Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks

Conference Paper · October 2020

DOI: 10.1109/ISSRE5003.2020.00014

11 authors, including:

Ping Liu
Capital University of Physical Education and Sports
3 PUBLICATIONS **92** CITATIONS
SEE PROFILE

Qianyu Ouyang
Tsinghua University
2 PUBLICATIONS **62** CITATIONS
SEE PROFILE

Shenglin Zhang
Nankai University
52 PUBLICATIONS **1,063** CITATIONS
SEE PROFILE

Jiahai Yang
Tsinghua University
195 PUBLICATIONS **1,186** CITATIONS
SEE PROFILE

Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks

Ping Liu [‖¶], Haowen Xu [‖¶], Qianyu Ouyang[‖¶], Rui Jiao[‖¶], Zhekang Chen[§], Shenglin Zhang[‡], Jiahai Yang [‖††¶], Linlin Mo [†], Jice Zeng [†], Wenman Xue [†], Dan Pei* [‖¶]
[‖]Tsinghua University [‡]Nankai University [†]WeBank [§]BizSeer [††]Peng Cheng Laboratory
[¶]Beijing National Research Center for Information Science and Technology (BNRist)

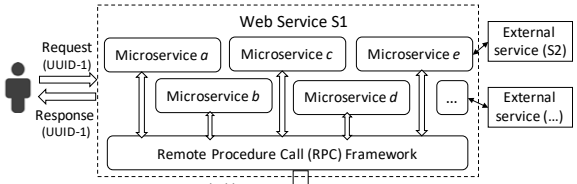
***Abstract*—The anomalies of microservice invocation traces (traces) often indicate that the quality of the microservice-based large software service is being impaired. However, timely and accurately detecting trace anomalies is very challenging due to: 1) the large number of underlying microservices, 2) the complex call relationships between them, 3) the interdependency between the response times and invocation paths. Our core idea is to use machine learning to automatically learn the overall normal patterns of traces during periodic offline training. In online anomaly detection, a new trace with a small anomaly score (computed based on the learned normal pattern) is considered anomalous. With our novel trace representation and the design of deep Bayesian networks with posterior flow, our unsupervised anomaly detection system, called *TraceAnomaly*, can accurately and robustly detect trace anomalies in a unified fashion. *TraceAnomaly* has been deployed on 18 online services in a company *S*. Detailed evaluations on four large online services which contain hundreds of microservices and a testbed which contains 41 microservices show that the recall and precision of *TraceAnomaly* are both above 0.97, outperforming the existing approach in *S* (hard-coded rule) by 19.6% and 7.1%, and seven other baselines by 57.0% and 41.6% on average.**

***Index Terms*—trace; anomaly detection; microservice**

I. Introduction

Recently, microservice [1] architecture has become more and more popular for large-scale Web-based services, but it poses additional challenges to guarantee the reliability of services. This architecture decouples a Web service into multiple microservices. For example, four large online services studied by this paper in company *S* (which is a digital bank in China and serves tens of millions of users) contain 61 to 344 microservices (see Table I). The top of Fig. 1 shows a service in *S*. This service consists of many microservices, and some microservices may call other external services (*e.g.*, payment service of the bank). Each microservice can be individually upgraded, which enables more agile software development and deployment, but makes the troubleshooting more challenging due to the complex call relationships and large scale.

One important step in troubleshooting microservice-based Web services is to detect anomalies (*e.g.*, unexpected response time or invocation path) when they first appear in the microservice invocation traces (recorded by Remote Procedure Call (RPC) tracing framework such as Google Dapper [2]). If an anomalous trace is detected, operators will analyze the microservices along the trace to localize the root cause. Fig. 1 shows how a simple real trace for one specific user request is collected. When a user request is received, a Web service will generate a UUID (Universally Unique Identifier) that uniquely identifies all messages for this user request (*e.g.*, UUID-1 in Fig. 1). When microservice m calls microservice n, m sends a call message to n (denoted as *call(m→n)*). After the processing at n is completed, n sends the response message to m (denoted by *response(n→m)*). Both messages are routed between m and n via the RPC framework, thus the tracing mechanism can naturally record these messages with timestamps and UUIDs. All messages with the same UUIDs constitute one **microservice invocation trace** (**trace** for short hereinafter). The table in Fig. 1 shows a real trace with *UUID-1*. Note that call messages to external service do not go through the RPC framework, thus are not monitored.



Message order	UUID	Sending time of (m → n) at m (msec)	Receiving time of (m → n) at n (msec)	Message (m→n)
①	UUID-1	-	1519747202138	call(start→a)
②	UUID-1	1519747202144	1519747202146	call(a→b)
③	UUID-1	1519747202149	1519747202151	call(b→c)
④	UUID-1	1519747202149	1519747202150	call(b→d)
⑤	UUID-1	1519747202155	1519747202156	*response*(c→b)
⑥	UUID-1	1519747202159	1519747202160	call(d→e)
⑦	UUID-1	1519747202188	1519747202190	*response*(e→d)
⑧	UUID-1	1519747202194	1519747202196	*response*(d→b)
⑨	UUID-1	1519747202253	1519747202256	call(b→e)
⑩	UUID-1	1519747202323	1519747202324	*response*(e→b)
⑪	UUID-1	1519747202355	1519747202356	*response*(b→a)
⑫	UUID-1	1519747202360	-	*response* (a→end)

Fig. 1: An example trace with UUID-1. The first column is the message order based on the timestamps. The sending time of first call message (message ①) and the receiving time of last response message (message ⑫) are not collected, which are denoted by "-".

Given that the trace data are relatively new, the deployed anomaly detection approaches in the industry are still manual rules based, to the best of our knowledge. after a call ends,

*Dan Pei is the correspondence author.

each microservice will return a code reflecting the running status of the microservice. Therefore, operators have designed some fixed rules based on these return codes to detect anomalies of traces. However, a Web service may contain hundreds of microservices (*e.g.*, Service-1 in company *S* contains 344 microservices, see Table I), and the values and meanings of these microservices' return codes can change over time, making the rule-based maintenance very difficult.

To solve these problems, we proposed *TraceAnomaly*, which can automatically learn the complex patterns of traces, and the anomalous traces can be detected when their patterns deviate from those of normal traces.

A. Related work

Multimodal LSTM [3] proposed a multimodal long short-term memory (LSTM) model to learn the sequential nature of both response time and calls in the normal traces. A trace is anomalous if its pattern deviates from the learned normal patterns. However, our evaluation in §V shows that this approach has difficulty learning the relationships between response time and invocation paths in an end-to-end way, thus its performance suffers. Furthermore, this approach lacks interpretability for the anomalous trace: *i.e.*, which invocation paths are anomalous, or which microservices' response time is anomalous. AEVB [4] only focuses on the response time anomalies of traces. It proposed an unsupervised deep Bayesian networks model to detect the response time anomalies of tracing data. However, this approach trains one model for each microservice, and a trace is considered anomalous if one of its constituent microservice's response time is detected as abnormal. Thus, its training overhead is too high: if a large service contains hundreds of microservices (*e.g.*, Service-1 in Table I), then hundreds of models need to be trained.

Due to the difficulty of collecting trace data (the service's framework has to be changed) in legacy software services, some previous works [5]–[8] extract log keys from system log files firstly, then construct traces based on log keys. WFG-based [5] approach detects trace anomalies by an offline learned workflow graph (WFG), which is a Finite State Automaton (FSA) model. A trace has an execution path anomaly if it cannot be accepted by the WFG. Meanwhile, WFG-based [5] approach uses the 3-sigma method to detect time consumption anomalies for each transition in the WFG. CFG-based [6] approach detects anomalies of traces by an offline learned control flow graph. A trace is anomalous when none of the children of a parent node is seen within an expected time lag interval. CPD-based [7] approach detects anomalies by checking the similarity between different traces' control flow graph. The similarity is calculated based on the conditional probability distribution of the next symbol given a preceding log key sequence. A trace is anomalous when the trace's similarity is too small. DeepLog [8] uses a neural network model utilizing LSTM to detect execution path anomaly. It also trains an LSTM model for each node to detect time consumption anomaly.

B. Challenges, Core Ideas and Contributions

Although aforementioned related works [3]–[8] have been proposed in the literature, none of them has been deployed in reality, due to high overhead and/or low accuracy (which will be detailed in §V). We now present the main challenges, our core ideas and major contributions.

