# LogFlash: Real-time Streaming Anomaly Detection and Diagnosis from System Logs for Large-scale Software Systems

Tong Jia, Yifan Wu, Chuanjia Hou, Ying Li*
*Peking University*
Beijing, China
{ jia.tong, yifanwu, cjhou, li.ying}@pku.edu.cn

***Abstract*—Today, software systems are getting increasingly large and complex and a short failure time may cause huge loss. Therefore, it is important to detect and diagnose anomalies accurately and timely. System logs are a straightforward and important source of information for anomaly detection and diagnosis. However, existing log-based approaches have three key limitations. First, they are not designed for processing real-time log streams. Second, they require restrictions on training log data. Third, they lack the adaptiveness to system update. To break through these limitations, we propose LogFlash, a real-time streaming anomaly detection and diagnosis approach that enables both training and detection in a real-time streaming processing manner. By assigning a dynamic pairwise transition rate to each template pair and model the transition possibility as typical power-law distribution, our approach achieves real-time model construction and updates. Experiment results show that it reduces over 5 times of training and detection time compared with the state-of-art works while maintaining the capability of accurate anomaly diagnosis.**

***Keywords—Real-time, Streaming Computing, Anomaly Detection, Log Analysis***

## I. Introduction

Software systems are getting increasingly large and complex that often contain hundreds of components, and support a large number of concurrent users. Correspondingly, a short failure time may cause a huge loss. One particular challenge for large-scale software systems is anomaly detection and diagnosis. That is, how to quickly detect system anomalies at run-time and diagnose system problems. System logs are a straightforward and important source of information for anomaly detection and diagnosis. Typically, administrators manually check log files and search for problem-related log entries. However, in today's large-scale systems, logs can be overwhelmingly large. For instance, in some large-scale systems that provide global services, the amount of daily log data could reach tens of terabytes (TBs). A Microsoft online service system even generates over one petabyte (PB) of logs every day [1]. Therefore, manually diagnosing problems can be time-consuming and error-prone. Besides, problems of today's systems can be very complex in a cross-component and cross-service manner, it is hard to diagnose anomalies based on certain "error" logs.

To improve diagnosis efficiency and reduce human efforts, it is important yet challenging to perform automatic anomaly detection and diagnosis precisely and timely. Specifically, automatic anomaly detection and diagnosis encounter three key challenges. First, fast and efficient log processing and modeling are difficult. Massive unstructured and heterogeneous log texts from multiple components often require large computation efforts [2]. Second, precisely capturing various anomaly symptoms in logs is not easy. Today's software system suffers from multiple types of faults such as network failure, software bug, configuration error, etc. Different faults may expose various implicit and complex anomaly symptoms in logs instead of simple "exception" or "error" messages. Capturing these complex symptoms to perform precise anomaly detection is not easy. Third, the usage of asynchronous and non-blocking services poses difficulties to log-based anomaly diagnosis. For traditional multi-threaded applications, log entries generated by the same request could be identified by context information, e.g., thread id (TID) or process id (PID), supported by standard logging libraries. However, logs generated by non-blocking applications lack this context information as one thread or process serves more than one request by multiplexing. Therefore, logs generated by concurrent request servers are interleaved together which brings great complexity in constructing diagnosis models [3-5].

Facing the above challenges, many log-based anomaly detection and diagnosis approaches have been proposed. To efficiently process logs, researchers have proposed online real-time log template mining approaches [6-10], A log template is an abstraction of a print statement in source code, which manifests itself in logs with different embedded parameter values in different executions. Represented as a set of invariant keywords and parameters (denoted by parameter placeholder), a template can be used for summarization of multiple log entries. To precisely capture complex anomaly symptoms in logs, researchers either leverage machine/deep learning algorithms ([11-18][54-58]) or build graph-based models to trace fine-grained request execution paths with logs and perform sophisticated comparison analysis in the graph to detect anomalies ([3-5][19-28]). To overcome the log interleaving problem caused by asynchronous and non-blocking services, researchers either assume logs contain shared parameters such as request/transaction ID that can tie logs together in a request [19-25] or leverage probabilistic inference algorithms to capture interleaved log features in a fuzzy manner ([3-5][26]). Generally, the process of these approaches includes offline training and online detection. Offline training utilizes historical system logs as training log data for model construction while online detection is applied to online log stream for anomaly detection.

However, existing approaches have faced several key limitations due to the advance of modern large-scale software systems.

1. Lack of real-time streaming anomaly detection and diagnosis models. In today's large-scale software systems, an hour failure time may cause a huge loss of 100,000 dollars [29], thus real-time anomaly detection and diagnosis are a

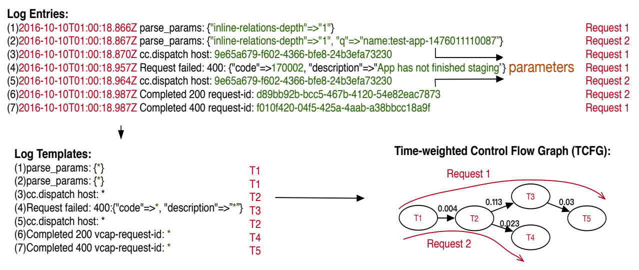
must. However, due to the complexity of log data, most existing approaches perform offline "posterior" anomaly detection and diagnosis, that is, after system failure occurs, these models recall historical logs to locate anomaly points. A few real-time anomaly detection approaches extract coarse-grained statistical features of logs such as template appearance frequency and utilize clustering algorithms to detect "outliers" as anomalies [30-34]. These approaches sacrifice the rich information and the diagnosis ability of logs, thus are not capable of anomaly diagnosis. Other online detection approaches ([11][18][54-56]) utilize deep learning models to predict the current log entry in the log stream. However, these approaches are not fast enough to support real-time anomaly detection because of the complex model structure, especially when the model combines lots of other knowledge [54-56].

2. Restrictions on training log data. Some existing works leverage classification algorithms and require labeled training log data [15-16]. However, labeling requires deep system understanding and extensive system management experience. Even for experienced software engineers, labeling such massive log data is still time-consuming and tedious work that is almost impossible to accomplish. Other works aim to construct a descriptive model with normal log data and compare the runtime log data with the model to locate deviations ([3-5][17-28][54-56]). These works require normal log data for training which is inaccessible in real-world systems. In practice, even the system functionality is normal, benign and untriggered faults may hide in system programs. These faults will not cause immediate system failure, but may display anomaly symptoms in system logs. In fact, for real-world large-scale software systems, it is almost impossible to ensure that log data is normal at a time period.

3. Lack of adaptiveness to system update. In most enterprises, developers need to move fast and implement changes quickly in response to changing business demands in DevOps mode [35]. Therefore, software systems usually experience frequent updates in which the evolution and maintenance of logging code is a crucial activity [36]. Most existing works [3-5,17-28] require an offline training phase and have to retrain the model as long as the logging code is updated. This makes existing approaches almost unavailable.

The only work that targets this problem is LogRobust[54]. It utilizes TF-IDF and word vectorization to transform logs into semantic vectors. In this way, updated logs can also be transformed into semantic vectors and participate in model training and deduction. However, the semantic vectors can only handle small changes of log updates such as single word changing. If developers add a new logging code, the model cannot recognize its semantic vector and must be retrained.

