremely large (e.g., thousands), while the feature dimension of *DéjàVu* (i.e., Z) is much lower (e.g., three). It indicates that *DéjàVu* would not consider most of these time-series features. To remove the useless and noisy time-series features, we first reconstruct metrics from the unit-level features given by *DéjàVu* and then compare the time-series features extracted from the original and reconstructed metrics. If a time-series feature differs a lot in the original and reconstructed metrics, it is not reserved by the *DéjàVu* model and should be removed.

We train a decoder for each failure class for metric reconstruction, which comprises a one-dimensional transpose convolution (1-D DeConv) layer, which can be treated as a transpose operation of Conv1D [21], and a fully-connected layer. To train the decoders, we minimize the loss function $\mathcal{L}_{AE}$, which measures the mean square error between each pair of original and reconstructed metrics:

$$\mathcal{L}_{AE} = \frac{1}{N_H} \sum_{T \in \{T_1, \cdots, T_{N_H}\}} \left[\frac{1}{|V|} \sum_{v \in V} \frac{\sum_{i=1}^{M_v} ||A_{T,i}^{(v)} - \tilde{A}_{T,i}^{(v)}||_2^2}{M_v} \right] \qquad (3)$$

where M_v is the number of metrics of a failure unit v, and $A_{T,i}^{(v)}$ is the i-th metric's values of v of the training failure T.

The training sample for decision trees of a failure unit comprises the selected time-series features of its metrics (as input) and the categorized suspicious score given by the *DéjàVu* model (as target). When interpreting a *DéjàVu* model, we are not concerned about the knowledge on which it is unconfident. Thus, we categorize the suspicious scores (i.e., $s(v)$ in (1)) into three different categories. Specially speaking, if $s(v) > 0.9$, then failure unit v is classified as a *faulty* failure unit; if $s(v) < 0.1$, then failure unit v is a *normal* failure unit; otherwise, the *DéjàVu* model is *uncertain*. With the training samples, we train decision trees which mimic the *DéjàVu* model (part 3 in Fig. 7) and classify failure units as "faulty" or "normal." Since failure units in different failure classes contain different metrics, we train a decision tree for each failure class. On the trained decision trees, we focus on the decision paths leading

to only faulty or normal failure units (part 4 in Fig. 7). These paths are the diagnosis rules that the *DéjàVu* model learns from historical failures. The rules are used to help engineers understand and trust the *DéjàVu* model rather than replace it. One the one hand, the performance of the decision trees are still worse than *DéjàVu*. One the other hand, the interpretations provided by the decision trees can also make mistakes [47].

4.2 Local Interpretation

Since the *DéjàVu* model is trained with historical failures, it is straightforward to interpret how it diagnoses a given failure by figuring out from which historical failures it learns to recommend the faulty failure units. To achieve this goal, we compare the incoming failure with each historical failure based on the aggregated features extracted by the trained *DéjàVu* model. The comparison is conducted on each pair of failure classes instead of failure units, since two similar failures can occur at different localization (thus have different faulty failure units). Due to the space limit, we omit the details, which can be found in our code [1].

5 EXPERIMENTS

In this section, we aim to address the following research questions.
- **RQ1**: How does *DéjàVu* perform in fault localization?
- **RQ2**: How does *DéjàVu* perform in various situations?
- **RQ3**: How does *DéjàVu* perform in terms of *efficiency*?
- **RQ4**: How does *DéjàVu* perform on previously unseen failures?
- **RQ5**: How do our interpretation methods perform?

5.1 Experiment Setup

5.1.1 Study Data. In the study, we use four datasets ($\mathcal{A}$, $\mathcal{B}$, C, and $\mathcal{D}$), containing 601 failures in total. In particular, 16 failures have multiple root causes and thus multiple faulty failure units. $\mathcal{A}$, $\mathcal{B}$, and C are from production systems, and $\mathcal{D}$ is based on an open-source benchmark system. The statistics of them are shown in Table 2. In all datasets, the FDG definition follows the process in § 2. To obtain failures for training, we split each dataset into a training set (40%), a validation set (20%), and a testing set (40%). All experiment results are calculated on the testing sets only.

The ground truths of $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{D}$ are determined by the locations and types of failure injection. In $\mathcal{A}$ and $\mathcal{B}$, their engineers injected ten types of failures: 1) CPU exhaustion on containers, physical servers (only $\mathcal{B}$), or middlewares (only $\mathcal{B}$); 2) packet loss or delay on physical servers; 3) database session limit (only $\mathcal{A}$) or halt (only $\mathcal{A}$); 4) low free memory for JVM/Tomcat (only $\mathcal{B}$); 5) disk I/O exhaustion (only $\mathcal{B}$). For $\mathcal{D}$, we deploy Train-Ticket [62] on a 4-node Kubernetes cluster. Train-Ticket is one of the largest open-source microservice benchmarks, containing 64 services. Following existing works [42, 59], we performed eight types of failure injection at random locations with ChaosMesh [4]: CPU/memory stress on pods/nodes, pod failure, and packet corrupt/loss/delay on pods. Details of the deployment and failure injection for $\mathcal{D}$ can be found in our replication package [1].

For C, we collected 99 real-world failures on *sysC*, spanning several months, and we labeled their ground-truth failure units with the engineers. Different from the other three, *sysC* is not a typical online service system and has only one component. Thus,

each failure class in C contains only one failure unit. The definition of the failure units in C are confirmed by the experienced engineers. In the FDG of *sysC*, all failure units are only connected to a virtual vertex (there is no metric in it), as we do not assume any prior diagnosis knowledge on the causal dependency among the metrics.