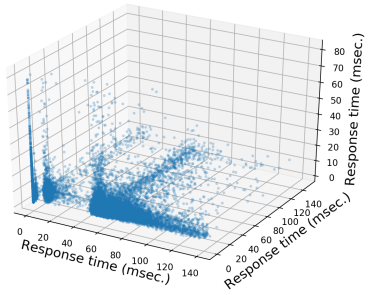


Fig. 2: The distribution of traces' response time without anomalies from a small online service in company *S*. This service only contain three microservices, and the three axes denote the three microservices' response time, respectively.

Challenge 1 and our core idea: unify response time and invocation paths of traces in an interpretable way for anomaly detection. We have this challenge because: 1) the response time of a *microservice* can change significantly without being anomalous. Fig. 2 shows the distribution of traces' response time without anomalies from a small online service. Although there are only three microservices in this service, the response time varies significantly. For a microservice, its response time is determined by both itself and the invocation path from the entrance microservice to it. For example, microservice e in Fig. 1 is invoked twice, with different response time (see Fig. 4), making the microservice-level modeling in AEVB [4] ineffective. This mandates that response time modeling must consider the invocation path from the service entrance to a microservice. However, the number of individual paths is large (*e.g.*, hundreds for one service, and there are hundreds of services in company *S*). This scale makes it infeasible to directly apply existing time series anomaly detection on path *individually* because these algorithms either require operators to manually pick algorithm and tune parameters for each path [9]–[27], or need a dedicated learned model for each path [28], [29]. 2) An anomalous trace and a normal trace might have the same structure, and only the difference of response time can help distinguish them (see a detailed example in Fig. 6). 3) When an anomalous trace is detected, the root cause of the trace has to be localized quickly. So the unified way must be interpretable to help operators localize root cause, which Multimodal LSTM [3] cannot do.

The above difficulties call for an anomaly detection approach which unifies response time and invocation paths of traces *at the trace level*, as opposed to *individual node, edge, or path level*. However, most previous works [4]–[8], [30] do not try a unified detection approach. Although Multimodal LSTM [3] tries a unified detection approach, it lacks interpretability for the anomalous trace, and has difficulty learning the relationships between response time and invocation paths.

Our core idea is to treat each trace for a service as one training sample, and use machine learning to capture the overall patterns of the traces of a *service*. Therefore, we have one model for each service, as opposed to one model for each *microservice* in AEVB [4]. To this end, we encode the response time and invocation paths of a trace in a service into

a vector (called **service trace vector** or **STV** hereinafter), a necessity required by most mature deep learning algorithms. The encoded information should not only represent the traces' overall patterns, but also can be easily interpreted by the operators for root cause localization. Indeed, our interpretable service trace vector enables us to develop a straightforward but effective root cause localization algorithm. This is why we choose to handcraft the service trace vector design with physical significance, as opposed to using network/graph representation learning (embedding) [3] to automatically learn the vector which are not easy to interpret.

Challenge 2 and our core idea: designing an accurate, robust, unsupervised learning architecture that captures the characteristics of complex patterns of traces, with reasonable training overhead. As described in Challenge 1, a microservice's response time is related to both itself and its invocation path. For a single service which contains many microservices, there can be hundreds of individual paths, thus hundreds of response time distributions need to be learned conditional on their invocation paths. With such a complexity, a high-capacity model is needed. Furthermore, because anomaly labels are infeasible to obtain in such a complex context with vast amount of data, we have to use an unsupervised algorithm. Above requirements overall call for an unsupervised high-capacity model, such as deep Bayesian networks. The model should also be robust in that its performance is not sensitive to hyper-parameters. Our core idea to tackle this challenge is to apply posterior flow [31], which can use nonlinear mappings to increase the complexity of the latent variables in the Bayesian networks, allowing the model to capture the complex patterns in a robust, accurate and unsupervised manner.

Our ideas in a nutshell: (Fig. 3) Our proposed system *TraceAnomaly* processes each trace as a whole, constructs a service trace vector that encodes both invocation path and response time (*e.g.*, the information in the right table in Fig. 4), then learns the overall normal trace patterns for a service during offline training. After that, in online anomaly detection, for each new trace, an anomaly score is computed based on the learned model of the service, and a trace with a small score is considered anomalous. Finally, the root cause of the anomalous trace will be localized by an algorithm based on the service trace vector.

The contributions of this paper are summarized as follows:

Contribution 1. We propose a novel approach to construct feature vectors for traces of a service, called service trace vector (STV), which efficiently encodes both the response time information and the invocation path information of traces, and enables both effective learning at the service-level, accurate anomaly detection at the trace level, and effective localization at the microservice level. **We believe STV can be used as the trace representation for other ML-based trace anomaly detection and localization algorithms beyond the ones proposed in this paper.**

Contribution 2. We propose *TraceAnomaly*, an unsupervised deep learning algorithm which can learn the complex trace patterns in a service and accurately detect trace anomalies, with our trace representation and our design of deep Bayesian networks with posterior flows. *TraceAnomaly* has been deployed on 18 online services in company *S* which serves tens of millions of users, and is **the first deployed trace anomaly detection approach based on machine learning**, to the best of our knowledge.

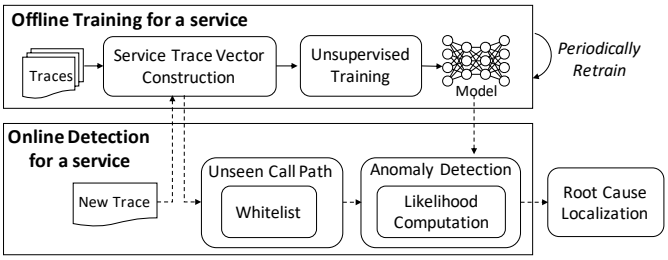


Fig. 3: The architecture of TraceAnomaly. Solid lines denote offline flow, and dashed lines denote online flow.

Contribution 3. Detailed evaluations on four large online services which contain hundreds of microservices and a TrainTicket [32] testbed which contains 41 microservices show that the recall and precision of *TraceAnomaly* are both above 0.97, outperforming the existing approach in *S* (hard-coded rule) by 19.6% and 7.1%, and seven other baselines by 57.0% and 41.6% on average. Moreover, we have open-sourced the prototype of TraceAnomaly [33] to help researchers better understand our work.

Contribution 4. We propose a root cause localizing algorithm based on our designed service trace vector, which successfully localized the correct root causes on all 73 anomalous traces from four large services, and its localization accuracy outperforms three baselines by 55% in testbed results.

The rest of the paper is organized as follows. §II presents the overall design of *TraceAnomaly*, the details of service trace vector construction and the root cause localization algorithm. §III presents the design of our deep Bayesian networks with posterior flows. §IV shows the details of deployment. §V evaluates *TraceAnomaly* on four large online services and a testbed. §VI evaluates and analyzes the internal mechanisms of *TraceAnomaly*. §VII concludes the paper.

II. TraceAnomaly Overview, Service Trace Vector Construction and Localization Algorithm

This section first presents the overall *TraceAnomaly* design, then we introduce how to construct service trace vector (STV) for a service. Finally, we present the root cause localization algorithm designed by STV.

A. TraceAnomaly Overall Architecture

Fig. 3 shows the architecture of *TraceAnomaly*. During offline training for a service, training traces are encoded as vectors, denoted by $\mathbf{x}$, using the method shown in §II-C. The vectors are then fed into our designed deep Bayesian networks to learn the distribution $p(\mathbf{x})$ (see §III), and generate a model. To adapt to potential service upgrade, the model will be retrained periodically.

During online detection, each new trace is encoded as a vector. If a trace contains a previously unseen call path (see §II-C), *TraceAnomaly* declares it as an anomaly. The unseen call paths are handled by a whitelist approach (see §IV-B). If there is no unseen call path, then the trained model outputs a likelihood, *i.e.*, $\log p(\tilde{\mathbf{x}})$, for each vector, and use it as the anomaly score. If the anomaly score of a trace is too small, it is considered

Sending time of (m→n) at m (msec)	Message (m→n)
-	call(start→a)
1519747202144	call(a→b)
1519747202149	call(b→**c**)
1519747202149	call(b→**d**)
1519747202159	call(d→e)
1519747202253	call(b→e)

Microservice *s*	Call path of microservice *s* (*s*, call path)	Response time of (*s*, call path) (msec)
a	(a, (start→a))	222
b	(b, (start→a, a→b))	209
c	(c, (start→a, a→b, b→c))	4
d	(d, (start→a, a→b, b→c, b→d))	44
e	(e, (start→a, a→b, b→c, b→d, d→e))	28
e	(e, (start→a, a→b, b→c, b→d, d→e, b→e))	67

Fig. 4: An example of call path extraction (trace in Fig. 1).