To break through these limitations, in this paper, we propose LogFlash, a real-time streaming anomaly detection and diagnosis approach that enables concurrent model training and anomaly diagnosis at real-time. LogFlash considers anomaly detection and diagnosis as a real-time streaming processing task where each log entry is processed only once without any iterations or intermediate storage. It utilizes a dynamic pairwise transition rate to model the transition likelihood of each template pair and then builds a graph-based model to trace fine-grained request execution paths according to the transition likelihood. In real-time, it trains and updates the transition rate matrix consistently based on network inference algorithms[38]. Since all updates are processed in real-time, it only requires the online log stream without



```
Log Entries:
(1)2016-10-10T01:00:18.866Z parse_params: {"inline-relations-depth"=>"1"}                                          Request 1
(2)2016-10-10T01:00:18.867Z parse_params: {"inline-relations-depth"=>"1", "q"=>"name:test-app-1476011110087"}      Request 2
(3)2016-10-10T01:00:18.870Z cc.dispatch host: 9e65a679-f602-4366-bfe8-24b3efa73230                                 Request 1
(4)2016-10-10T01:00:18.957Z Request failed: 400: {"code"=>170002, "description"=>"App has not finished staging"}   Request 1
(5)2016-10-10T01:00:18.964Z cc.dispatch host: 9e65a679-f602-4366-bfe8-24b3efa73230                                 Request 2
(6)2016-10-10T01:00:18.987Z Completed 200 request-id: d89bb92b-bcc5-467b-4120-54e82eac7873                         Request 2
(7)2016-10-10T01:00:18.987Z Completed 400 request-id: f010f420-04f5-425a-4aab-a38bbcc18a9f                         Request 1

Log Templates:
(1)parse_params: {*}                                  T1
(2)parse_params: {*}                                  T1
(3)cc.dispatch host: *                                T2
(4)Request failed: 400:{"code"=>*, "description"=>"*"} T3
(5)cc.dispatch host: *                                T2
(6)Completed 200 vcap-request-id: *                   T4
(7)Completed 400 vcap-request-id: *                   T5
```

**Figure 1: Log Templates and TCFG model.**

specific training datasets. After a system update, it will follow the update to change the transition rate matrix automatically based on the edge aging mechanism and the inference algorithm so as to adapt the change of logging code.

Experiment results show that LogFlash reduces over 5 times of training and detection time compared with the state-of-art works while maintaining the capability of accurate problem diagnosis. We further conduct extensive performance evaluation and results show that LogFlash converges fast to follow frequent system updates. The contributions of this paper are as follows:

1. We propose a new real-time streaming anomaly detection and diagnosis approach for large-scale software systems.

2. To the best of our knowledge, we are the first to achieve both model training and anomaly detection in real-time without any restrictions or assumptions on system log data.

3. We have conducted an extensive evaluation of the proposed approach. Results have shown the effectiveness and efficiency of our approach.

The rest paper is organized as follows. Section 2 elaborates the preliminaries of our work. Section 3 describes the details of LogFlash. Section 4 represents the experiments and corresponding results. We introduce the related work in Section 5 before concluding the paper in Section 6.

## II. Preliminaries

Before we describe the proposed approach, it is necessary to clear log templates and time-weighted control flow graph (TCFG) model as the basis of our work at first. A TCFG is a directed graph consisting of edges and nodes and each edge has a time weight recording the transition time. TCFG model stitches together various log templates and represents the baseline normal system state. It is used to flag deviations from expected behaviors at runtime. A template is an abstraction of a print statement in source code, which manifests itself in logs with different embedded parameter values in different executions. Represented as a set of invariant keywords and parameters (denoted by parameter placeholder *), a template can be used for the summarization of multiple log lines. The TCFG is such a graph where the nodes are templates and the edges represent the transition from one template to another. In addition, every log has a timestamp indicating its print time, thus the difference between two log timestamps represents the program execution time between the two logs. This difference is recorded as the time weight of each edge in the TCFG. Fig. 1 shows an example of log templates and the TCFG model. Each log has some invariant keywords and some variable parameters (shown in green) and log templates only reserve invariant keywords. Nodes in the TCFG are different log

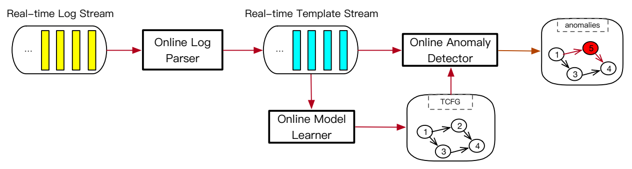


**Figure 2: Workflow of our approach.**

templates, edges represent how each request flow passes between nodes, and the weight of edges indicates the transition time between two nodes.

TCFG model enables fine-grained anomaly detection and diagnosis through recording detailed request flows with log templates. In this paper, we leverage TCFG as the anomaly detection and diagnosis model.

## III. The Proposed Approach

We propose an online self-updating anomaly detection and diagnosis approach called LogFlash. Fig. 2 shows the overall workflow of LogFlash.

The input data is an online log stream $\boldsymbol{l} := (l_1, l_2, l_3, \ldots)$, where $l_i$ is a log entry. Our approach consists of three main components, namely online log parser, online model learner and online anomaly detector. In online log parser, multiple log templates are mined from a log stream $\boldsymbol{l}$ and each log entry $l_i$ is replaced by its corresponding template $p_i$. In this way, the log stream $\boldsymbol{l}$ is transformed into a template stream $\boldsymbol{p} := (p_1, p_2, p_3, \ldots)$. This template stream then goes through the online model learner and online anomaly detector concurrently. Online model learner infers and updates the structure of TCFG through mining the template stream based on network inference algorithm [38]. Online anomaly detector utilizes the latest TCFG model to detect and diagnose anomalies in the template stream. Note that the computations of the online model learner and anomaly detector are fully asynchronous.

We leverage existing online template mining algorithm [6] in online log parser. Due to space limitations, we will focus on online model learner and online anomaly detector in the following sections.

### A. *Online Model Learner*

We aim to construct a TCFG model in a black-box manner only with the template stream $\boldsymbol{p}$. Our key idea is to define a dynamic pairwise transition rate $\boldsymbol{\alpha}_{j,i}$ which models how frequently a request flows from template $\boldsymbol{j}$ to template $\boldsymbol{i}$ and train/update the transition rate $\boldsymbol{\alpha}_{j,i}$ over time with template stream $\boldsymbol{p}$.

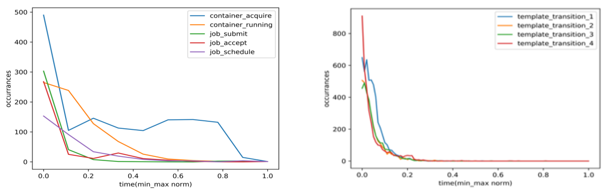
We further define $f(t_i|t_j, \alpha_{j,i})$ to be the conditional likelihood of transition between template $j$ and template $i$ where $t_j$ and $t_i$ are the timestamps of two occurrences of template $j$ and template $i$ in $p$. We assume the conditional likelihood depends on the transition time $(t_j, t_i)$ and the transition rate $\alpha_{j,i}$. To model this parametric likelihood, we first conduct a statistical analysis on the distribution of template transitions.

*1) Statistical Analysis:* We first collect system logs of 434 job executions from a Hadoop cluster in our lab and system logs of 5 minutes from an industrial software system called Ada in this paper. Then we mine log templates from these logs and record the transition time between every occurrence of two neighboring templates in the same request by calculating the difference of their timestamps. Next, we count the number of occurrences with the same transition time and plot the distribution of each template transition.

**Table 1. Log Templates and TCFG model.**

| Transmission Description | Predecessor Template | Successor Template |
|---|---|---|
| container acquire | Container Transitioned from ALLOCATED to ACQUIRED | Container Transitioned from ACQUIRED to RUNNING |
| container running | Container Transitioned from ACQUIRED to RUNNING | Container Transitioned from RUNNING to COMPLETED |
| job submit | State change from NEW to SUBMITTED | State change from SUBMITTED to SCHEDULED |
| job accept | State change from NEW_SAVING to SUBMITTED | Accepted application application_*_* from user: * |
| job schedule | State change from SUBMITTED to SCHEDULED | State change from SCHEDULED to ALLOCATED_SAVING |

* *Due to space limitation, we only display the key part of each log template.*



(a) Hadoop (b) Ada

**Figure 3: The distribution of template transitions.**

Fig. 3 plots the distributions of several random template transitions. To better display the distribution, we apply Min-Max Normalization [37] to the time axis. Table 1 shows the details of five transitions from Hadoop system logs including two container status change and three job status change. The distributions of these transitions show obvious power-law (long-tail) characteristics where most transitions cost less than 0.2 norm-value of time. Analysis results of Ada draw the same conclusion that template transitions are subject to the power-law distribution. We further utilize Kolmogorov-Smirnov test [51] to test the goodness of fit for these transition distributions and hypothesized power-law distributions. P-value [52] results are over 0.05 which means that these transition distributions have no significant differences with power-law distributions.