5.1.2 Baselines. We compare *DéjàVu* with the following state-of-the-art baseline methods:
- JSS'20 [8] represents failures as graphs and finds similar diagnosed failures by graph similarity for each incoming failure.
- iSQUAD [45] clusters historical failures, labels the root causes of each cluster, and assigns each incoming failure to a cluster.
- Decision Tree [64] takes the time-series features (see § 4.1) of a failure unit and classifies whether it is faulty.
- Random walk. As existing random walk-based works [31, 56] cannot take various numbers of metrics as input, we compare with two variants:
  - RandomWalk@Metric. Following existing works, we first construct a causal graph, calculate transition probabilities based on metric correlations, and obtain the score of each metric with personalized PageRank. Then, we obtain the score of each failure unit by summing up the scores of its metrics.
  - RandomWalk@FI. We score failure units (FI) directly by applying random walk on the FDGs and using the average correlation of all pairs of metrics of two failure units as transition probabilities.

The aggregation methods for RandomWalk@Metric (sum) and RandomWalk@FI (average) are selected so as to maximize the overall performance. Some methods [22, 38, 40, 63] are not applicable because they localize faulty services only.

5.1.3 Implementation. We implement *DéjàVu* and all baseline methods based on PyTorch[2] and DGL[54]. We extract the interpretable time-series features by tsfresh[16] and train decision trees by Scikit-Learn. We use Adam [32] to train the *DéjàVu* models and set the initial learning rate to 0.01 and weight decay to 0.01. To avoid gradient exploding, we clip the gradients of all parameters at 1. We train each model for 3000 epochs and set the batch size to 16. After publication, our implementation, including the baselines, will be public in our replication package [1].

5.1.4 Evaluation Metrics. Following related works [40, 63], we validate the effectiveness with top-k accuracy ($A@k$) and mean average rank (MAR). $A@k$ is the fraction of failures whose ground truths are included in the top-k recommendations. It represents how many failures can be diagnosed by checking only the first top-k recommendations, and larger is better. MAR is the mean of the average suggested rank of all ground truths of each failure. It represents how many recommendations should be checked to diagnose a failure on average, and smaller is better. Compared with $A@k$, MAR considers all recommended failure units but could be affected by extreme bad cases. We also measure the efficiency with respect to localization and training time.

5.2 RQ1: Effectiveness

5.2.1 Overall Performance. In Table 3, we present the fault localization results. We repeat each experiment ten times due to the stochastic training and present the average results. On every dataset,

Table 2: Dataset summary

Dataset	#Failures	#Metrics	#Failure units	#Failure Classes	System	Failure Source
$\mathcal{A}$	188	710	102	18	A production microservice system of a major ISP (*sysA*)	Injected by the engineers
$\mathcal{B}$	158	2419	189	20	A production serivce-oriented system of a commercial bank (*sysB*)	Injected by the engineers
C	99	2594	41	41	A production Oracle database system of a commercial bank (*sysC*)	Real-world failures
$\mathcal{D}$	156	5724	1044	23	A open-source microservice benchmark system, Train-Ticket [62] (*sysD*)	Injected by us