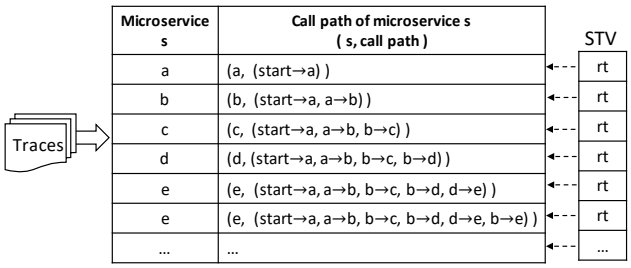


Fig. 5: Process of STV construction.

an anomaly, and a localization algorithm can localize the root cause of the anomalous trace.

B. Extracting call paths and response times from a trace

To address Challenge 1, our design goal for service trace vector is that both the response time pattern and invocation path pattern of traces for a given service can be learned from the service trace vector data. In order to capture "what has happened" in the trace which potentially had impacted on microservices' response time, we propose *call path*.

*A **call path** of a microservice s in a trace, denoted as (s, call path), is the sequence of call messages (sorted by sending time) before s is called*. Fig. 4 shows the process of extracting the call path for each microservice in a trace, where a call message is denoted by (*m*→*n*). First, all call messages are sorted by their sending time. If two call messages have the same sending time, then they are sorted by the IDs of callees (to ensure the uniqueness of the order). As shown in the left half of Fig. 4, the sending time of call($b \rightarrow c$) and call($b \rightarrow d$) are the same, then the two call messages are sorted by the alphabetical order (c is ranked ahead of d). Second, for a callee microservice *n*, the call message sequence starts from the service entrance call message and ends at the current call message (*m*→*n*), which represents the call path of the callee microservice *n*. Fig. 4 shows call paths extracted from the trace in Fig. 1.

The response time rt of (s, call path) can be computed by $rt = srt - rct$, where rct is the receiving time of the call message of s, and srt is the sending time of the corresponding response message. For example, in Fig. 4, the response time of microservice b (*209ms*) is computed using the receiving time of message ② ((*1519747202146* in Fig. 1) and the sending time of message ⑪ (*1519747202355* in Fig. 1). Note that the last two rows of the right table in Fig. 4 show that microservice e's two different call paths can have different response times.

C. Service Trace Vector Construction

We now introduce the details of encoding a trace as service trace vector (STV). We choose to handcraft the vector design based on our domain knowledge of trace (see §II-B), as opposed to applying representation learning, so that the resulting vector has physical significance, thus is interpretable. We believe STV can be used as the trace representation for other ML-based trace anomaly detection and localization algorithms beyond the ones proposed in this paper.

Before the periodic training time (see Fig. 3), all call paths of the training traces from a service are extracted via the method shown in §II-B. The set of all unique call paths form a *call path list* of the service. Given the large amount of training trace data, we assume that all existing "normal" call paths are covered by the *call path list* that we constructed during training time. Because the training set is updated at the periodic training, the *call path list* is also updated periodically. As the online service evolves, historical invalid call paths in the *call path list* will disappear, and new call paths will occur.

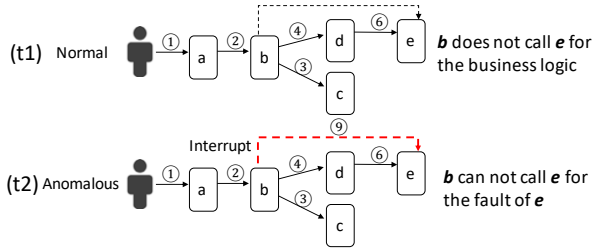
A specific trace's STV is constructed with the *call path list*. As shown in Fig. 5, each (s, call path) of *call path list* is the *dimension* ID of the STV, and the value of each dimension is the response time of corresponding (s, call path). If a (s, call path) in *call path list* is not included by the specific trace, then the value of the dimension corresponding to the (s, call path) not included is set to -1, which denotes an *invalid dimension* of the specific trace's STV; Otherwise, the dimension whose value is not -1 is a *valid dimension*. During online detection, if the call path of a new trace is not in the *call path list*, then the call path is an unseen call path, which are handled by a whitelist approach (see §IV-B).

In summary, through call paths, the invocation paths are naturally encoded in the dimension ID of the STV, and the value of each dimension is the response time in a specific trace. Above encoding approach enables *TraceAnomaly* to unify invocation path patterns and response time patterns. Note that, for the same service, two traces might have the same "structure" (*i.e.*, nodes and edges), and only the difference of response times can help distinguish between normal and anomalous traces. As shown in Fig. 6, the structures of trace $t1$ and trace $t2$ are the same, while $t1$ is normal because its business logic to this service demands such a structure, and $t2$ is anomalous because of some fault at microservice e during runtime. The table in Fig. 6 shows the difference of response times. The STV encoding can naturally deal with this by declaring $t2$ as an anomaly.

D. Localizing Root Cause

For an anomalous trace t in a given service C, the **root cause** in our context is defined as a dimension ID of t's STV, *i.e.*, the tuple $(s, callpath)$, where $callpath$ is the call path of microservice s, and s is a problematic/failed internal microservice in C, or a microservice (in C) which calls a problematic/failed external services (not directly monitored) (e.g., microservice e calls an external service $S2$ in Fig. 1). Both above two types of root causes are very helpful to operators. The root cause can be clearly interpreted by the failed microservice s and the corresponding call path. Once a root cause is localized, operators can then just focus on this $(s, callpath)$ to further (maybe manually) figure out the exact root cause (bugs in codes, error configuration *etc.*).

Our root cause localization algorithm is enabled by the clear physical significance of service trace vector (each dimension ID is a call path, and each dimension's value is a response



Microservice s	Call path of microservice s (s, call path)	(t1) Response time of (s, call path) (msec)	(t2) Response time of (s, call path) (msec)
a	(a, (start→a))	222	1002
b	(b, (start→a, a→b))	209	909
c	(c, (start→a, a→b, b→c))	4	6
d	(d, (start→a, a→b, b→c, b→d))	44	53
e	(e, (start→a, a→b, b→c, b→d, d→e))	28	528

Fig. 6: Trace t1 and t2 have the same structures, but t1 is normal while t2 is anomalous.

time). Before we introduce the algorithm details, we first introduce the concept of homogeneous service trace vector (HSTV). If trace t_x's STV has the same *valid dimension*s (see §II-C) as trace t_y's, then t_y's STV is a HSTV of t_x's, and vice versa. Fig. 7 shows an example of a HSTV.

When an anomalous trace t is detected online, *TraceAnomaly* checks the training set to find all the traces whose STVs are the HSTVs of t's STV. For a valid dimension d of t, we calculate the mean and standard deviation (std) of values from all found HSTVs' dimension d. If the value of the dimension d in t's STV is greater than $mean + 3 * std$ or less than $mean - 3 * std$, then the dimension d of t's STV is an anomalous dimension. Because the response time anomalies propagate from callees to callers by nature, all the response times of root cause's upstream (who directly or indirectly call it) will have response time anomalies as well. Therefore, given t, there will be multiple anomalous dimensions in t's STV, including those of the root cause and its upstream dimension. Among these anomalous dimensions, the one with the longest call path is considered by our algorithm as the root cause because it naturally reflects the response time propagation pattern. As shown in Fig. 7, Three anomalous dimensions are detected (dimensions 1, 2, 6 in Fig. 7), and the dimension with the longest call path is the right root cause, see the dimension 6. If no HSTV is found, the trace t may have invocation path anomaly (*e.g.*, call interrupt). Then *TraceAnomaly* finds the longest common path between trace t's longest call path and all call paths in the *call path list* (see §II-C). The longest common path and the next called microservice of the longest common path are the possible root cause.

III. Anomaly Detection Algorithm Design

This section presents the design of training and detection, and explains how Challenge 2 mentioned in §I-B is addressed.