Based on the above observations and analysis, pow-law likelihood is appropriate to model $f(t_i|t_j, \alpha_{j,i})$, that is:

$$f(t_i|t_j, \alpha_{j,i}) = \begin{cases} \dfrac{\alpha_{j,i}}{\delta}\left(\dfrac{t_i - t_j}{\delta}\right)^{-1-\alpha_{j,i}} & if\ t_j + \delta < t_i \\ 0 & otherwise \end{cases} \quad (1)$$

where $\delta$ states the minimum transition time from template $j$ to template $i$. As $\alpha_{j,i} \to 0$, the expected transition time from template $j$ to template $i$ becomes arbitrarily long.

*2) Dynamic TCFG Structure Inference:* After modeling the conditional likelihood of transitions between templates as pow-law likelihood, we compute the template stream likelihood based on the conditional likelihood. Then we

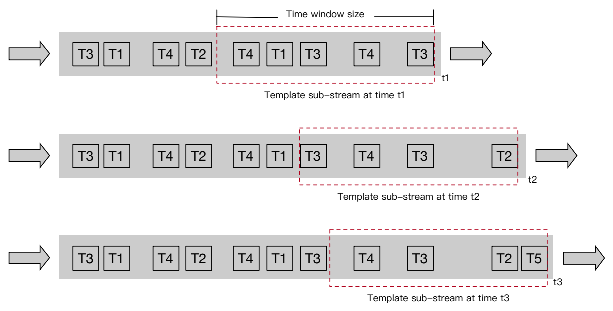

Time window size
T3 T1 T4 T2 T4 T1 T3 T4 T3
t1
Template sub-stream at time t1
T3 T1 T4 T2 T4 T1 T3 T4 T3 T2
t2
Template sub-stream at time t2
T3 T1 T4 T2 T4 T1 T3 T4 T3 T2 T5
t3
Template sub-stream at time t3

**Figure 4: An example of sub-streams at different time.**

**Table 2. Computations of transition likelihood for pow-law model.**

| Computation Entity | Computation Method |
|---|---|
| Log survival function: $logS(t_i\|t_j,\alpha_{j,i})$ | $-\alpha_{j,i}\log\left(\frac{t_i-t_j}{\delta}\right)$ |
| Hazard function: $H(t_i\|t_j,\alpha_{j,i})$ | $\alpha_{j,i}\cdot\frac{1}{t_i-t_j}$ |
| Gradient for $\alpha_{j,i}$ in $c$: $\nabla_{\alpha_{j,i}}L_c(\mathrm{A})$ | $\log\left(\frac{t_i^c-t_j^c}{\delta}\right)-\frac{(t_i^c-t_j^c)^{-1}}{\sum_{k:t_k^c<t_i^c}\alpha_{k,i}(t_i^c-t_k^c)^{-1}}$ |

reduce the TCFG Structure Inference problem as a maximum likelihood problem.

*a) Template Stream Likelihood.* In the template stream $\boldsymbol{p}$, transitions from different templates to a certain template are independent, that is, each occurrence of template $i$ can only be transmitted to once from the occurrence of one parent template. Therefore, given an occurrence of template $i$ at time $t_i$ in $\boldsymbol{p}$ and a collection of previously occurred templates $(t_1,\dots,t_N|t_k\le t_i)$, only one certain template $j$ at time $t_j$ transmits to $i$. To model the likelihood of transition $j\to i$, we first define a survival function $S(t_i|t_k,\alpha_{k,i})$:

$$S(t_i|t_k,\alpha_{k,i})=1-F(t_i|t_k,\alpha_{k,i}) \tag{2}$$

where $F(t_i|t_k,\alpha_{k,i})=\int_{t_j}^{t_i}f(t|t_k,\alpha_{k,i})dt$ is a cumulative transition density function. The survival function computes the survival likelihood that template $k$ has *not* transmitted to $i$ from time $t_j$ to $t_i$.

Then the likelihood of transition $\mathrm{j}\to\mathrm{i}$ in $\boldsymbol{p}$ results from multiplying the conditional likelihood of $\mathrm{j}\to\mathrm{i}$ and the survival likelihoods of other transitions $\mathrm{k}\to\mathrm{i}$ where $k\in\{1,\dots,N\}, k\ne j, t_k<t_i$ and $\mathrm{A}=\{\alpha_{j,i}|\mathrm{I},\mathrm{j}=1,\dots,\mathrm{N},\mathrm{i}\ne\mathrm{j}\}$:

$$f(t_i|t_j,\mathrm{A})=f(t_i|t_j,\alpha_{j,i})\times\prod_{k:k\ne j,t_k<t_i}S(t_i|t_k,\alpha_{k,i}) \tag{3}$$

The likelihood of occurrence of template $i$ at time $t_i$ given a collection of previous occurred templates $(t_1,\dots,t_N|t_k\le t_i)$ results from summing over the likelihood of the mutually disjoint transition from each previous occurred template to template $i$:

$$f(t_i|t_1,\dots,t_{N\setminus t_i},\mathrm{A})=\sum_{j:t_j<t_i}f(t_i|t_j,\mathrm{A}) \tag{4}$$

To simplify the modeling process, we assume that transitions are conditionally independent given a set of parent templates. The likelihood of all transitions in the template stream results

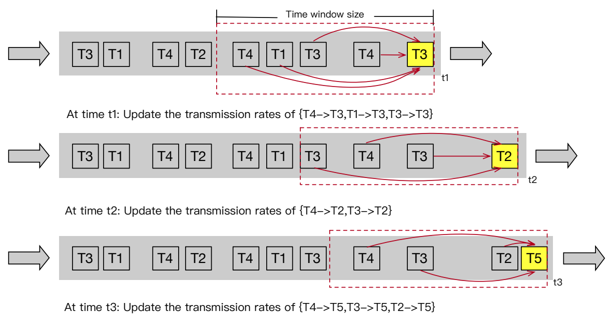

Time window size
T3 T1 T4 T2 T4 T1 T3 T4 T3
t1
At time t1: Update the transmission rates of {T4->T3,T1->T3,T3->T3}
T3 T1 T4 T2 T4 T1 T3 T4 T3 T2
t2
At time t2: Update the transmission rates of {T4->T2,T3->T2}
T3 T1 T4 T2 T4 T1 T3 T4 T3 T2 T5
t3
At time t3: Update the transmission rates of {T4->T5,T3->T5,T2->T5}

**Figure 5: Transition rate update process at different time.**

**Algorithm 1.** Sub-stream Revision Algorithm

```
Input: A template sub-stream c = $(\mathrm{p}_1,\mathrm{p}_2,\dots,\mathrm{p}_\mathrm{n})$
Output: Filtered template sub-stream c′
Definition: Function $checkParent(\mathrm{p}_m,S)$ returns true if $\mathrm{p}_m$ is the parent of one of the templates in set $S$, otherwise returns false.
1.  $m\leftarrow n-1,\ S\leftarrow\emptyset,\ c'\leftarrow c, \mathrm{I}\leftarrow\{\mathrm{p}_\mathrm{n}\}$
2.  while $m\ge 1$
3.    do if $checkParent(\mathrm{p}_m,S)=True$
4.         then $c'.remove(\mathrm{p}_m)$
5.         $S.add(\mathrm{p}_m)$
6.       elseif $checkParent(\mathrm{p}_m,I)=True$
7.         then $S.add(\mathrm{p}_m)$
8.    $m\leftarrow m-1$
9.    end
10. return c′
```

from multiplying the likelihood of occurrence of each template in $\boldsymbol{p}$:

$$f(\boldsymbol{t}^{\le T},\mathrm{A})=\prod_{t_i\le T}f(t_i|t_1,\dots,t_{N\setminus t_i},\mathrm{A}) \tag{5}$$

where $\boldsymbol{t}^{\le T}$ denotes the time of template stream is up to T. After removing the condition $k\ne j$ makes the product independent of $j$:

$$f(\boldsymbol{t}^{\le T},\mathrm{A})=\prod_{t_i\le T}\left(\prod_{t_k<t_i}S(t_i|t_k,\alpha_{k,i})\times\sum_{j:t_j<t_i}\frac{f(t_i|t_j,\alpha_{j,i})}{S(t_i|t_j,\alpha_{j,i})}\right) \tag{6}$$