Table 3: Fault localization results

Dataset	Method	Effect Size	p-value	MAR	⇑MAR	A@1	⇑A@1	A@2	⇑A@2	A@3	⇑A@3	A@5	⇑A@5
	DéjàVu	-	-	1.66	-	77.18%	-	90.38%	-	93.40%	-	96.28%	-
	JSS'20	Huge (40.75)	5.4e-40	24.31	93.18%	16.66%	363.15%	25.64%	252.48%	35.89%	160.21%	47.44%	102.97%
	iSQUAD	Huge (10.90)	2.3e-18	12.91	87.17%	23.07%	234.48%	30.77%	193.71%	39.74%	135.00%	46.16%	108.60%
	Decision Tree	Huge (9.45)	1.2e-40	18.05	90.82%	53.20%	45.06%	63.46%	42.42%	65.77%	42.01%	66.03%	45.82%
$\mathcal{A}$	RandomWalk@Metric	Huge (41.36)	3.0e-40	25.68	93.55%	3.85%	1905.58%	6.38%	1315.70%	6.38%	1362.91%	8.92%	979.34%
	RandomWalk@FI	Huge (28.70)	5.2e-34	33.64	95.07%	16.51%	367.54%	19.04%	374.76%	20.32%	359.68%	20.32%	373.88%
	DéjàVu w/o GRU	Large (1.01)	2.5e-04	3.25	49.02%	74.36%	3.79%	87.56%	3.22%	91.67%	1.89%	93.46%	3.02%
	DéjàVu w/o AGG	Medium (0.70)	6.8e-03	2.32	28.60%	76.80%	0.50%	92.44%	-2.22%	94.49%	-1.16%	97.05%	-0.79%
	DéjàVu w/o BAL	Large (1.04)	1.8e-04	2.39	30.70%	71.28%	8.27%	85.00%	6.33%	88.97%	4.97%	92.44%	4.16%
	DéjàVu	-	-	5.03	-	66.21%	-	71.21%	-	75.61%	-	79.24%	-
	JSS'20	Huge (55.03)	1.7e-17	47.92	89.51%	15.15%	337.04%	19.70%	261.49%	24.24%	211.91%	30.30%	161.52%
	iSQUAD	Huge (33.00)	1.7e-15	30.75	83.65%	7.58%	773.51%	10.61%	571.19%	15.15%	399.05%	24.24%	226.91%
	Decision Tree	Huge (16.29)	1.3e-18	30.70	83.62%	55.91%	18.42%	60.30%	18.09%	63.49%	19.09%	65.00%	21.91%
$\mathcal{B}$	RandomWalk@Metric	Huge (21.25)	9.0e-14	21.59	76.72%	9.38%	605.88%	14.06%	406.49%	20.31%	272.26%	29.69%	166.90%
	RandomWalk@FI	Huge (32.88)	1.8e-15	30.66	83.60%	21.88%	202.61%	21.88%	225.47%	28.12%	168.87%	40.62%	95.08%
	DéjàVu w/o GRU	Huge (3.80)	5.2e-08	15.97	68.53%	65.91%	0.46%	70.76%	0.64%	73.18%	3.31%	77.42%	2.35%
	DéjàVu w/o AGG	Large (0.95)	2.4e-02	5.77	12.89%	69.70%	-5.00%	75.15%	-5.24%	78.03%	-3.11%	81.06%	-2.24%
	DéjàVu w/o BAL	Small (0.44)	1.7e-01	5.75	12.51%	62.43%	6.06%	69.39%	2.62%	74.85%	1.01%	78.18%	1.36%
	DéjàVu	-	-	1.70	-	61.84%	-	82.63%	-	90.79%	-	96.32%	-
	JSS'20	Huge (335.31)	1.5e-24	26.34	93.56%	34.21%	80.77%	52.63%	57.01%	60.53%	49.99%	68.42%	40.77%
	iSQUAD	Huge (24.53)	2.5e-14	3.50	51.51%	31.58%	95.83%	57.89%	42.74%	63.16%	43.75%	76.32%	26.20%
	Decision Tree	Huge (13.20)	5.4e-17	11.21	84.87%	38.95%	58.79%	44.74%	84.70%	45.27%	100.58%	58.42%	64.87%
C	RandomWalk@Metric	Huge (58.55)	9.9e-18	6.00	71.72%	5.26%	1075.70%	15.79%	423.32%	28.95%	213.61%	65.79%	46.40%
	RandomWalk@FI	Huge (73.25)	1.3e-18	7.08	76.03%	23.68%	161.16%	36.84%	124.30%	42.11%	115.60%	52.63%	83.01%
	DéjàVu w/o GRU	Large (0.90)	3.0e-02	2.10	19.07%	63.95%	-3.29%	90.00%	-8.19%	92.63%	-1.99%	96.05%	0.27%
	DéjàVu w/o AGG	Small (0.32)	2.4e-01	1.75	3.19%	58.95%	4.91%	88.68%	-6.82%	94.48%	-3.90%	97.37%	-1.08%
	DéjàVu w/o BAL	Very Large (1.24)	6.4e-03	2.16	21.25%	50.00%	23.68%	74.47%	10.95%	86.84%	4.54%	94.21%	2.24%
	DéjàVu	-	-	2.63	-	75.62%	-	84.69%	-	90.42%	-	94.27%	-
	JSS'20	Huge (529.06)	4.8e-40	303.39	99.13%	8.70%	769.24%	8.70%	873.44%	13.04%	593.37%	18.84%	400.38%
	iSQUAD	Huge (451.10)	4.5e-39	259.07	98.98%	21.74%	247.86%	30.43%	178.31%	30.43%	197.13%	31.88%	195.70%
	Decision Tree	Huge (24.37)	5.0e-27	128.98	97.96%	64.37%	17.48%	72.19%	17.31%	73.13%	23.64%	73.60%	28.09%
$\mathcal{D}$	RandomWalk@Metric	Huge (179.88)	1.7e-33	104.89	97.49%	2.90%	2507.72%	5.80%	1360.16%	8.70%	939.26%	13.04%	622.93%
	RandomWalk@FI	Huge (776.04)	2.2e-42	443.79	99.41%	0.00%	inf%	0.00%	inf%	0.00%	inf%	1.45%	6401.43%
	DéjàVu w/o GRU	Huge (2.84)	2.2e-07	13.32	80.25%	68.12%	11.01%	80.94%	4.63%	89.06%	1.52%	91.41%	3.13%
	DéjàVu w/o AGG	Huge (2.58)	9.5e-07	3.81	30.92%	83.60%	-9.54%	91.40%	-7.35%	92.19%	-1.92%	93.75%	0.56%
	DéjàVu w/o BAL	Very Small (0.17)	3.4e-01	2.74	3.85%	72.97%	3.64%	83.13%	1.88%	90.47%	-0.06%	94.69%	-0.44%

* ⇑ denotes the improvement rate (%) of *DéjàVu* over the compared method. Effect sizes (Cohen's d [17]) and p-values (*t*-test) are calculated based on MAR.

DéjàVu outperforms the five baselines with respect to all the metrics. Specifically speaking, the MAR of *DéjàVu* achieves 1.66~5.03 and outperforms the baselines by 51.51%~99.41% on all datasets; and the *A*@5 of *DéjàVu* achieves 79.41%~96.32% and outperforms others by 21.61%~6401.43%. On $\mathcal{B}$ and $\mathcal{D}$, all methods generally perform worse because there are many more candidate failure units (see Table 2). For statistical analysis, we calculate the effect size (Cohen's d [17]) and conduct *t*-test between the MARs of *DéjàVu* and the baselines. As shown in Table 3, the improvement over the baselines is huge and significant ($p < 0.05$).

JSS'20 and iSQUAD perform poorly for two reasons. First, they do not utilize historical failures until taking the ground truths of the found similar historical failures. Although carefully designed, their critical intermediate steps (e.g., anomaly detection methods and similarity functions) are completely unsupervised. Thus, they can be confused by irrelevant abnormal changes in other metrics, which are caused by noises or fluctuation, especially when the number of metrics or failure units is large. In contrast, *DéjàVu* learns to focus on important metric patterns from historical failures. Second, JSS'20 and iSQUAD localize faulty failure units by taking the ground truths of similar failures. However, there may not be a historical failure having the same faulty failure unit as the incoming failure, especially when there are lots of failure units. In such cases, the faulty failure units' ranks are half of the number of failure units.

For decision tree, its performance is poor because 1) there are too many useless and noisy time-series features and 2) the capacity of decision trees is much lower than deep learning models.