A. Learning Data Distribution

Variational Auto-Encoders (VAE) [34] is a latent probabilistic model [34]. By introducing an auxiliary latent variable $\mathbf{z}$ with prior $p_\theta(\mathbf{z})$, VAE fits the data distribution $p(\mathbf{x})$ by $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}|\mathbf{z})\, p_\theta(\mathbf{z})\, \mathrm{d}\mathbf{z}$, where θ is the parameters of the model. $p_\theta(\mathbf{z})$ is usually a unit Gaussian $\mathcal{N}(\mathbf{0}, \mathbf{I})$. $p_\theta(\mathbf{x}|\mathbf{z})$ is

Dimension ID	(s, (call path)) of valid dimension	A HSTV value	STV value	Anomalous dimension ?	Root cause ?
1	(a, (start→a))	222	1302	yes	no
2	(b, (start→a, a→b))	209	1138	yes	no
3	(c, (start→a, a→b, b→c))	4	9	no	no
4	(d, (start→a, a→b, b→c, b→d))	44	36	no	no
5	(e, (start→a, a→b, b→c, b→d, d→e))	30	32	no	no
6	(e, (start→a, a→b, b→c, b→d, d→e, b→e))	67	980	yes	yes

Fig. 7: A example of root cause localizing from a real anomalous trace. The root cause is a response time anomaly at microservice e when microservice b calls microservice e.

a diagonal Gaussian $\mathcal{N}(\boldsymbol{\mu}_\theta(\mathbf{z}), \boldsymbol{\sigma}^2_\theta(\mathbf{z})\,\mathbf{I})$ in our paper, where $\boldsymbol{\mu}_\theta(\mathbf{z})$ and $\boldsymbol{\sigma}_\theta(\mathbf{z})$ are neural networks to be learned.

A VAE is usually learned by using the *variational approximation* technique. A separated variational posterior distribution $q_\phi(\mathbf{z}|\mathbf{x})$ is introduced to approximate the computationally intractable true posterior $p_\theta(\mathbf{z}|\mathbf{x})$. Then VAE is trained by maximizing the ELBO $\mathcal{L}(\phi, \theta)$, a lower-bound of $\log p_\theta(\mathbf{x})$:

$$\begin{aligned}\mathcal{L}(\phi, \theta; \mathbf{x}) &= \log p_\theta(\mathbf{x}) - \mathrm{D}_{KL}\left(q_\phi(\mathbf{z}|\mathbf{x}) \,\|\, p_\theta(\mathbf{z}|\mathbf{x})\right) \\ &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log p_\theta(\mathbf{x}|\mathbf{z}) + \log p_\theta(\mathbf{z}) - \log q_\phi(\mathbf{z}|\mathbf{x})\right] \quad (1)\end{aligned}$$

A popular method to compute $p_\theta(\mathbf{x})$ is *Monte Carlo integration* with *importance sampling* [35]. L_z samples are drawn from $q_\phi(\mathbf{z}|\mathbf{x})$, denoted as $z_{(l)}$, then $p_\theta(\mathbf{x})$ is computed as:

$$p_\theta(\mathbf{x}) \approx \frac{1}{L_z} \sum_{l=1}^{L_z} \frac{p_\theta(\mathbf{x}|\mathbf{z}_{(l)})\, p_\theta(\mathbf{z}_{(l)})}{q_\phi(\mathbf{z}_{(l)}|\mathbf{x})}$$

It is crucial for $q_\phi(\mathbf{z}|\mathbf{x})$ to approximate $p_\theta(\mathbf{z}|\mathbf{x})$ well. The closer $q_\phi(\mathbf{z}|\mathbf{x})$ is to $p_\theta(\mathbf{z}|\mathbf{x})$, the tighter $\mathcal{L}(\phi, \theta)$ is as a lower-bound (thus serves better to train $\log p_\theta(\mathbf{x})$), and the better $p_\theta(\mathbf{x})$ is computed [35]. $q_\phi(\mathbf{z}|\mathbf{x})$ is chosen to be a diagonal Gaussian $\mathcal{N}(\boldsymbol{\mu}_\phi(\mathbf{x}), \boldsymbol{\sigma}^2_\phi(\mathbf{x})\,\mathbf{I})$ in a vanilla VAE which is proven to be insufficient [31], [36]. One approach to better approximate $p_\theta(\mathbf{z}|\mathbf{x})$, is to use a flow-transformed posterior $q_\phi(\mathbf{z}|\mathbf{x})$, instead of using a diagonal Gaussian. By applying an *invertible mapping* $\mathbf{z}' = f_\phi(\mathbf{z})$ on the diagonal Gaussian $q'_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_\phi(\mathbf{x}), \boldsymbol{\sigma}^2_\phi(\mathbf{x})\,\mathbf{I})$, a more flexible posterior $q_\phi(\mathbf{z}'|\mathbf{x}) = q_\phi(\mathbf{z}|\mathbf{x})\,|\det(\partial f(\mathbf{z})/\partial \mathbf{z})|^{-1}$ can be obtained, thus have the ability to approximate $p_\theta(\mathbf{z}|\mathbf{x})$ better [31].

In this paper, we choose to use a modified version of Glow's flow [37] along with VAE.

B. Model Architecture

The design of our model is shown in Fig. 8. $\mathbf{x}$ denotes the service trace vectors, while $\mathbf{z}$ and $\mathbf{z}^{(k)}$ are latent variables. The service trace vectors $\mathbf{x}$ are fed into the *variational net*, passing through hidden layers $h_\phi(\mathbf{x})$, to obtain hidden features. These features are then used to derive the mean $\boldsymbol{\mu}_{\mathbf{z}^{(0)}}$ and the standard deviation $\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}$ of $\mathbf{z}^{(0)}$: $\boldsymbol{\mu}_{\mathbf{z}^{(0)}} = \mathbf{W}_{\boldsymbol{\mu}_{\mathbf{z}^{(0)}}} h_\phi(\mathbf{x}) + \mathbf{b}_{\boldsymbol{\mu}_{\mathbf{z}^{(0)}}}$, and $\boldsymbol{\sigma}_{\mathbf{z}^{(0)}} = \mathrm{SoftPlus}\left(\mathbf{W}_{\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}} h_\phi(\mathbf{x}) + \mathbf{b}_{\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}}\right) + \boldsymbol{\epsilon}$, where $\mathrm{SoftPlus}(\mathbf{a}) = \log(\mathbf{1} + \exp(\mathbf{a}))$, applied on each element of $\mathbf{a}$. $\mathbf{W}_{\boldsymbol{\mu}_{\mathbf{z}^{(0)}}}$, $\mathbf{b}_{\boldsymbol{\mu}_{\mathbf{z}^{(0)}}}$, $\mathbf{W}_{\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}}$ and $\mathbf{b}_{\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}}$ are network parameters to be learned. $\boldsymbol{\epsilon}$ is a small constant vector, chosen as the minimum value for $\boldsymbol{\sigma}_{\mathbf{z}^{(0)}}$, which can help avoid numerical issues in training (see §III-C), as is adopted in [28]. $\mathbf{z}^{(0)}$ is sampled from $\mathcal{N}\left(\boldsymbol{\mu}_{\mathbf{z}^{(0)}}, \boldsymbol{\sigma}^2_{\mathbf{z}^{(0)}}\mathbf{I}\right)$, then passed through the posterior flow of length K, to obtain $\mathbf{z}^{(K)}$.

The output $\mathbf{z}^{(K)}$ of the posterior flow is used as the latent variable $\mathbf{z}$, and fed into the *generative net*, passing through the

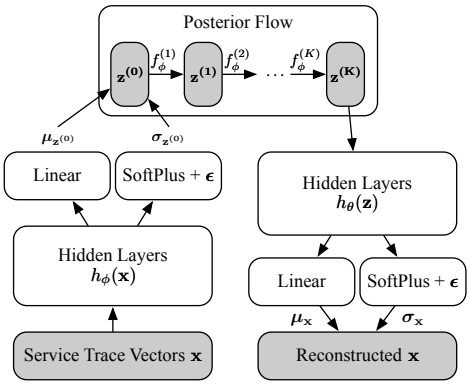


Fig. 8: The architecture of our model.

hidden layers $h_\theta(\mathbf{z})$, to again obtain hidden features. These features are then used to derive the mean $\boldsymbol{\mu}_{\mathbf{x}}$ and the standard deviation $\boldsymbol{\sigma}_{\mathbf{x}}$ of $\mathbf{x}$, just like for $\mathbf{z}^{(0)}$. The reconstructed $\mathbf{x}$ are then sampled from $\mathcal{N}\left(\boldsymbol{\mu}_{\mathbf{x}}, {\boldsymbol{\sigma}_{\mathbf{x}}}^2\mathbf{I}\right)$.

The re-parameterization trick [34] is applied on the initial $\mathbf{z}^{(0)}$. Since the Glow's flow [37] are continuous and deterministic mappings, the whole model can be viewed as having only one re-parameterized latent variable:

$$\mathbf{z}^{(K)} = (f^{(K)} \circ \cdots \circ f^{(1)})(\mathbf{z}^{(0)})$$

$$q_\phi^{(K)}(\mathbf{z}^{(K)}|\mathbf{x}) = q_\phi^{(0)}(\mathbf{z}^{(0)}|\mathbf{x}) \prod_{k=1}^{K} \left| \det \left(\frac{\partial f_\phi^{(k)}\left(\mathbf{z}^{(k-1)}\right)}{\partial \mathbf{z}^{(k-1)}} \right) \right|^{-1}$$

This largely reduces our effort to derive our training objective in§III-C, and our detection output in §III-D.