The fact that some templates are not shown in the observation window is also in formative. We therefore add multiplicative survival terms to equation 6 and rearrange it with hazard function [39] or instantaneous transition rate of transition $\mathrm{j}\to\mathrm{i}$ as $H(t_i|t_j,\alpha_{j,i})=f(t_i|t_j,\alpha_{j,i})/S(t_i|t_j,\alpha_{j,i})$. Then the likelihood of the template stream is reformulated as:

$$f(\boldsymbol{t},\mathrm{A})=\prod_{i:t_i\le T}\left(\begin{array}{c}\prod_{m:t_m>T}S(T|t_i,\alpha_{i,m})\times\\ \prod_{t_k<t_i}S(t_i|t_k,\alpha_{k,i})\times\sum_{j:t_j<t_i}H(t_i|t_j,\alpha_{j,i})\end{array}\right) \tag{7}$$

*b) TCFG Structure Inference Problem.* Our purpose is to infer a TCFG structure that is most possible to generate the template stream $\boldsymbol{p}$. Given a TCFG with constant edge transition rates $A$, the TCFG structure inference problem reduces to solving a maximum likelihood problem:

$$\begin{array}{ll}maximize_{\mathrm{A}} & logf(\boldsymbol{t},\mathrm{A})\\ subject\ to & \alpha_{j,i}\ge 0, i,j=1,\dots,N,i\ne j\end{array} \tag{8}$$

where $\mathrm{A}=\{\alpha_{j,i}|\mathrm{I},\mathrm{j}=1,\dots,\mathrm{N},\mathrm{i}\ne\mathrm{j}\}$ are the edge transitions we aim to train. The edges in TCFG are those pairs of templates with transition rates $\alpha_{j,i}\ge 0$.

To support online model update, we generalize the inference problem to dynamic TCFG structure with edge transition

**Algorithm 2.** Online TCFG Construction Algorithm

**Input**: Template stream $\boldsymbol{p}$, time window size $w$, decay rate $\beta$, update step size $\gamma$
**Output**: Transition rate matrix A, Time weight matrix $T$
**Definition**: Function $subStreamRevision$(c) is the Sub-stream Revision Algorithm.
Function $getSubStream(i, w)$ returns a sub-stream c in which the latest template is $i$ and the time window size of c is $w$.

```
1.  $k \leftarrow 0$
2.  for all $i$ in $\boldsymbol{p}$ do
3.      $c \leftarrow getSubStream(i, w)$
4.      $c' \leftarrow subStreamRevision$(c)
5.      for all $(j, i): j \neq i, t_j < t_i$ in $c'$ do
6.          $\alpha_{j,i}^{k} \leftarrow \left(\alpha_{j,i}^{k-1} - \gamma \nabla_{\alpha_{j,i}} L_c(\mathrm{A}^{k-1})\right)^{+}$
7.          if $tr_{j,i} < tw_{j,i}^{k-1} - \tau$
8.              then $tw_{j,i}^{k} \leftarrow \beta \cdot tw_{j,i}^{k-1}$
9.              elseif $tr_{j,i} > tw_{j,i}^{k-1}$
10.                 then $tw_{j,i}^{k} \leftarrow tr_{j,i}$
11.     end
12.     for all $(j, i): \alpha_{j,i}^{k-1} > 0$ not in $c'$ do
13.         $\alpha_{j,i}^{k} \leftarrow \beta \alpha_{j,i}^{k-1}$
14.     end
15.     $k \leftarrow k - 1$
16. end
17. return A, T
```

rates $\mathrm{A}(t)$ that may change over time. To this aim, we first split the template stream $\boldsymbol{p}$ to a set of sub-streams $\boldsymbol{c} = (c_1, c_2, c_3, \dots)$ based on the arrival of new templates. Given a time window size $w$, each time a template $i$ arrives, we split out a sub-stream in which $i$ is the latest template. An example is shown in Fig. 4. At time $t_1$, log stream in the red block is the current sub-stream. At time $t_2$, a new template $T_2$ is observed and the current sub-stream becomes $\{T_3, T_4, T_3, T_2\}$. When it comes to time $t_3$ when $T_5$ is observed, the current sub-stream becomes $\{T_4, T_3, T_2, T_5\}$. In this way, at any given time $t$, we solve a maximum likelihood problem over the set of sub-streams:

$$maximize_{\mathrm{A(t)}} \quad \sum_{c \in \boldsymbol{c}} f\left(\boldsymbol{t}^{c}, \mathrm{A(t)}\right)$$
$$subject\ to \qquad \alpha_{j,i}(t) \geq 0, i, j = 1, \dots, N, i \neq j \qquad (9)$$

where $c \in \boldsymbol{c}$. Next, we show how to efficiently solve the above optimization problem for all time points $t$.

*3) Real-time TCFG Construction:* As proved in [40], the problem defined by equation 8 is convex for the pow-law transition model. Therefore, we can aim to find optimal training solution at any given time $t$. Since in the condition of pow-law model, the edge transition rates usually vary smoothly. Therefore, classical stochastic gradient descent [41] can be a perfect method for our training as we can use the inferred TCFG structure from the previous time step as initialization for the inference procedure in the current time step. The training phase uses iterations of the form:

$$\alpha_{j,i}^{k}(t) = \left(\alpha_{j,i}^{k-1}(t) - \gamma \nabla_{\alpha_{j,i}} L_c\left(\mathrm{A}^{k-1}(t)\right)\right)^{+} \qquad (10)$$

where $k$ is the iteration number, $\nabla_{\alpha_{j,i}} L_c(\cdot)$ is the gradient of the log-likelihood $L_c(\cdot)$ of sub-stream $c$ with respect to the edge transition rate $\alpha_{j,i}$, $\gamma$ is the update step size, $(z)^{+} = \max(0, z)$. The computations of log survival function, hazard function and gradient of sub-stream $c$ for pow-law model (1) are given in Table 2.

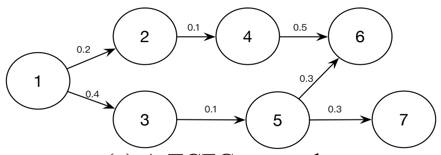
Importantly, in each iteration of the training phase, we only need to compute the gradients $\nabla_{\alpha_{j,i}} L_c(\mathrm{A}^k)$ for edges such that template $j \rightarrow i$ such that node $j$ has been observed in sub-stream $c$, and the iteration cost and convergence rate are independent of $|\boldsymbol{c}|$ [42][43].

*a) Conditional Independence Assumption.* In template stream likelihood deduction, we mentioned an assumption that transitions are conditionally independent given a set of parent templates. However, this assumption does not hold in many cases. Suppose a simple template sub-stream $c$ which contains three linear templates denoted as $(k, j, i)$. It is obvious that transition $j \rightarrow i$ and transition $k \rightarrow i$ are not conditional independent. In fact, as $k \rightarrow j \rightarrow i$ is the only transition pattern, the possibility of transition $k \rightarrow i$ is arbitrary zero given transition $j \rightarrow i$.