The two random walk-based methods perform poorly as their underlying intuition (i.e., the correlation of metrics faithfully reflects the probability of failure propagation) could not hold in all situations. Furthermore, for RandomWalk@Metric, it is hard to construct the causal graph from scratch, especially when the number of metrics is large. For RandomWalk@FI, there lacks an appropriate method to calculate the correlations of two groups of different metrics. Finally, as well as JSS'20 and iSQUAD, they are unsupervised and can be affected by noises.
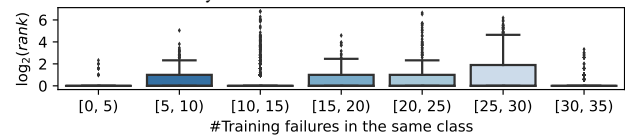

Figure 8: Sensitivity to frequencies of training failures

In Fig. 8, we group testing failures in all datasets by the number of training failures in the same classes (i.e., the failure classes of their faulty failure units) and show the recommended ranks of ground truths of each group. The results show that *DéjàVu* does not perform worse as the number of training failures in the same class decreases. Note that the number of outliers differs because the number of failures in each group differs a lot.

In summary, *DéjàVu* is effective in localizing faulty failure units for recurring failures so as to save much effort for engineers and reduce time to mitigate. Particularly, the performance on real-world failures demonstrates the effectiveness in real-world scenarios.

5.2.2 Contribution of Main Modules. We study the contribution of the main modules used in *DéjàVu* by removing each of them. In Table 3, we compare complete *DéjàVu* with *DéjàVu* without GRU feature extractor (i.e., using 1-D CNN directly, denoted as *DéjàVu w/o GRU*), *DéjàVu* without feature aggregator (i.e., directly feeding the unit-level features into the final classifier, denoted as *DéjàVu w/o AGG*), and *DéjàVu* without class balancing (*DéjàVu w/o BAL*). In all datasets, the MAR of *DéjàVu* outperforms *DéjàVu w/o GRU* by 19.07%~80.25%, outperforms *DéjàVu w/o AGG* by 3.19%~30.92%, and outperforms *DéjàVu w/o BAL* by 3.85% ~30.70%. By further calculation, on all datasets, the average MAR of *DéjàVu* outperforms *DéjàVu w/o GRU* by 69.03%, outperforms *DéjàVu w/o AGG* by 20.45%, and outperforms *DéjàVu w/o BAL* by 15.66%, and the improvement is significant. Thus, generally speaking, all three modules contribute significantly to the overall performance.

In particular, there is a small improvement compared with *DéjàVu w/o AGG* on C because, as introduced in § 5.1.1, we connect all failure units to a virtual vertex on the FDG of *sysC*. The result shows that modeling failure propagation contributes little without meaningful relationships among the failure units. *DéjàVu* can still achieve high performance in such a case because, in this dataset, the engineers are concerned about the metrics indicating more waiting events and consider little failure propagation.

On $\mathcal{B}$ and $\mathcal{D}$, the improvement over *DéjàVu w/o BAL* is small and insignificant. It is mainly because in $\mathcal{B}$ and $\mathcal{D}$, the numbers of failures in different classes are close to each other, and there are still many failures in the most minor class.

With respect to top-1 accuracy, the performance of *DéjàVu* keeps similar or slightly poorer than a variant without any one of these modules. For feature extractor and aggregator, the gradient vanishing problem causes the extracted features to tend to be similar to each other [36]. Thus, the features of ground truths get slightly harder to be distinguished in the first place in some cases. For class balancing, the modification on training data introduces extra noises and could incur performance degradation. However, the degradation is slight, and *DéjàVu* always performs well.

In conclusion, the individual modules in *DéjàVu* (GRU feature extractor, feature aggregator, and class balancing) in *DéjàVu* indeed contribute to the overall performance.

5.3 RQ2: Performance in Various Situations

We investigate how *DéjàVu* performs with varying or incomplete FDGs, insufficient training data, and different model architectures. Due to the space limit, without loss of much generality, we present results on part of the datasets.
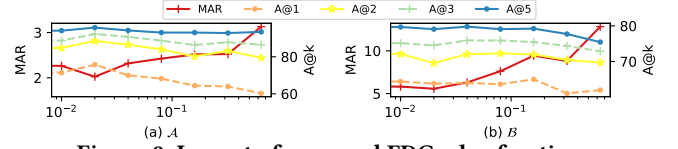

Figure 9: Impact of removed FDG edge fractions

5.3.1 Varying and Incomplete FDGs. The FDGs collected for a system could be incomplete and vary over time due to insufficient knowledge or software/deployment changes. To evaluate *DéjàVu* under such cases, we randomly remove a fraction of FDG edges for each failure and then train and evaluate *DéjàVu*. We repeated the experiments ten times. As shown in Fig. 9, as the fraction increases, the performance degrades, but the degradation is slight when the fraction is not greater than 10%. When too many edges are removed, the performance is worse than *DéjàVu w/o AGG* in Table 3 because the random FDGs give wrong structural information for feature aggregation. In conclusion, our approach can handle varying FDGs and achieve relatively good performance with incomplete FDGs.
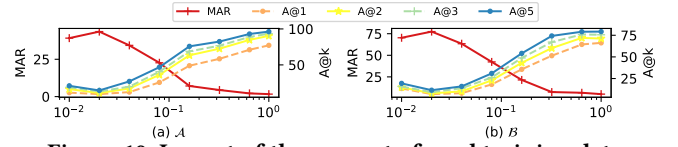