C. Training

It is straightforward to train our model with SGVB [34], by maximizing the ELBO $\mathcal{L}(\phi, \theta; \mathbf{x})$ over the training data:

$$\mathcal{L}(\phi, \theta; \mathbf{x}) = \mathbb{E}_{q_\phi^{(K)}(\mathbf{z}^{(K)}|\mathbf{x})} \Big[\log p_\theta(\mathbf{x}|\mathbf{z}^{(K)}) + \log p_\theta(\mathbf{z}^{(K)}) - \log q_\phi^{(K)}(\mathbf{z}^{(K)}|\mathbf{x}) \Big] \tag{2}$$

In actual deployment, the training data are obtained from the large number of traces data. Our anomaly detection must be unsupervised, but the training data may contain a few anomalous traces. However, our observation and assumption is that there are a few anomalies in the traces. We train our model by stochastic gradient descent (SGD), which can naturally tolerate rare anomalies. It updates the model parameters according to mini-batch samples of the input data, thus only captures the most significant patterns of the data distribution.

D. Anomaly Detection

To detect whether or not a service trace vector is anomalous, we use its log-likelihood against the model, *i.e.*, $\log p_\theta(\mathbf{x})$. As mentioned in §III-A, it can be computed by importance sampling as Eq. (3):

$$\log p_\theta(\mathbf{x}) \approx \log \frac{1}{L_z} \sum_{l=1}^{L_z} \left[\frac{p_\theta(\mathbf{x}|\mathbf{z}_{(l)}^{(K)})\, p_\theta(\mathbf{z}_{(l)}^{(K)})}{q_\phi^{(K)}(\mathbf{z}_{(l)}^{(K)}|\mathbf{x})} \right] \tag{3}$$

The criterion of judging a trace to be anomalous is that the pattern of the trace is significantly different from the normal traces' patterns. The criterion can be quantified as the differences of $\log p_\theta(\mathbf{x})$. Thus a trace is anomalous if the value of the trace's $\log p_\theta(\mathbf{x})$ is significantly smaller than the values of normal traces' $\log p_\theta(\mathbf{x})$. Instead of manually setting a threshold on $\log p_\theta(\mathbf{x})$ to split the normal traces and anomalous traces, we use Kernel Density Estimation (KDE) [38] to learn the distribution of normal traces' $\log p_\theta(\mathbf{x})$. If a trace's $\log p_\theta(\mathbf{x})$ does not follow the learned distribution with high probability, then the trace is an anomalous trace. We use p-value [39], which is widely used in statistical hypothesis testing, to judge whether the trace's $\log p_\theta(\mathbf{x})$ follows the trained KDE model. We set the significance level of p-value as 0.001, which is a commonly used value.

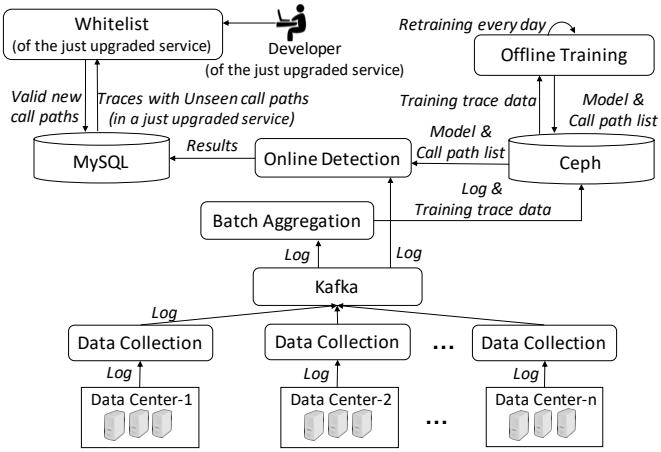


Fig. 9: Data pipeline in deployment at company *S*

IV. Implementation and Deployment

A. Deployment at online services

TraceAnomaly has been deployed on 18 real online services for about two months in company *S*. Fig. 9 shows the data pipeline in the deployment. 1) *Data Collection* modules collect RPC log data from different Data Centers, then push these collected log data into Kafka [40], which is a widely used open-source distributed streaming platform for building real-time data pipelines and stream processing applications. 2) *Batch Aggregation* module obtains log data from Kafka, then periodically aggregates them to extract the trace data. 3) The raw log data and the trace data are then stored into Ceph [41] database to be used by the *Offline Training* module (no labeling is needed for training as *TraceAnomaly* is unsupervised). To adapt to potential service upgrade, the model will be retrained every day at 00:00 based on the traces in the last one day. The average training time per day is 1.5 hours. We have three Tesla P40 GPUs in company *S*, thus models for at most three services (each for one service) can be trained in parallel. After a new model is trained, the trained model and the call path list (see §II-C) are stored in Ceph, then the *Online Detection* module is updated. 4) *Online Detection* directly gets log data from Kafka, performs real-time trace aggregation and then conducts detection using the latest model for a service to detect anomalies in the service, which are stored in MySQL database. *Offline Training* module and *Online Detection* module can plug in different approaches for parallel detection to facilitate our evaluation of baseline and alternative approaches in §V-B and §VI-A. According to our measurement, the average detection overhead of one trace is only 0.004 second. For a service that has about 800,000

TABLE I: Details of the four large evaluation services from company *S*.

	No. of Microservices	Evaluation duration	Average No. of traces/day	No. of STV Dimensions	No. of call graph structures	No. of manually confirmed anomalous traces	Description (all for mobile users)
Service-1	344	5 days (Sun. - Thu.)	801,021	690	368	108	transaction query.
Service-2	61	4 days (Sun. - Wed.)	600,806	173	61	68	account opening.
Service-3	233	4 days (Wed. - Sat.)	502,408	508	302	81	repayment.
Service-4	113	4 days (Wed. - Sat.)	500,921	412	186	66	account balance query.

traces per day, the total detection overhead per day is only 3200 seconds, which is acceptable to company *S*.

B. Dealing with Unseen Call Paths

We now present how to deal with false alarms caused by the unseen call paths after a service upgrade. If a service upgrade introduces previously unseen call paths, these call paths will accumulate sufficiently during the day and will be automatically treated as the "new normals" by the daily retrained model next day. What remains to be solved is how to deal with the false alarms before the daily retraining. We tackle this problem using a simple whitelist approach, shown in the top left of Fig. 9. The developers of the upgraded service are the ones who initiated the upgrade, thus they are asked to manually validate the previously unseen call paths detected by *TraceAnomaly*. Those validated new call paths are stored into a whitelist so that no more alarms are raised for them before the next daily retraining. Note that the manual validation overhead in practice is not much for the developers since 1) they wrote the code that generates the new call paths, and 2) the number of unseen call paths in our practice is very small. In fact, during the two-month deployment, we observe that the average total number of unseen call paths per day is ~ 4 per service. Therefore, the validation overhead is acceptable.

V. Evaluation

In this section, we will introduce the evaluation on four large online services (shown in Table I) from company *S* and a testbed. 1) §V-A introduces the methodology of our evaluation. 2) §V-B describes the baselines. 3) The evaluation results are presented in §V-C. 4) §V-D analyzes the improvement over a hard-coded rule approach. 5) §V-E evaluates the root cause localization algorithm. 6) §V-F summarizes *TraceAnomaly*'s impact on company *S*'s troubleshooting.

A. Evaluation Methodology

For *evaluation* purpose, ideally we should label each trace as either "anomalous" or "normal" to obtain the ground truth. However, it is infeasible to manually label online services trace data (each with hundreds of thousands of traces per day). We thus choose to construct a testbed to generate accurately labeled traces for evaluation firstly, then some better-performing approaches are selected for the large-scale online evaluation.

1) Testbed Evaluation: We choose a benchmarking microservice system, TrainTicket [32], as our testbed. TrainTicket is a train ticket booking system based on microservice architecture which contains 41 microservices. Each microservice is deployed on a Docker [43] container.

The traces for evaluation are generated by simulating different user requests randomly visiting different functions of TrainTicket. We simulate one day's traffic for training data with 40 peak periods and about 380,000 normal traces. Fig. 10 shows number of visits per 1 minute in the simulated traffic.