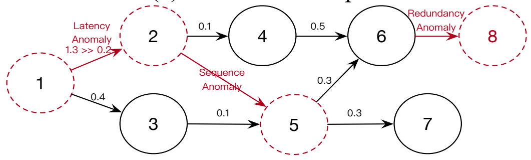
To meet this assumption, we design a sub-stream revision algorithm that enables filtering out indirect ancestor templates such as $k$. As shown in Algorithm 1, for a sub-stream $c = (p_1, p_2, \dots, p_n)$, we reversely traverse $c$ from $p_{n-1}$ to $p_1$. If a template $p_m$ is the indirect ancestor template of $p_n$, we remove $p_m$ (line 3,4). Then, if $p_m$ is the parent template of $p_n$, we reserve $p_m$ in $c$ (line 5,6). In this way, only possible parent template candidates of $p_n$ is reserved in each iteration. Transitions from these candidates to $p_n$ is usually conditional independent. To validate the sub-stream revision algorithm, we first collect system logs of 434 job executions from a Hadoop cluster in our lab. Then we manually compare the templates in a sub-stream before and after each iteration of the revision algorithm separately with expert knowledge. Results show that the revision algorithm can filter 82.3% indirect ancestor templates on average for all sub-streams

*b) Aging edges.* In each iteration $k$ of the training phase, we only update the edge transition rate $\alpha_{j,i}^{k}$ if template $j$ is observed in sub-stream $c$. Suppose a template transition rate $\alpha_{j,i}$ is over $\varepsilon$. Then during a system update, the logging statement of template $j$ is removed. After that, template $j$ never appears in any of the future sub-stream, and transition rate $\alpha_{j,i}$ will never be updated, thus the edge $j \rightarrow i$ will remain in TCFG forever. However, we would like such edges to decay and eventually vanish so as to adapt system updates. To achieve this, we define a decay rate $\beta$ and multiply transition rates of unobserved transition by $\beta$ in every training iteration.

*c) Time Weight Determination.* Each edge in TCFG has a time weight on it to record the transition time. The intuition behind this design is that if the transition time between two templates exceeds the time weight, then system or service may suffer from latency anomalies. Existing works [4][5] record the longest transition time in normal log data without anomalies. However, anomalies definitely exist in online log stream, thus the longest transition time may represent abnormal system status. To solve this problem, we use the decay rate $\beta$ mentioned above to reduce the time weight constantly, that is, we multiply the current time weight $tw_{j,i}^{k}$ by $\beta$ if the observed transition time $tr_{j,i}$ is shorter than $tw_{j,i}^{k}$, otherwise, we replace $tw_{j,i}^{k}$ with $tr_{j,i}$ in every training iteration. However, this simple decay strategy may bring extra false alarms as an appropriate time weight may be over-decayed. We further define a threshold $\tau$ to decide whether a decay operation should be applied in each iteration. Only if



(a) A TCFG example



(b) Different types of anomalies

**Figure 6: An example of different types of anomalies in TCFG.**

$tr_{j,i}$ is shorter than $tw^k_{j,i} - \tau$, current time weight is decayed. We set $\tau = 1$ second in our experiments.

*4) Real-time self-update TCFG Construction algorithm:* Now we can finally present our online TCFG construction algorithm. To further accelerate the construction process, we only update the transition rates from other templates to the latest template in the current sub-stream, because transition characteristics between other templates have been captured in prior sub-streams. Fig. 5 shows an example of the basic update process. At time $t_1$, we only update the transition rates from other templates to the latest template $T_3$. At time $t_2$ when a new template $T_2$ is observed, the current update iteration only focuses on the transitions from $T_3, T_4$ to $T_2$ in the sub-stream. When it comes to time $t_3$, it is obvious that transition rates from prior templates to $T_3$ and $T_2$ have been updated at time $t_1$ and $t_2$ respectively, thus only transition rates from other templates to $T_5$ are updated. Algorithm 2 describes the details of our algorithm. For each template i in **p**, we first split a sub-stream c with i to be the latest template and length to be the time window size w. Then we filter out indirect ancestor templates of i with sub-stream revision algorithm (Algorithm 1). Next, we update the transition rates $\alpha_{j,i}$ with stochastic gradient decent method (line 6). After that, we update the time weight on each edge $j \rightarrow i$ with decay rate $\beta$ and threshold $\tau$ (line 7-10). Note that if a same template occurs multiple times in c, we treat the transition time from the latest occurrence to i as $tr_{j,i}$. At last, for those edges $j \rightarrow i$ in which j is not in c, we apply decay operations to them.

When outputting the final TCFG structure, we simply omit edges with transition rates less than a threshold parameter $\varepsilon$. Note that $\varepsilon$ should be assigned to a smaller value so as to capture enough transition information from the template stream. We use $\varepsilon = 0.1$ in all experiments. Then we assign $tw^k_{j,i}$ in time weight matrix T to each edge as the time weight.

## B. Online Anomaly Detector

The basic idea for anomaly detection and diagnosis is to compare log stream with TCFG to find the deviation. We first define three types of deviations/anomalies including *sequence anomaly*, *redundancy anomaly*, and *latency anomaly*. A sequence anomaly is raised when the log that follows the occurrence of a parent node cannot be mapped to any of its children. A redundancy anomaly is raised when unexpected logs occur that cannot be mapped to any node in the temporal path of the TCFG. An unexpected log can be an obvious abnormal log that cannot be matched to any template or a redundant occurrence of a log template. A latency anomaly is raised when the child of a parent node is seen but the transition time exceeds the time weight recorded on the edge. When anomalies are found, we flag a sub-structure of TCFG as the anomaly flag based on the anomaly types for administrators to diagnose root causes. For sequence anomaly, we flag the minimal sub-tree starting with the parent node as well as the undesirable child node. For redundancy anomaly, we flag the unexpected node and its parent node due to the abnormal log stream. Fig. 6 shows an example of different types of anomalies. Fig. 6a is an example of TCFG with 7 nodes. As shown in Fig. 6b, suppose the transition time between node 1 and node 2 exceeds the time weight 0.2, then they suffer a latency anomaly. Node 5 appears after node 2 unexpectedly and suffers a sequence anomaly. Node 8 appears after node 6 while node 8 is a new template which has not been recorded in the TCFG, thus a redundancy anomaly occurs.

With the three types of anomalies, we reduce the problem into detecting these anomalies at real-time. For each template $\boldsymbol{i}$ in $\boldsymbol{p}$, we first split a sub-stream $\boldsymbol{c}$ with $\boldsymbol{i}$ to be the latest template and length to be the time window size $\boldsymbol{w}$. Then if $\boldsymbol{i}$ is not a node in TCFG, the algorithm returns a redundancy anomaly. Next, we traverse sub-stream $\boldsymbol{c}$ to detect if there exist occurrences of parent nodes $\boldsymbol{j}$ of $\boldsymbol{i}$ in TCFG, otherwise, the algorithm returns sequence anomaly. At last, we check if the transition time from $\boldsymbol{j}$ to $\boldsymbol{i}$ is shorter than the time weight $\boldsymbol{tw_{j,i}}$ recorded in the TCFG, otherwise, the algorithm returns latency anomaly.

## IV. Experiments and Evaluation

LogFlash is built on top of Apache Flink. We use shared memory to store the TCFG model and implement communication among components. Each component can read or write to shared memory directly, and through mutual exclusion, memory consistency can be guaranteed. In order to reduce the computation cost caused by frequent updates of the TCFG model, we optimize the algorithm to make it simple. Furthermore, online model learner updates the TCFG model every 100-time windows.

We summarize three research questions in evaluation:

RQ1: How effective is LogFlash in anomaly detection and diagnosis?

RQ2: How efficient is LogFlash?

RQ3: Can LogFlash be adaptive to system update?