Figure 10: Impact of the percent of used training data

5.3.2 Insufficient Training Data. In Fig. 10, we present the performance when only part of the training data is used. As the training set grows larger, *DéjàVu* performs better in all datasets. When more than 50% of historical failures are used, the performance increases slowly. A possible explanation is that we require only a few training failures for each failure class due to the generalizability of *DéjàVu*. Thus, it is possible to achieve good enough performance with much fewer training data.
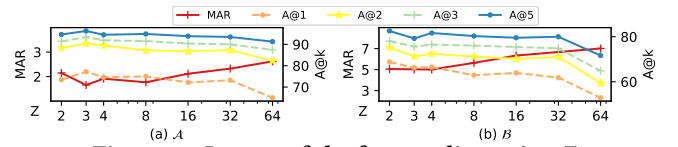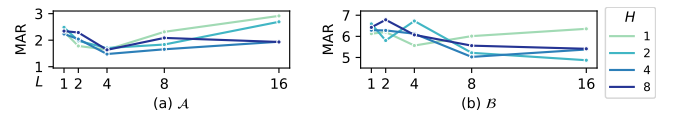

Figure 11: Impact of the feature dimension Z

Figure 12: Impact of feature aggregator architecture

5.3.3 Different Model Architectures. The model architecture is controlled by Z (the dimension of features), H (#GAT heads), and L (#GAT layers). As shown in Fig. 11, the performance keeps steady as Z changes in a large range. When Z is small, the feature extractors and aggregators can be considered to use their former layers to extract/aggregate features and use their latter layers to help map the features into suspicious scores. Therefore, though the feature is of small dimension, it can still keep enough information for the classifier. In summary, it is acceptable to set Z to our default value (3) when applying *DéjàVu* on other systems. Moreover, as shown in Fig. 12, using multiple heads and layers improves the performance compared with vanilla GAT in general. However, the performance can degrade when H or L is too large. The best setting of H and L

differ in different datasets. In practice, we set H and L to the default values (4 and 8, respectively), and, when necessary, we can fine-tune them in each online service system for better performance.

In conclusion, *DéjàVu* performs well in various situations.
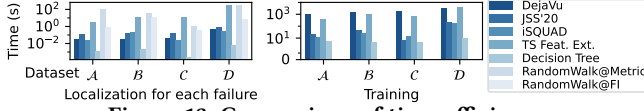
5.4 RQ3: Efficiency

Figure 13: Comparison of time efficiency

We compare the localization and training time with the baselines on servers with Xeon E5-2680 v4, 30G RAM, and GeFore 2080Ti. As shown in Fig. 13, *DéjàVu* produces localization results for a failure within one second, which is enough in practice. Compared with manual localization (several minutes to tens of minutes), *DéjàVu* can automatically diagnose a failure much more rapidly. Furthermore, though it costs more time to train a *DéjàVu* model than most baselines, the overall time consumption is not large (tens of minutes). Moreover, offline training does not affect the efficiency of online localization. Note that the two random-walk methods have no training stage. In conclusion, our approach has acceptable training time and negligible localization time, indicating it is practical.
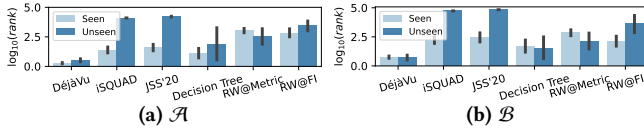
5.5 RQ4: Generalization

Figure 14: The recommended ranks of the ground truths of previously seen/unseen failures. RW denotes RandomWalk.

We evaluate the generalizability of *DéjàVu* by comparing the performance of previously seen (the ground truths are faulty in some historical failures) and unseen failures. As shown in Fig. 14, for *DéjàVu*, the ranks of ground truths of previously unseen failures are close to those of previously seen failures. In contrast, iSQUAD and JSS'20, perform significantly worse on previously unseen failures. It is because they cannot recommend failure units other than those faulty in some historical failures. A decision tree is shared inside each failure class and thus, can generalize in many cases. RW@Metric and RW@FI can also generalize since they are heuristic methods and do not rely on historical failures. However, their overall performance is poor compared with *DéjàVu*. In conclusion, *DéjàVu* has good generalizability to localize faulty failure units effectively for previously unseen failures.

5.6 RQ5: Interpretation

5.6.1 Global Interpretation. We compare our global interpretation method with two widely-used methods, LR (logistic regression) and LIME [48, 51]. In Fig. 15, we present the decision tree given by our interpretation method for *OS Network* failures in $\mathcal{A}$. It shows reasonable rules for discovering network-related issues. For example, the path Ⓐ→Ⓑ leads to most normal failure units, and its corresponding rule is that range_count(Sent_queue, max=1, min=-1)>16.5→normal. When most values of *Sent_queue* are within the

normal range (i.e., [-1, 1]), the send queue is not stuck, and the network works as expected. The path Ⓐ→Ⓒ→Ⓓ→Ⓔ leads to most faulty failure units. The first two conditions (Ⓐ and Ⓒ) mean that many values of *Sent queue* are large, indicating the send queue gets stuck. The third one (Ⓓ) means that the *ss_total* is abnormal. We also confirmed with the engineers that the decision tree is reasonable for this dataset.

For LR, we train LR models with exactly the same features and targets as the decision trees in our interpretation method and take the coefficients of the features as feature importances. In Fig. 16, we present the top-5 feature importances for *OS Network* failures in $\mathcal{A}$. Compared with decision tree, logistic regression cannot give illustrative rules but only importances.