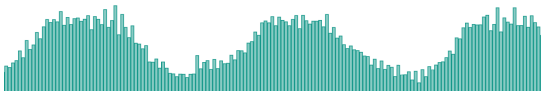


Fig. 10: Number of visits per 1 minute in the simulated traffic.

Note that, although these training traces are collected without any intentional anomaly injection, a small number of traces are still anomalous due to occasional system glitches.

To obtain the testing set, we inject anomalies to each of the 41 microservices with simulated traffic similar to the training set one by one. In reality, anomalies are more likely to occur during peak traffic periods, thus the anomalies are injected during the simulated peak periods to obtain the testing set. During a peak period, firstly the system runs normally, resulting normal test traces. Then one microservice is selected to first inject *response time anomaly* to it, by controlling the network card latency of the Docker container for 5 minutes. Then after less than 30 seconds, an *invocation path anomaly* is injected to it, by removing a microservice node from TrainTicket system for 5 minutes. Then the system is rebooted and let system run for 10 minutes to settle down in order to restore the system to normal. Then, another microservice is chosen to inject anomalies into. Overall, the above anomaly injection process runs for about 24 hours, and we collected 30,356 normal traces, 2,699 response time anomaly traces and 2380 invocation path anomaly traces as the testing set.

2) Obtaining Labeling of Online Traces: During online evaluation, it is infeasible to manually label all 18 services in which *TraceAnomaly* is deployed (each with hundreds of thousands of traces per day) for 2 months. Therefore, we obtained the ground truth as follows.

As described in §IV-A, *Offline Training* module and *Online Detection* module in Fig. 9 allow us to plug in all baselines in parallel. The union of all detected anomalous traces by all these baselines during the evaluation period was considered as the *candidate* anomaly trace set, and manually validated by two experienced operators separately. Finally, all the anomalous traces confirmed by both operators are labeled as *anomalous*, and labeled as *normal* otherwise. For the evaluation of various approaches using such an ground truth set, the precision is accurate but the recall might be biased towards 1 for all approaches, since some anomalous traces might be missed by all above approaches thus do not make it to the candidate anomaly ground truth set.

B. Anomaly Detection Baseline Approaches

We first introduce eight baseline approaches that conduct end-to-end trace anomaly detection. The details of Multimodal LSTM [3], AEVB [4], WFG-based [5], CFG-based [6] and CPD-based [7] have been introduced in related work (§I-A). The remaining baseline approaches are:

Hard-coded Rule. Company *S* adopts an approach based on the microservices' return codes. This approach was the

TABLE II: Evaluation results of different approaches on a TrainTicket testbed which contains 41 microservices. The test set contains 30,356 normal test traces, 2,699 response time anomaly traces and 2380 invocation path anomaly traces.

	Overall		Response Time Anomaly		Invocation Path Anomaly		Training (minutes) for 24-hour traces	Test (seconds) for 4-hour traces
	Precision	Recall	Precision	Recall	Precision	Recall		
WFG-based [5]	0.76	0.92	0.65	0.96	0.96	0.87	0.06	0.4
DeepLog* [8]	0.52	0.71	0.65	0.96	0.34	0.42	306	78
CPD-based [7]	0.30	0.47	N/A	N/A	0.30	1.0	N/A	9
CFG-based [6]	0.70	0.49	0.70	0.94	N/A	N/A	N/A	0.1
AEVB [4]	0.17	0.52	0.17	0.98	N/A	N/A	6120	121
OmniAnomaly [42]	0.45	0.49	0.45	0.93	N/A	N/A	530	113
Multimodal LSTM [3]	0.60	0.96	N/A	0.94	N/A	0.97	9.5	109
TraceAnomaly	**0.98**	**0.97**	N/A	**0.94**	N/A	**0.99**	94	19

TABLE III: Online evaluation results of different approaches on four large online services which contain hundreds of microservices, whose statistics are shown in Table I.

	Service-1		Service-2		Service-3		Service-4		Overall (Union of 4 services)	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Hard-coded Rule	0.910	0.800	0.920	0.792	0.911	0.812	0.930	0.800	0.910	0.804
WFG-based [5]	0.020	0.500	0.012	0.323	0.050	0.410	0.032	0.300	0.031	0.386
DeepLog* [8]	0.270	0.680	0.241	0.560	0.320	0.643	0.302	0.601	0.290	0.628
CPD-based [7]	0.52	0.063	0.43	0.090	0.57	0.110	0.64	0.072	0.531	0.081
CFG-based [6]	0.170	0.610	0.250	0.570	0.102	0.503	0.180	0.630	0.164	0.562
TraceAnomaly	**0.980**	**1.000**	**0.982**	**1.000**	**0.981**	**1.000**	**0.973**	**1.000**	**0.981**	**1.000**

practice of trace anomaly detection at company *S* prior to *TraceAnomaly*'s deployment. For each call it serves, a microservice is required to return a status code to indicate its status during the call (*e.g.*, whether the response time is above a fixed threshold $2000ms$, or some error has occurred). The return codes of all the microservices involved in a trace constitute its code set. The developers of a service manually design some *fixed* rules based on the code set to conduct anomaly detection of each trace. We call such a detection approach as *Hard-coded Rule*.

OmniAnomaly [42]. It proposed a stochastic recurrent neural network for multivariate time series anomaly detection. The response time data of traces' microservices can form a multivariate time series, thus multivariate time series anomaly detection can be used to detect response time anomalies.

DeepLog* [8]. DeepLog [8] uses a Long Short-Term Memory (LSTM) model to detect *execution path* anomaly. However, DeepLog trains an LSTM model for each *microservice* to detect *response time* anomaly **while *TraceAnomaly* trains a model for each service** . If we use LSTM to detect response time anomalies of microservices, then hundreds of LSTM models have to be trained in our scenario (*e.g.*, Service-1 has 344 microservices in Table I). Due to the resource limitation of online services, we replace the response time anomaly detection with the 3-sigma method (same as WFG-based's), and denote the modified approach DeepLog*.

C. Results

Table II shows the evaluation results of various approaches on a testbed which contains 41 microservices, and Table III shows the evaluation results of various approaches on four large online services which contain hundreds of microservices. The detection performance of different approaches can be measured by precisions and recalls. Because the ground truth of anomalous traces from testbed is known, we also calculate the precisions and recalls of the response time anomaly and invocation path anomaly. CPD-based, CFG-based, AEVB and OmniAnomaly only detect invocation path anomalies or response time anomalies, so some precisions and recalls cannot be calculated in Table II. *TraceAnomaly* and Multimodal LSTM cannot distinguish the anomaly type (response time anomaly or invocation path anomaly), so the precisions of response time anomaly and invocation path anomaly cannot be calculated. In reality, the anomalous traces usually contain both response time anomalies and invocation path anomalies. It is distinguished in testbed only for evaluation purpose.

The response times of a normal microservice can change significantly with different invocation paths. As shown in Fig. 4, the microservice e is invoked twice, with different response times. Although WFG-based, DeepLog*, CFG-based, AEVB and OmniAnomaly also detect microservice-level response time anomalies, they do not consider the different invocation paths of a microservice, which causes the bad performance. CPD-based only detects invocation path anomalies. However, only analyzing invocation path is not suitable to detect trace anomalies. A real example in Fig. 6 shows that an anomalous trace and a normal trace can have the same invocation paths. **These baselines' performance demonstrates the value of a unified detection approach**. Furthermore, due to the various structures of traces, the time series data of microservices' response time contain many missing data points. These missing data points make the performance of AEVB and OmniAnomaly worse. Although Multimodal LSTM tries a unified approach, it is hard to learn the relationships between response times and invocation paths in an end-to-end way. Due to the bad performance and relative long training time (6120 minutes and 530 minutes), AEVB, OmniAnomaly and Multimodal LSTM are not selected in online evaluation. It takes *TraceAnomaly* about 90 minutes to train for one-day traces for both testbed and online, and this training overhead is acceptable in practice.

The hard-coded rule is inferior to *TraceAnomaly*, especially in the recall metric (an improvement of ~ 0.2). Since in real applications like ours, the number of anomalous traces is far less than normal traces', it would require much more efforts to manually fix the false negatives than the false positives.

TABLE IV: The evaluation results of root cause localization approaches on a TrainTicket testbed.

	Precision@1	Precision@2	Precision@3
MonitorRank [44]	0.35	0.47	0.53
RCSF [45]	0.29	0.41	0.47
MEPFL [46]	0.44	0.62	0.67
TraceAnomaly	**0.99**	N/A	N/A

TABLE V: The evaluation results of *TraceAnomaly* for localizing root causes on four large services.