### A. Experiment Setup

In the experiments, we use Hadoop [44], Spark [45], and Flink [46] as our lab systems. We run wordcount benchmark hundreds of times on each system, and inject faults into the system at runtime with state-of-art fault injection tool SSFI [47]. SSFI defines 12 types of faults such as value revision, object revision, etc. For each execution of wordcount jobs, we inject a fault into a called function at run time. For all wordcount jobs, all types of faults are injected comprehensively into every function in the execution path. We generated 1,056, 762 and 442 anomaly jobs of Hadoop, Spark and Flink respectively, and collect the system logs of each job as abnormal log sets, then we generate the same

**Table 3. Overall Results of Different models trained with Normal Training Set. P. denotes Precision and R. denotes Recall**

| Approaches | Hadoop |  | Spark |  | Flink |  |
|---|---|---|---|---|---|---|
|  | P. | R. | P. | R. | P. | R. |
| DeepLog[11] | **0.99** | 0.89 | 0.95 | 0.86 | 0.97 | **0.94** |
| LogAnomaly[18] | **0.99** | 0.88 | **0.96** | 0.84 | 0.96 | 0.89 |
| LogRobust[54] | 0.97 | 0.89 | 0.95 | 0.84 | 0.96 | **0.94** |
| LogSed[4] | 0.86 | 0.72 | 0.90 | 0.74 | 0.89 | 0.82 |
| LogFlash | 0.95 | **0.91** | 0.95 | **0.89** | **0.98** | 0.93 |

**Table 4. Overall Results of Different models trained with Noisy Training Set. P. denotes Precision and R. denotes Recall**

| Approaches | Hadoop |  | Spark |  | Flink |  |
|---|---|---|---|---|---|---|
|  | P. | R. | P. | R. | P. | R. |
| DeepLog[11] | 1 | 0.73 | 0.99 | 0.62 | 0.98 | 0.78 |
| LogAnomaly[18] | 1 | 0.72 | 0.98 | 0.60 | 0.99 | 0.78 |
| LogRobust[54] | 1 | 0.70 | 0.99 | 0.60 | 0.96 | 0.79 |
| LogSed[4] | 0.92 | 0.70 | 0.90 | 0.58 | 0.92 | 0.77 |
| LogFlash | 0.94 | **0.88** | 0.95 | **0.82** | 0.98 | **0.93** |

amount of normal jobs and collect their logs for the training dataset. The testing dataset consists of all these logs above.

We test the execution time and convergence time of our approach on logs of different sizes and systems. The testing lab environment includes four 1.4 GHz Intel Core i5 CPUs and 16 GB memory. We present the evaluation results in the following sections.

## *B. RQ1: How Effective is LogFlash?*

In this section, we evaluate the effectiveness of our proposed approach, LogFlash. We choose state-of-art log-based anomaly detection and diagnosis approach DeepLog [11], LogAnomaly [18], LogRobust [54] and LogSed [4] as baselines. The first three approaches leverage LSTM [48] to model template sequences and detect anomalies through computing the distance between observed templates and predicted templates. We call them LSTM-based models in the rest of the paper. LogSed first proposes the TCFG model and infers the TCFG model based on the idea of frequent sequence mining. We evaluate the metrics under different settings and present the best results in the following. Towards this end, the hidden size, number of candidates, and batch size are set to be 256, 19, and 1024 respectively in LSTM-based models. The successor group time period is set to be 1 second and the filtering threshold is set to be 0.5 in LogSed.

Two abilities are the most important in real-world online anomaly detection and diagnosis. The first ability is to correctly detect as many anomalies as possible. The second ability is to output as few false alarms as possible. To evaluate the two abilities, we use *Recall* and *Precision* as our evaluation metrics which are defined as follows:

$$Precision = \frac{TP}{TP + FP} \tag{11}$$

$$Recall = \frac{TP}{TP + FN} \tag{12}$$

where $TP, FP, TN, FN$ are referred to as true positive, false positive, true negative and false negative. *Precision* represents the ability of avoiding false alarms. *Recall* represents the ability of detecting anomalies.

*1) Overall Results:* We run 3,000 wordcount jobs without any fault on each system as normal log sets. And we first use normal log sets to train each model. The results are shown in Table 3. LogFlash shows similar precision and recall with LSTM-based models and much better result than LogSed.

We demonstrate that in real industrial systems, training dataset may contain anomalies. Therefore, we add 5% abnormal logs into the training set to generate a noisy training set. Then we use the noisy training set to train each model and the same testing dataset to test. Evaluation results are shown in Table 4. LogFlash shows over 10% higher recall than other models because LogFlash is able to reduce the affections of abnormal logs with the edge aging mechanism. However,

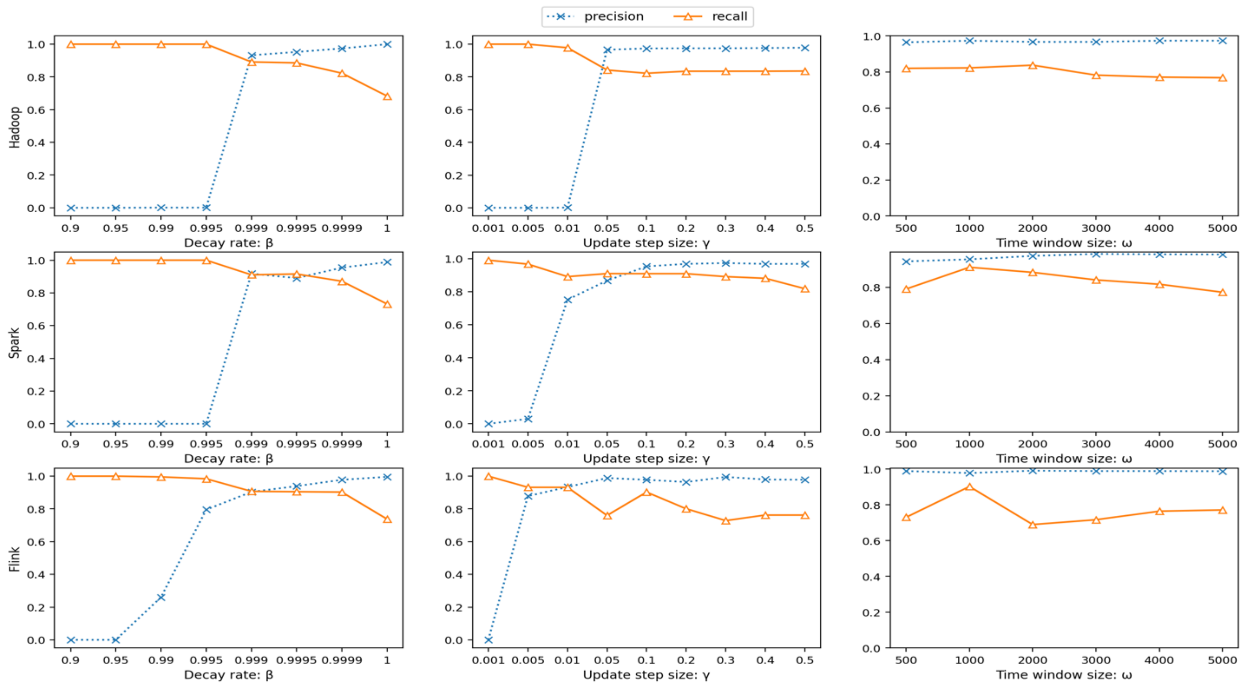
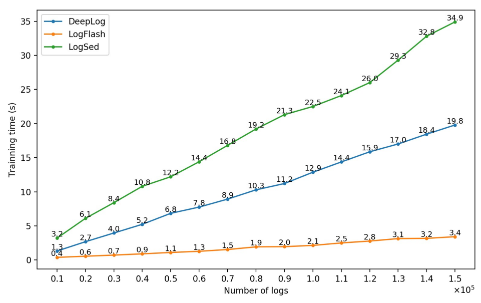