LIME interprets individual predictions by learning a local linear approximation of the model to interpret. In Fig. 17, we present the average metric importances by LIME on all *OS Network* failures in $\mathcal{A}$ for global interpretation. LIME gives only importance scores as well as LR, which are not illustrative. Furthermore, LIME gives very different metric importances for similar failures, which could be confusing (see § 5.6.2).
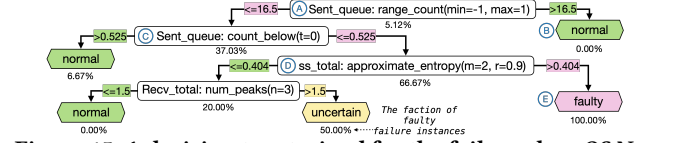

Figure 15: A decision tree trained for the failure class *OS Network* of $\mathcal{A}$. Note that the metrics values are normalized.
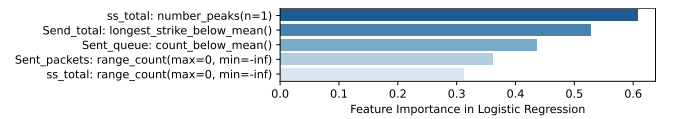

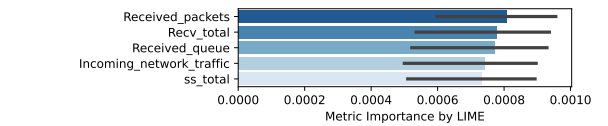**Figure 16: The top-5 feature importances given by LR for the failure class *OS Network* of $\mathcal{A}$**

Figure 17: The top-5 average metric importances given by LIME on all *OS Network* failures in $\mathcal{A}$
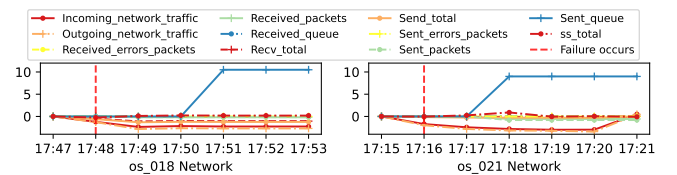

Figure 18: The normalized metrics of the ground truth of a failure (left) and its top-1 similar failure (right).

5.6.2 Local Interpretation. We compare our local interpretation method with GNNExplainer [58], a state-of-the-art model-agnostic interpretation method for graph neural networks, and LIME. In Fig. 18, we present a failure of *OS Network* in $\mathcal{A}$ (T_1) and its top-1 similar failure found by our method (T_2). By presenting T_2 to engineers, engineers can understand that we recommend *os_018 Network* for T_1 because its failure scenario is similar to that of T_2 in *DéjàVu*'s perspective. With the help of the failure ticket of T_2, engineers could easily understand the underlying failure scenario

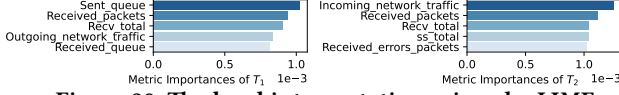**Figure 19: The local interpretations given by GNNExplainer**

Figure 20: The local interpretations given by LIME

of T_1 and find correct mitigation process. As a result, the time to diagnose and mitigate can be saved. For comparison, we present the interpretations of T_1 (left) and T_2 (right) from GNNExplainer in Fig. 19. GNNExplainer can only show engineers the important neighbors (green) of the ground truth (red), which requires further efforts to discover why they are important. In Fig. 20, we present the top-5 metric importances of T_1 and T_2 given by LIME. LIME gives very different metric importances for the two similar failures, making engineers confused.

In summary, *DéjàVu* provides more helpful interpretation.

6 DISCUSSION

6.1 Lessons Learned

We summarize some lessons learned from our industrial experience in *compA*. First, the current industrial practice of fault localization relies on the hard-earned intuition from engineers' diagnosing experience, inspired by which we explore **learning from historical failures** in this paper. The superiority of *DéjàVu* over the heuristic or unsupervised methods demonstrates that learning from historical failures is effective for fault localization. Second, **periodical re-training** is necessary in practice due to the frequent software and deployment change. In Fig. 5, we propose re-training the *DéjàVu* model after mitigating a new failure, which costs only tens of minutes at most each time. Besides, *DéjàVu* supports using a new FDG, even with new failure units, for each new failure. Finally, in practice, there are also many **non-recurring failures**. Since they have not occurred previously, the suspicious scores of the recommended root causes by *DéjàVu* would be small. In such cases, we will warn engineers of the potential non-recurring failures.

6.2 Limitations and Future Works

Not only the faulty metric names but also their patterns are helpful to determine which kind of failure occurs. Thus, we can integrate **metric patterns** in the definition of failure units in the future.

Engineers usually expect failure localization solutions compatible with their **existing knowledge**. Currently, engineers can integrate it into *DéjàVu* by labeling ground truths of historical failures or manually specifying FDGs and failure units. However, such integration is neither flexible nor accurate enough for all kinds of diagnosis knowledge or rules, which is also a limitation of *DéjàVu*.

In different online service systems, engineers may define completely different failure units, so currently, we cannot transfer a trained model to another system. However, **transfer learning** in similar systems could be our future work.

Though this paper focuses on metrics as discussed in § 2.2, we can design appropriate feature extractors for **more data sources** (e.g., logs or traces) to utilize them in the future.

6.3 Threats to Validity

The *internal* threat to validity mainly lies in the implementation of our approach and baselines. To reduce this threat, our implementation is based on mature frameworks (§ 5.1.3), and the authors carefully checked and tested the code.

The *external* threat to validity mainly lies in the study subjects. We evaluate *DéjàVu* on four systems, which cannot represent all online service systems. But we believe our approach is general enough for two reasons. First, the four systems are from different companies and have completely different architectures. Particularly, *DéjàVu* is even applicable to traditional software systems (Oracle database for C). Second, our approach is applicable to other online service systems, as the input data of it (e.g., metrics, historical failures, and the relationships between components) are common, and *DéjàVu* is scalable as discussed in § 5.4. In the future, we will reduce this threat by evaluating *DéjàVu* on more systems.