	#anomalous traces	TraceAnomaly root cause localization	
		internal microservice (successful)	external service (successful)
Service-1	18	18(18)	0
Service-2	8	6(6)	2(2)
Service-3	27	24(24)	3(3)
Service-4	20	18(18)	2(2)

Thus, the recall metric is much more important than precision. According to this, we conclude that the *TraceAnomaly* works far better than hard-coded rules.

As mentioned in §V-A, the precision using the approximate ground truth is accurate, but the recall might be biased towards 1 for all approaches in Table III. Note that although the recall metrics of *TraceAnomaly* are 1.0 on all four services, there might actually still be some anomalies not successfully detected. Nevertheless, Table III is still a good evidence of the superiority of *TraceAnomaly*, in that it can detect all the anomalies reported by all other algorithms, with higher precision than all baselines.

D. Improvement Over Hard-coded Rule

As described in §V-B, Hard-coded Rule is the most widely used method in company *S* production systems. We thus provide some detailed comparison between it and *TraceAnomaly*. Some false positives of hard-coded rule approach were due to its not-well-designed rules. For example, a developer designed a rule r_1 on Service-1 (see Table I) to detect anomalous traces: *the trace is anomalous if any of its microservices returns a "fault" return code*. During our evaluation, we found that some traces detected by r_1 were not detected by *TraceAnomaly*. After analyzing the detailed source codes with developers, we found that r_1 mistakenly declares the following normal case as anomalous. A microservice of Service-1 regularly calls an external API which has a rate limit, exceeding of which results in a fault return code of the API but the corresponding trace should not be considered anomalous, according to the application developers. *TraceAnomaly* successfully determines such traces as normal since similar traces appears not uncommonly, yet hard-coded rule approach simply considers the trace with a fault return code as anomalous.

Hard-coded Rule cannot detect some response time anomalies. For example, a developer set a (conservative) timeout threshold of microservice g to 2 seconds. The normal response times of g are smaller than 0.3 seconds. *TraceAnomaly* detected some anomalous traces, and we found that the microservice g's response times of these detected traces are greater than 1 second and less than 1.5 seconds. When we presented these traces by *TraceAnomaly*, the operators thought that these traces are anomalous and need to be detected.

E. Localizing Root Cause

We firstly compare the root cause localization performance of *TraceAnomaly* with three baseline approaches on the testbed, these baseline approaches are:

MonitorRank [44] generates personalized pagerank vector by adjacency matrix and time series correlation among nodes, then uses pagerank algorithm to localize root causes.

RCSF [45] aggregates error paths from alarming frontend nodes to error backend nodes, and mines frequent sequential patterns from collecting paths to localize root cause.

MEPFL [46]. Based on a set of features defined on the traces, MEPFL trains prediction models in supervised way to localize the faulty microservices.

Table IV shows the results. Precision@K (Precision at top K) is a commonly used metric for ranking, and it indicates the probability that top K results given by an approach contain the root cause. The localization results of MonitorRank [44], RCSF [45] and MEPFL [46] are ranking lists, so Precision@K are used to measure their performance. Due to the localization result of TraceAnomaly are not ranking list, we only compute Precision@1 for TraceAnomaly.

MonitorRank [44] localizes the root cause by analyzing the time series response time data of traces' microservices. Due to the various trace structures, microservices' response time data contain many irregular fluctuations and missing data points, which makes the algorithms' performance worse. RCSF [45] only mines the frequent sequence of trace paths to localize the root cause, which is not enough to handle the various structures of traces and response time anomalies. The defined features of MEPFL [46] cannot reflect the invocation path anomaly, which leads to the bad performance. Therefore, we only evaluate *TraceAnomaly* localization in online evaluation.

For online evaluation, we collected all 73 recorded anomalous traces from four large services which have been analyzed by operators. As shown in Table V, the root causes of these traces includes internal microservice and external service, and *TraceAnomaly* successfully localizes the root causes of all 73 anomalous traces. In the future, we plan to analyze the exact reasons of the root causes by analyzing more information, includes the logs, configurations, source codes, *etc.*.

F. Impacts

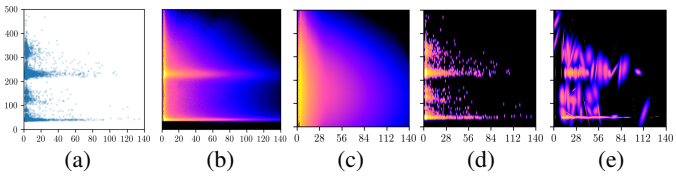
Thanks to the good performance of *TraceAnomaly*, it has been used as an important alerting building block for monitoring and troubleshooting systems at company *S*:

Early fault warning. Some faults will show evidences in the traces much earlier before the service is significantly hurt. For example, a microservice's response time becomes very large due to a deteriorating fault, then some traces will be anomalous immediately. However, it may take minutes to hours before the deteriorating fault could influence the $average$ response time of the service, because the $average$ response time changes very slowly due to the large number of traces. For another example, a small number of trace anomalies might reveal some hidden bugs, discovering of which can help developers to fix the bug *before* it eventually triggers severe damage.

Real-time fault detection. If a service has a fault, a batch of similar anomalous traces may appear in a short time (*e.g.* 1 minute). Then the monitoring system will immediately alarm, and operators can quickly mitigate and troubleshoot the faults.

TABLE VI: The evaluation results of the explored approaches on four large services and a TrainTicket testbed.

	Service-1		Service-2		Service-3		Service-4		TrainTicket	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
TraceAnomaly with GMM	0.700	0.540	0.670	0.500	0.713	0.551	0.690	0.564	0.890	0.850
TraceAnomaly with KDE	0.660	0.392	0.593	0.300	0.600	0.410	0.530	0.332	0.820	0.75
TraceAnomaly with Vanilla VAE	0.910	0.800	0.900	0.810	0.890	0.792	0.913	0.800	0.930	0.910
TraceAnomaly	**0.980**	**1.000**	**0.982**	**1.000**	**0.981**	**1.000**	**0.973**	**1.000**	**0.980**	**0.970**



(a) (b) (c) (d) (e)

Fig. 11: Density/heat maps of: (a) the 2-d dataset; (b) $\log p_\theta(\mathbf{x})$ of *TraceAnomaly*; (c) $\log p_\theta(\mathbf{x})$ of VAE; (d) log density of KDE; (e) log density of GMM.

VI. Analysis

Recall that our core idea is to learn the distribution $p(\mathbf{x})$ of the service trace vectors (STVs) by an unsupervised learning algorithm, and apply the learned model to detect anomalies. Some unsupervised algorithms can potentially serve as drop-in replacements of the Bayesian networks proposed in this paper: Gaussian Mixture Model (GMM) [47], Kernel Density Estimation (KDE) [38] and Vanilla VAE [34]. In this section, we replace the Bayesian networks with these alternatives in *TraceAnomaly*, and then compare their performances and analyze why *TraceAnomaly* works better than the alternatives.

A. Explored Alternative Approaches

Table VI shows the evaluation results of the explored approaches. Firstly, among our explored approaches, deep models consistently outperform non-deep models (*i.e.*, KDE, GMM), which strongly supports our intuition that deep learning techniques are necessary to learn $p(\mathbf{x})$ from complicated data. The second observation is that, TraceAnomaly consistently outperforms Vanilla VAE, as what we have expected in §III-A. This highlights the need of a strong (*i.e.*, having large modeling capacity) posterior $q_\phi(\mathbf{z}|\mathbf{x})$ when applying VAE on trace anomaly detection. We thus conclude that among all learned models, TraceAnomaly is the best one to choose.

B. Internal Mechanisms Analysis

To analyze the internal mechanisms of the algorithms in §VI-A, we plot the density of a 2-d dataset (Fig. 11a) and the heat maps of the models trained on this dataset (Figs 11b to 11e). The data come from an online service for one day, whose number of STV dimensions is two. We choose this dataset because higher dimensional datasets are hard to analyze through visualization. The density of the dataset (Fig. 11a) is plotted by directly drawing each STV as a 20% transparent point on the figure. The heat maps of trained models are plotted as follows. First, we pick 500 values along each of the x-axis and y-axis, with equal intervals, and obtain 250,000 points from the Cartesian product of x- and y-values. These points are the input for the models. We then compute the score ($\log p_\theta(\mathbf{x})$ or log density, respectively) of each $\mathbf{x}$ corresponding to each model. Finally, we plot these scores as the heat maps in Fig. 11.