**Figure 7: Evaluation results of parameter sensitivity analysis.**


(a) Training time

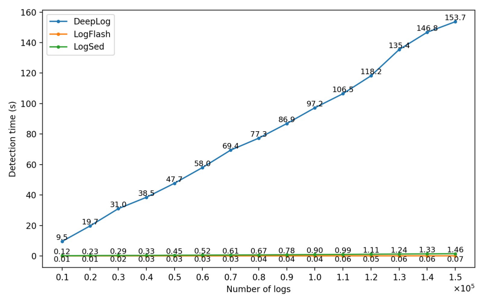
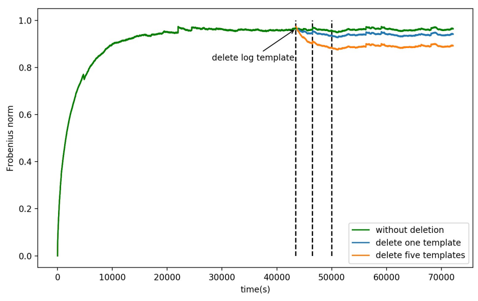

(b) Detection time

**Figure 8: Evaluation results for training and detection time.**


(a) Delete random templates

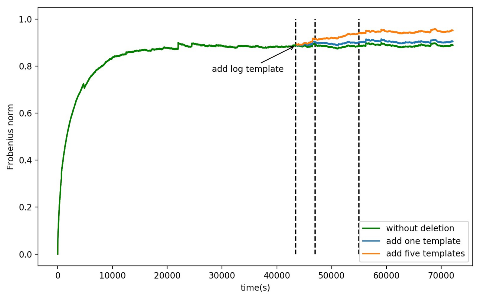

(b) Add random templates

**Figure 9: Evaluation of update convergence time.**

baseline approaches will learn the feature of abnormal logs from the noisy training dataset, so they can not recognize partial abnormal logs in the testing dataset. Therefore, the recall is much lower. High precision is because they only output very abnormal logs as anomalies such as "exception xx". Results show LogFlash is much more effective for real-world large-scale systems than state-of-art models.

*2) Parameter Sensitivity Analysis:* We further test the affections of different parameter settings. There are three important parameters in our approach: time window size w, decay rate $\beta$, update step size $\gamma$. We take $w = 1000$ milliseconds, $\beta = 0.9999$ and $\gamma = 0.1$ as the base configuration based on our experience and adjust the value of each parameter. Results are shown in Fig. 7.

We adjust decay rate $\beta$ from 0.9 to 1.0. Results show that $\beta = 0.995$ is a clear dividing line. When $\beta$ is smaller than $0.995$, transition rates decay too fast that TCFG cannot capture any transition relations between templates leading to the result of recall $= 1$ and precision $= 0$. When $\beta$ is larger than $0.995$, TCFG shows satisfied results. Note that $\beta = 1$ denotes that we remove the edge aging and time weight decay mechanism, and it shows an obvious low recall rate.

We adjust step size $\gamma$ from 0.001 to 0.5. Larger $\gamma$ prefers to generate less edges with shorter transition time in the TCFG while smaller $\gamma$ allows more edges from other templates to a certain template. Results show that when $\gamma \geq 0.05$, the model performs stably. When $\gamma \leq 0.005$, the model becomes unavailable with recall $= 1$ and precision $= 0$. This is because when $\gamma$ is too small, decay speed is faster than the update of transition rates, thus TCFG cannot capture any transitions. In the experiment of Flink, we observe that when $\gamma = 0.1$, the model performs much better than other parameter values. We compare the diagnosis results and find that a few extra sequence anomalies are detected when $\gamma = 0.1$. This shows that $\gamma = 0.1$ is a superior parameter value. If $\gamma > 0.1$, incorrect transitions may be captured to hide sequence anomalies. On the contrary, if $\gamma < 0.1$, too many transitions with relatively low transition rates will also hide the sequence anomalies.

We adjust time window size w from 500ms to 5000ms and find that w hasn't shown much affection to recall. As for precision, it improves when w is reduced from 5000ms to 1000ms, however, when 500ms is assigned to w, recall drops. Time window size w decides the coverage of transition rate updates. Larger w enables edges in the TCFG to record longer transitions. Therefore, some sequence anomalies may be hidden by longer transitions, thus precision improves as w reduces. However, if w is too small, some correct longer transitions may not be captured in the TCFG leading to the drop of recall rate.

### C. RQ2: How Efficient is LogFlash?

We compare the training time and anomaly detection time of LogFlash and baseline models with different sizes of log data respectively. DeepLog is the simplest model in LSTM-based models, so it is the fastest. Therefore, we choose DeepLog as a representation of LSTM-based models. To get the shortest training time of DeepLog, we set the number of training epochs to be 1, that is, each log entry is used only once for training. Results are shown in Fig. 8. It is noted that LogFlash is over 5 times faster than the state-of-art approaches in the training process. LogSed and DeepLog take about 3 seconds and 1.5 seconds on average to process 1000 log entries, while LogFlash takes about 0.3 seconds per 1000 log entries. In the detection process, LogFlash consumes a negligible time of 0.07 seconds for 150,000 log entries. However, DeepLog takes about 10 seconds to process 1000 log entries. Therefore, LogFlash is much more efficient than state-of-art approaches.

### D. RQ3: Can LogFlash be adaptive to system update?

We simulate system updates by changing the code in Hadoop source code. Specifically, we first run the Hadoop system for 43,000 seconds, then we delete/add one or five random logging statements (templates) in Hadoop source code.

To visualize the real-time changing of LogFlash model, we choose the Frobenius norm [49] of transition rate of matrix

A to show the changing process. Frobenius norm denotes the overall average value of all elements in a matrix which can strongly reflect the change of matrix. It is defined as follows:

$$\|A\|_F = \sqrt{\sum_{j=1}^{m} \sum_{i=1}^{n} {\alpha_{j,i}}^2} \tag{13}$$

Results are shown in Fig. 9. It shows that the Frobenius norm experiences a smooth rise and fall without fluctuation. After deleting/adding one template, it takes about 2,000 seconds to converge. If we delete five templates, it takes about 6,000 seconds to converge. If we add five templates, it takes about 10,000 seconds to converge. Note that in our experiment, we ensure that at any time only one job is processing in the system. In real-world large-scale cloud systems, thousands of requests or jobs are submitted concurrently. LogFlash can easily reach convergence in tens of seconds to adapt system updates.

We also compare LogFlash with LogRobust on the ability of adaptation of system update. Results show that LogRobust reported all newly added logs as anomalies. LogFlash also reported them as anomalies at first. When the transition rate matrix begins to converge again, LogFlash can accurately report real anomalies again without any other operations.

## V. Related Work

### A. Anomaly Detection and Diagnosis via Log Analysis

Analyzing logs for problem detection and identification has been an active research area ([1-5] [11-28][50]). These work first parse logs into log templates based on static code analysis or clustering mechanism, and then build anomaly detection and diagnosis models. These models include template frequency-based model, graph-based model, and deep learning-based model. Template frequency-based models [30-33] usually count the number of different templates in a time window, and set up a vector for each time window. Then it utilizes methods such as machine learning algorithms to distinguish outliers. These models sacrifice the abundant information and the diagnosis ability of logs and are not accurate and efficient, thus they cannot provide help for problem identification and diagnosis. Graph-based model ([3-5] [19-28]) is the current research hotspot. It extracts template sequence at first, and then generating a graph-based model to compare with log sequences in production environment to detect conflicts. This model has three advantages: First, it can diagnose problems that are deeply buried in log sequences such as performance degradation. Second, it can provide engineers with the context log messages of problems. Third, it can provide engineers with the correct log sequence and tell engineers what should have happened. Deep learning-based models ([11][18][54-56]) leverage LSTM to model the sequence of templates. With enough training log data, it can present superior results. However, this model takes a long time for training and inference, thus cannot support online anomaly detection and diagnosis.

### B. Mining Graph-based Model from System Logs

Some existing works ([19][28][50]) assume there exist some unique identifiers such as task ID or request ID. They use these IDs to correlate different templates and generate the graph structure. Some works ([17][23][24]) do not require a certain transaction ID, instead, they adopt multiple IDs such as UUID, thread ID, 32-char ID, etc. to tie templates together. Other works ([3][25]) leverage classical process mining or frequency mining approaches to infer the graph structure in a fuzzy manner. These works are designed for posterior anomaly diagnosis and consume heavy computation overhead. Besides, they share an assumption that system behavior is stable, thus log templates do not change.