The *construct* threat to validity mainly lies in the hyperparameters and evaluation metrics. We tuned the hyperparameters for *DéjàVu* and baselines by grid-search following existing works [12]. We also use widely used evaluation metrics (see § 5.1.4).

7 RELATED WORK

Fault localization. Most existing fault localization methods [10, 25, 31, 42, 55] are unsupervised and heuristic. For example, MonitorRank [31] takes the historical and current metrics as its input and ranks all root cause candidates with random walk strategy on call graphs. Prior works also use historical failures to find similar historical failures [7, 8, 45, 50] or train supervised localization models [20, 41, 63]. For example, MEPFL [63] treats faulty microservice localization for traces as a multi-class classification problem and directly builds supervised machine learning models (e.g., Random Forest [9]) for it. Though not directly localizing root causes in a supervised manner, some prior works [44] utilize historical failures to tune parameters. Other than metrics, logs [27, 39, 49] and traces [38, 59, 63] are extensively studied for fault localization by many prior works.

Interpretability. Most existing fault localization methods [7, 8, 20, 41, 50, 62] do not provide interpretation explicitly. iSQUAD [45] interprets localization results locally by finding representative historical failures of the assigned clusters [30]. Fingerprint [7] and JSS'20 [8] work by finding similar historical failures. As discussed in § 4.1, many existing interpretation methods for deep-learning models [6, 19, 60] focus on understanding the inner model mechanism and do not meet our requirements.

Duplicated bugs. Traditional duplicated bugs retrieval methods [? ? ? ?] find similar bug reports with machine learning and natural language processing techniques. In contrast, fault localization for online service systems focuses on metrics, as summarizing failure reports is time-consuming for rapid diagnosis. Furthermore, we must model the dependencies among components and achieve generalizability, which is less important in traditional monolithic software systems.

Deep learning-based program debugging aims to localize the faulty code elements by deep learning techniques with various features, such as spectrum-based suspiciousness, mutation-based suspiciousness, and complexity-based fault proneness [29, 34, 37,

57]. In contrast, the faulty failure units we are concerned about could be either software bugs (e.g., when the components are services), deployment issues, or hardware failures.

8 CONCLUSION

This paper proposes an actionable and interpretable fault localization approach, *DéjàVu*. Inspired by the manual localization process, we learn a localization model from historical failures of a given online service system. The trained model online recommends faulty failure units and presents the corresponding interpretation to engineers when a failure occurs. In this way, engineers can recognize the underlying failure locations and types and take action rapidly to ensure the quality of software services. More specifically, we design a novel deep learning-based model, which utilizes GRU-based feature extractors to represent various failure units uniformly and GATs to model failure propagation based on the FDGs. Our designed model can generalize to a previously unseen failure once there exist historical failures of the same class. We also design two novel local and global interpretation methods. An extensive study on four systems demonstrates the effectiveness (the average MAR of 1.66~5.03) and efficiency (less than one second localization time for one failure) of *DéjàVu*. The results also confirm the contributions of the main modules in *DéjàVu*. Particularly, the results on production systems and real-world failures demonstrate our practical performance.

REFERENCES

[1] [n. d.]. Anonymous-Submission-22/DejaVu. https://github.com/anonymous-submission-22/DejaVu

[2] 2021. PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration. pytorch. https://github.com/pytorch/pytorch

[3] 2021. Tsfresh. Blue Yonder GmbH. https://github.com/blue-yonder/tsfresh

[4] 2022. Chaos-Mesh/Chaos-Mesh. Chaos Mesh. https://github.com/chaos-mesh/chaos-mesh

[5] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. 2018. Sanity Checks for Saliency Maps. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*.

[6] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. Network Dissection: Quantifying Interpretability of Deep Visual Representations. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*.

[7] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. 2010. Fingerprinting the Datacenter: Automated Classification of Performance Crises. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*.

[8] Álvaro Brandón, Marc Solé, Alberto Huélamo, David Solans, María S. Pérez, and Victor Muntés-Mulero. 2020. Graph-Based Root Cause Analysis for Service-Oriented and Microservice Architectures. *Journal of Systems and Software* 159 (Jan. 2020).

[9] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001).

[10] Yang Cai, Biao Han, Jie Li, Na Zhao, and Jinshu Su. 2021. ModelCoder: A Fault Model Based Automatic Root Cause Localization Framework for Microservice Systems. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*.

[11] Junjie Chen, Xiaoting He, Qingwei Lin, Yong Xu, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. An Empirical Investigation of Incident Triage for Online Service Systems. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*.

[12] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. Continuous Incident Triage for Large-Scale Online Service Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[13] Zhuangbin Chen, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, Yingnong Dang, Feng Gao, Pu Zhao, Bo Qiao, Qingwei Lin, Dongmei Zhang, and Michael R Lyu. 2020. Towards Intelligent Incident Management: Why We Need It and How We Make It. (2020).

[14] Zhuangbin Chen, Jinyang Liu, Yuxin Su, Hongyu Zhang, Xiao Ling, Yongqiang Yang, and Michael R. Lyu. 2022. Adaptive Performance Anomaly Detection for Online Service Systems via Pattern Sketching. In *The 44th International Conference on Software Engineering (ICSE 2022)*. arXiv:2201.02944

[15] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[16] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W. Kempa-Liehr. 2018. Time Series FeatuRe Extraction on Basis of Scalable Hypothesis Tests (Tsfresh - A Python Package). *Neurocomputing* 307 (2018).

[17] J. Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates.

[18] Yingnong Dang, Qingwei Lin, and Peng Huang. 2019. AIOps: Real-World Challenges and Research Innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*.

[19] Vincenzo Di Cicco, Donatella Firmani, Nick Koudas, Paolo Merialdo, and Divesh Srivastava. 2019. Interpreting Deep Learning Models for Entity Resolution: An Experience Report Using LIME. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '19)*.