There is a sharp contrast between the deep models (Figs 11b and 11c) and non-deep models (Figs 11d and 11e). The deep models manage to learn a smooth $p(\mathbf{x})$, where the non-deep models are only able to obtain a non-smooth $p(\mathbf{x})$, leaving "holes" on areas not covered by training samples. Since the values of STVs are response times, it is reasonable to believe these values are continuous by nature, thus having a smooth $p(\mathbf{x})$ should benefit the detection performance (which is indeed the situation in Table VI). As the dimension grows, non-deep models will struggle even harder to learn a smooth, transient, or realistic $p(\mathbf{x})$. This fact strongly supports our preference to use deep models in §III-A. As for the deep models (Figs 11b and 11c), we can see that the vanilla VAE fails to capture the fine-grained patterns of data. This is because the lack of a good posterior makes the ELBO (Eqn (1)) fail to serve as a tight lower-bound of $\log p_\theta(\mathbf{x})$ in training, resulting in a sub-optimal model, as suggested in §III-A. We thus conclude that *TraceAnomaly* can effectively learn $p(\mathbf{x})$ of STVs.

VII. Conclusion

This paper presents *TraceAnomaly*, an unsupervised anomaly detection approach that can automatically learn the overall normal patterns of the traces for a service, and detect anomalies via computing the likelihood based on the learned normal pattern. *TraceAnomaly* has a novel service trace vector encoding and deep variational Bayesian networks with posterior flow. *TraceAnomaly* has been deployed on 18 large online services in a company *S* which serves tens of millions of users. Detailed evaluations on four large online services (with 61 to 344 microservices) and a testbed show that the recall and precision of *TraceAnomaly* are both above 0.97, outperforming the existing approach in *S* (hard-coded rule) by 19.6% and 7.1%, and seven other baselines by 57.0% and 41.6% on average. Furthermore, *TraceAnomaly* localized the correct root causes on all 73 anomalous traces from four large services, and its localization accuracy outperforms three baseline approaches by 55% in testbed results. Thanks to the good performance of *TraceAnomaly* during two-month deployment, more monitoring and troubleshooting systems are planned to be develop based on *TraceAnomaly* in *S*.

Acknowledgment

The authors acknowledge the anonymous reviewers for their valuable feedbacks. The authors thank Wenxiao Chen and Juexing Liao for their helpful suggestions and proofreading. This work has been supported by the National Key R&D Program of China (2019YFB1802504), the National Natural Science Foundation of China (61902200), the China Postdoctoral Science Foundation (2019M651015), the Key-Area Research and Development Program of Guangdong Province (2019B010136001), the Science and Technology Planning Project of Guangdong Province LZC0023, and the Beijing National Research Center for Information Science and Technology (BNRist) key projects.

REFERENCES

[1] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[2] B. H. Sigelman, L. A. Barroso *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," Technical report, Google, Inc, Tech. Rep., 2010.

[3] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *CLOUD'19*. IEEE, 2019.

[4] S. Nedelkoski, J. Cardoso *et al.*, "Anomaly detection and classification using distributed tracing and deep learning," 2018.

[5] Q. Fu, J.-G. Lou *et al.*, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 149–158.

[6] A. Nandi, A. Mandal *et al.*, "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, 2016, pp. 215–224.

[7] L. Bao, Q. Li, P. Lu, J. Lu, T. Ruan, and K. Zhang, "Execution anomaly detection in large-scale systems through console log analysis," *Journal of Systems and Software*, vol. 143, pp. 172–186, 2018.

[8] M. Du, F. Li *et al.*, "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, 2017, pp. 1285–1298.

[9] A. H. Yaacob, I. K. Tan *et al.*, "Arima based network anomaly detection," in *Communication Software and Networks, 2010. ICCSN'10. Second International Conference on*. IEEE, 2010, pp. 205–209.

[10] H. Yan, A. Flavel *et al.*, "Argus: End-to-end service anomaly detection and localization from an ISP's point of view," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 2756–2760.

[11] Y. Chen, R. Mahajan, B. Sridharan, and Z.-L. Zhang, "A provider-side view of web search response time," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, 2013.

[12] S.-B. Lee, D. Pei *et al.*, "Threshold compression for 3g scalable monitoring," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1350–1358.

[13] B. Krishnamurthy, S. Sen *et al.*, "Sketch-based change detection: methods, evaluation, and applications," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 2003, pp. 234–247.

[14] B. Pincombe, "Anomaly detection in time series of graphs using arma processes," *Asor Bulletin*, vol. 24, no. 4, p. 2, 2005.

[15] F. Knorn and D. J. Leith, "Adaptive kalman filtering for anomaly detection in software appliances," in *INFOCOM Workshops 2008, IEEE*. IEEE, 2008, pp. 1–6.

[16] A. Mahimkar, Z. Ge *et al.*, "Rapid detection of maintenance induced changes in service performance," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 13.

[17] W. Lu and A. A. Ghorbani, "Network anomaly detection based on wavelet analysis," *EURASIP Journal on Advances in Signal Processing*, vol. 2009, p. 4, 2009.

[18] A. A. Mahimkar, H. H. Song *et al.*, "Detecting the performance impact of upgrades in large operational networks," in *SIGCOMM 2010*.

[19] Y. Himura, K. Fukuda, K. Cho, and H. Esaki, "An automatic and dynamic parameter tuning of a statistics-based anomaly detection algorithm," in *Communications, 2009. ICC '09. IEEE International Conference on*.

[20] A. B. Ashfaq *et al.*, "An information-theoretic combining method for multi-classifier anomaly detection systems," in *Communications (ICC), 2010 IEEE international conference on*.

[21] S. Shanbhag and T. Wolf, "Accurate anomaly detection through parallelism," *Network, IEEE*, vol. 23, no. 1, pp. 22–28, 2009.

[22] A. Mahimkar, Z. Ge, J. Wang *et al.*, "Rapid detection of maintenance induced changes in service performance," in *CoNEXT 2011*.

[23] D. R. Choffnes, F. E. Bustamante, and Z. Ge, "Crowdsourcing service-level network event monitoring," in *SIGCOMM 2010*.

[24] S.-B. Lee, D. Pei, M. Hajiaghayi *et al.*, "Threshold compression for 3g scalable monitoring," in *INFOCOM 2012*.

[25] A. Soule *et al.*, "Combining filtering and statistical methods for anomaly detection," in *IMC 2005*.

[26] Y. Zhang, Z. Ge, A. Greenberg, and M. Roughan, "Network anomography," in *IMC 2005*.

[27] F. Silveira, C. Diot *et al.*, "Astute: Detecting a different class of traffic anomalies," in *SIGCOMM 2010*.

[28] H. Xu, W. Chen *et al.*, "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*. International World Wide Web Conferences Steering Committee, 2018, pp. 187–196.

[29] J. An and S. Cho, "Variational autoencoder based anomaly detection using reconstruction probability," SNU Data Mining Center, Tech. Rep., 2015.

[30] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," in *ACM SIGPLAN Notices*, vol. 51, no. 4. ACM, 2016, pp. 489–502.

[31] D. Rezende and S. Mohamed, "Variational Inference with Normalizing Flows," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1530–1538.

[32] X. Zhou, X. Peng *et al.*, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *TSE'18*, 2018.

[33] "The traceanomaly project." https://github.com/traceanomaly/anomalyDetection, June 7, 2020.

[34] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," in *Proceedings of the International Conference on Learning Representations*, 2014.

[35] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[36] D. P. Kingma, T. Salimans *et al.*, "Improved variational inference with inverse autoregressive flow," in *Advances in Neural Information Processing Systems*, pp. 4743–4751.

[37] D. P. Kingma and P. Dhariwal, "Glow: Generative flow with invertible 1x1 convolutions," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 10 236–10 245.

[38] V. A. Epanechnikov, "Non-parametric estimation of a multivariate probability density," *Theory of Probability & Its Applications*, vol. 14, no. 1, pp. 153–158, 1969.

[39] Y. P. Chaubey, "Resampling-based multiple testing: Examples and methods for p-value adjustment," 1993.

[40] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[41] S. A. Weil, S. A. Brandt *et al.*, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[42] Y. Su, Y. Zhao *et al.*, "Robust anomaly detection for multivariate time series through stochastic recurrent neural network," in *SIGKDD'19*, 2019.

[43] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, 2014.

[44] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, 2013.

[45] K. Wang, C. Fung *et al.*, "A methodology for root-cause analysis in component based systems," in *IWQoS'15*. IEEE, 2015.

[46] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," 2019.

[47] D. Reynolds, "Gaussian mixture models," *Encyclopedia of biometrics*, pp. 827–832, 2015.