## VI. Conclusion and Future Work

In this paper, we propose a novel online self-updating anomaly detection and diagnosis approach LogFlash for large-scale software systems. LogFlash constructs and updates a TCFG model in real-time based on the distribution of template transitions. To the best of our knowledge, we are the first to achieve both model training and anomaly detection as real-time streaming processing without any restrictions or assumptions on system log data. Experiment results have shown its effectiveness and efficiency.

In the future, we will apply human knowledge on LogFlash to handle the complexity of real-world system logs. Besides, we will combine other real-time data such as performance metrics and system tracing data to improve the diagnosis results.

## Acknowledgment

This work is supported by Key-Area Research and Development Program of Guangdong Province, China (NO.2020B010164003).

## References

[1] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE Companion)*, 2016, p. 102–111.

[2] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, p. 55–61, Feb. 2012.

[3] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "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 (KDD)*, 2016, p. 215–224.

[4] T. Jia, L. Yang, P. Chen, Y. Li, F. Meng, and J. Xu, "Logsed: Anomaly diagnosis through mining time-weighted control flow graph in logs," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 447–455.

[5] T. Jia, P. Chen, L. Yang, Y. Li, F. Meng, and J. Xu, "An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 25–32.

[6] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 33–40.

[7] K. Q. Zhu, K. Fisher, and D. Walker, "Incremental learning of system log formats," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, p. 85–90, Mar. 2010.

[8] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM)*, 2016, p. 1573–1582.

[9] L. Tang, T. Li, and C.-S. Perng, "Logsig: Generating system events from raw textual logs," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM)*, 2011, p. 785–794.

[10] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, 2016, pp. 859–864.

[11] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, p. 1285–1298.

[12] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings*

*of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009, p. 117–132.

[13] T.-F. Yen, A. Oprea, K. Onarlioglu, T. Leetham, W. Robertson, A. Juels, and E. Kirda, "Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks," in *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, 2013, p. 199–208.

[14] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2002, pp. 595–604.

[15] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: A discriminative pattern mining approach," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2009, p. 557–566.

[16] T. Reidemeister, M. A. Munawar, and P. A. Ward, "Identifying symptoms of recurrent faults in log files of distributed information systems," in *2010 IEEE Network Operations and Management Symposium (NOMS)*, 2010, pp. 187–194.

[17] J. Xu, P. Chen, L. Yang, F. Meng, and P. Wang, "Logdc: Problem diagnosis for declartively-deployed cloud applications with log," in *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*, 2017, pp. 282–287.

[18] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI)*, 2019, pp. 4739–4745.

[19] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *2009 Ninth IEEE International Conference on Data Mining (ICDM)*, 2009, pp. 149– 158.

[20] A. Babenko, L. Mariani, and F. Pastore, "Ava: Automated interpretation of dynamically detected anomalies," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, 2009, p. 237–248.

[21] Aharon M， Barash G， Cohen I， et al. "One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs," in *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*, 2009, p. 227-243

[22] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "lprof: A non-intrusive request flow profiler for distributed systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 629–644.

[23] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," *SIGPLAN Not.*, vol. 51, no. 4, p. 489–502, Mar. 2016.

[24] B. C. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan, "Logan: Problem diagnosis in the cloud using log-based reference models," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 62– 67.

[25] W. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 16, no. 9, pp. 1128–1142, 2004.

[26] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2010, p. 613–622.

[27] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, p. 143–154, Mar. 2010.

[28] Q. Fu, J.-G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie, "Contextual analysis of program logs for understanding system behaviors," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 397–400.

[29] S. Elliot, "Devops and the cost of downtime: Fortune 1000 best practice metrics quantified," *International Data Corporation (IDC)*, 2014.

[30] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection." in *USENIX Annual Technical Conference (ATC)*, 2010, pp. 1–14.

[31] A. J. Oliner and A. Aiken, "Online detection of multi-component interactions in production systems," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN)*, 2011, pp. 49– 60.

[32] C. Chen, N. Singh, and S. Yajnik, "Log analytics for dependable enterprise telephony," in *2012 Ninth European Dependable Computing Conference (EDCC)*, 2012, pp. 94–101.

[33] S. Du and J. Cao, "Behavioral anomaly detection approach based on log monitoring," in *2015 International Conference on Behavioral, Economic and Socio-cultural Computing (BESC)*, 2015, pp. 188–194.

[34] K. Kc and X. Gu, "Elt: Efficient log-based troubleshooting system for cloud computing infrastructures," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems (SRDS)*, 2011, pp. 11–20.

[35] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016.

[36] B. Chen and Z. Jiang, "Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, 02 2017.

[37] C. Saranya and G. Manikandan, "A study on normalization techniques for privacy preserving data mining," *International Journal of Engineering and Technology (IJET)*, vol. 5, pp. 2701–2704, 06 2013.

[38] M. Gomez Rodriguez, J. Leskovec, and B. Scholkopf, "Structure and¨ dynamics of information pathways in online media," in *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining (WSDM)*, 2013, p. 23–32.

[39] J. F. Lawless, *Statistical models and methods for lifetime data*. John Wiley & Sons, 2011, vol. 362.

[40] M. Gomez-Rodriguez, D. Balduzzi, and B. Scholkopf, "Uncovering¨ the temporal dynamics of diffusion networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML)*, 2011, p. 561–568.

[41] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li, "Parallelized stochastic gradient descent," in *Proceedings of the 23rd International Conference on Neural Information Processing Systems (NIPS)*, 2010, p. 2595–2603.

[42] F. Bach and E. Moulines, "Non-asymptotic analysis of stochastic approximation algorithms for machine learning," in *Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS)*, 2011, p. 451–459.

[43] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro, "Robust stochastic approximation approach to stochastic programming," *SIAM Journal on Optimization*, vol. 19, no. 4, pp. 1574–1609, 2009.

[44] "Hadoop," https://hadoop.apache.org/.

[45] "Spark," https://spark.apache.org/.

[46] "Flink," https://flink.apache.org/.

[47] Y. Yang, Y. Wu, K. Pattabiraman, L. Wang, and Y. Li, "How far have we come in detecting anomalies in distributed systems? an empirical study with a statement-level fault injection method," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 59–69.

[48] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[49] A. Custodio, H. Rocha, and L. Vicente, "Incorporating minimum frobe- ´ nius norm models in direct search," *Computational Optimization and Applications*, vol. 46, pp. 265–278, 2010.

[50] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *2009 Ninth IEEE International Conference on Data Mining (ICDM)*, 2009, pp. 588– 597.

[51] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American Statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[52] N. Altman and M. Krzywinski, "Points of significance: interpreting p values," *Nature Methods*, vol. 14, no. 3, pp. 213–215, 2017.

[53] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang, "Capturing and enhancing in situ system observability for failure detection," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2018, p. 1–16.

[54] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "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, p. 807–817.

[55] K. Yin, M. Yan, L. Xu, Z. Xu, Z. Li, D. Yang, and X. Zhang, "Improving log-based anomaly detection with component-aware analysis," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 667–671.

[56] X. Li, P. Chen, L. Jing, Z. He, and G. Yu, "Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults," in *2020*

*IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 92–103.

[57] Y. Yuan, W. Shi, B. Liang, and B. Qin, “An approach to cloud execution failure diagnosis based on exception logs in openstack,” in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 124–131.

[58] J. Kim, V. Savchenko, K. Shin, K. Sorokin, H. Jeon, G. Pankratenko, S. Markov, and C.-J. Kim, “Automatic abnormal log detection by analyzing log history for providing debugging insight,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering:Software Engineering in Practice (ICSE-SEIP)*, 2020, p. 71–80.