[20] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Kumar Saini, George Varghese, and Ravi Netravali. 2021. Revelio: ML-Generated Debugging Queries for Distributed Systems. *arXiv:2106.14347 [cs]* (June 2021). arXiv:2106.14347 [cs]

[21] Vincent Dumoulin and Francesco Visin. 2016. A Guide to Convolution Arithmetic for Deep Learning. *arXiv preprint arXiv:1603.07285* (2016). arXiv:1603.07285

[22] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical and Scalable ML-driven Performance Debugging in Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

[24] Jindong Gu and Volker Tresp. 2019. Semantics for Global and Local Interpretation of Deep Neural Networks. (Oct. 2019).

[25] 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*.

[26] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *31st Conference on Neural Information Processing Systems*.

[27] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2018. Identifying Impactful Service System Problems via Log Analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*.

[28] Dan Hendrycks and Kevin Gimpel. 2020. Gaussian Error Linear Units (GELUs). *arXiv:1606.08415 [cs]* (July 2020). arXiv:1606.08415 [cs]

[29] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. 2021. Deep Transfer Bug Localization. *IEEE Transactions on Software Engineering* 47, 7 (July 2021).

[30] Been Kim, Cynthia Rudin, and Julie Shah. 2014. The Bayesian Case Model: A Generative Approach for Case-Based Reasoning and Prototype Classification. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*.

[31] Myunghwan Kim, Roshan Roshan, and Sam Shah. 2013. Root Cause Detection in a Service-Oriented Architecture. *ACM SIGMETRICS Performance Evaluation Review* (June 2013).

[32] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[33] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

[34] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug Localization with Combination of Deep Learning and Information Retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*.

[35] I. Lee and R.K. Iyer. 2000. Diagnosing Rediscovered Software Problems Using Symptoms. *IEEE Transactions on Software Engineering* 26, 2 (2000).

[36] Guohao Li, Matthias Muller, Ali Thabet, and Bernard Ghanem. 2019. DeepGCNs: Can GCNs Go as Deep as CNNs?. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE Computer Society.

[37] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2019*.

[38] 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)*.

[39] Q. Lin, H. Zhang, J. Lou, Y. Zhang, and X. Chen. 2016. Log Clustering Based Problem Identification for Online Service Systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*.

[40] Dewei Liu. 2020. MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems. In *ICSE 2021 Software Engineering in Practice*.

[41] Ping Liu, Yu Chen, Xiaohui Nie, Jing Zhu, Shenglin Zhang, Kaixin Sui, Ming Zhang, and Dan Pei. 2019. FluxRank: A Widely-Deployable Framework to Automatically Localizing Root Cause Machines for Software Service Failure Mitigation. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*.

[42] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. 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)*.

[43] Jian-Guang Lou, Qingwei Lin, Rui Ding, Qiang Fu, Dongmei Zhang, and Tao Xie. 2013. Software Analytics for Incident Management of Online Services: An Experience Report. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[44] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. 2020. AutoMAP: Diagnose Your Microservice-based Web Applications Automatically. In *Proceedings of The Web Conference 2020 (WWW '20)*.

[45] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, Feifei Li, Changcheng Chen, and Dan Pei. 2020. Diagnosing Root Causes of Intermittent Slow Queries in Cloud Databases. *Proceedings of the VLDB Endowment* 13, 8 (April 2020).

[46] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Rui Xin. 2020. Predicting Failures in Multi-Tier Distributed Systems. *Journal of Systems and Software* 161

(March 2020).

[47] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. 2020. Interpreting Deep Learning-Based Networking Systems. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication.*

[48] Emanuel Metzenthin. 2022. LIME For Time. https://github.com/emanuel-metzenthin/Lime-For-Time

[49] 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.*

[50] Cuong Pham, Long Wang, Byung Chul Tak, Salman Baset, Chunqiang Tang, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2017. Failure Diagnosis for Distributed Systems Using Targeted Fault Injection. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS* 28, 2 (2017).

[51] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.).

[52] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. 2017. Sieve: Actionable Insights from Monitored Metrics in Distributed Systems. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17).*

[53] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.*

[54] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019). arXiv:1909.01315

[55] Canhua Wu, Nengwen Zhao, Lixin Wang, Xiaoqin Yang, Shining Li, Ming Zhang, Xing Jin, Xidao Wen, Xiaohui Nie, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2021. Identifying Root-Cause Metrics for Incident Diagnosis in Online Service Systems. In *2021 IEEE 32th International Symposium on Software Reliability Engineering (ISSRE).*

[56] Li Wu, Johan Tordsson, Jasmin Bogatinovski, Erik Elmroth, and Odej Kao. 2021. MicroDiag: Fine-grained Performance Diagnosis for Microservice Systems. In *ICSE21 Workshop on Cloud Intelligence.*

[57] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2021. A Survey on Deep Learning for Software Engineering. *Comput. Surveys* (Dec. 2021).

[58] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. Gnnexplainer: Generating Explanations for Graph Neural Networks. *Advances in neural information processing systems* 32 (2019).

[59] Guangba Yu, Pengfei Chen, Hongyang Chen, Zijie Guan, Zicheng Huang, Linxiao Jing, Tianjun Weng, Xinmeng Sun, and Xiaoyun Li. 2021. MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments. (2021).

[60] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 8689)*, David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.).

[61] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. 2019. Robust Log-Based Anomaly Detection on Unstable Log Data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).*

[62] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* (2018).

[63] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent Error Prediction and Fault Localization for Microservice Applications by Learning from System Trace Logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).*

[64] Zhi-Hua Zhou. 2021. *Machine Learning.* Springer Singapore. https://www.springer.com/gp/book/9789811519666