# Identifying Impactful Service System Problems via Log Analysis

Shilin He*†
The Chinese University of Hong Kong
Hong Kong, China
slhe@cse.cuhk.edu.hk

Qingwei Lin
Microsoft Research
Beijing, China
qlin@microsoft.com

Jian-Guang Lou
Microsoft Research
Beijing, China
jlou@microsoft.com

Hongyu Zhang
The University of Newcastle
NSW, Australia
hongyu.zhang@newcastle.edu.au

Michael R. Lyu*
The Chinese University of Hong Kong
Hong Kong, China
lyu@cse.cuhk.edu.hk

Dongmei Zhang
Microsoft Research
Beijing, China
dongmeiz@microsoft.com

## ABSTRACT

Logs are often used for troubleshooting in large-scale software systems. For a cloud-based online system that provides 24/7 service, a huge number of logs could be generated every day. However, these logs are highly imbalanced in general, because most logs indicate normal system operations, and only a small percentage of logs reveal impactful problems. Problems that lead to the decline of system KPIs (Key Performance Indicators) are impactful and should be fixed by engineers with a high priority. Furthermore, there are various types of system problems, which are hard to be distinguished manually. In this paper, we propose Log3C, a novel clustering-based approach to promptly and precisely identify impactful system problems, by utilizing both log sequences (a sequence of log events) and system KPIs. More specifically, we design a novel cascading clustering algorithm, which can greatly save the clustering time while keeping high accuracy by iteratively sampling, clustering, and matching log sequences. We then identify the impactful problems by correlating the clusters of log sequences with system KPIs. Log3C is evaluated on real-world log data collected from an online service system at Microsoft, and the results confirm its effectiveness and efficiency. Furthermore, our approach has been successfully applied in industrial practice.

## CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*; *Maintaining software*;

## KEYWORDS

Log Analysis, Problem Identification, Clustering, Service Systems

*Also with Shenzhen Research Institute, The Chinese University of Hong Kong.
†Work done mainly during internship at Microsoft Research Asia.


*ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA*

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236083

**ACM Reference Format:**
Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2018. Identifying Impactful Service System Problems via Log Analysis. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236083

## 1 INTRODUCTION

For large-scale software systems, especially cloud-based online service systems such as Microsoft Azure, Amazon AWS, Google Cloud, high service quality is vital. Since these systems provide services to hundreds of millions of users around the world, a small service problem could lead to great revenue loss and user dissatisfaction.

Large-scale software systems usually generate logs to record system runtime information (e.g., states and events). These logs are frequently utilized in the maintenance and diagnosis of systems. When a failure occurs, inspecting recorded logs has become a common practice. Particularly, logs play a crucial role in the diagnosis of modern cloud-based online service systems, where conventional debugging tools are hard to be applied.

Clearly, manual problem diagnosis is very time-consuming and error-prone due to the increasing scale and complexity of large-scale systems. Over the years, a stream of methods based on machine learning have been proposed for log-based problem identification and troubleshooting. Some use supervised methods, such as classification algorithms [43], to categorize system problems. However, they require a large number of labels and substantial manual labeling effort. Others use unsupervised methods, such as PCA [41] and Invariants Mining [23] to detect system anomalies. However, these approaches can only recognize whether there is a problem or not but cannot distinguish among different types of problem.

To identify different problem types, clustering is the most pervasive method [7–9, 21]. However, it is hard to develop an effective and efficient log-based problem identification approach through clustering due to the following three challenges:

1) First, large-scale online service systems such as those of Microsoft and Amazon, often run on a $7 \times 24$ basis and support hundreds of millions of users, which yields an incredibly large quantity of logs. For instance, a service system of Microsoft that we studied can produce dozens of Terabytes of logs per day. Notoriously, conducting conventional clustering on data of such order-of-magnitude

consumes a great deal of time, which is unacceptable in practice [1, 12, 15, 18].

2) Second, there are many types of problems associated with the logs and clustering alone cannot determine whether a cluster reflects a problem or not. In previous work on log clustering, developers are required to verify the problems manually during the clustering process [21], which is tedious and time-consuming.

3) Third, log data is highly imbalanced. In a production environment, a well-deployed online service system operates normally most of the time. That is, most of the logs record normal operations and only a small percentage of logs are problematic and indicate impactful problems. The imbalanced data distribution can severely impede the accuracy of the conventional clustering algorithm [42]. Furthermore, it is intrinsic that some problems may arise less frequently than others; therefore, these rare problems emerge with fewer log messages. As a result, it is challenging to identify all problem types from the highly imbalanced log data.

To tackle the above challenges, we propose a novel problem identification framework, Log3C, using both log data and system KPI data. System KPIs (Key Performance Indicators such as service availability, average request latency, failure rate, etc.) are widely adopted in industry. They measure the health status of a system over a time period and are collected periodically.

To be specific, we propose a novel clustering algorithm, Cascading Clustering, which clusters a massive amount of log data by iteratively sampling, clustering, and matching log sequences (sequences of log events). Cascading clustering can significantly reduce the clustering time while keeping high accuracy. Further, we analyze the correlation between log clusters and system KPIs. By integrating the ***C**ascading **C**lustering* and ***C**orrelation analysis*, Log3C can promptly and precisely identify impactful service problems.

We evaluate our approach on real-world log data collected from a deployed online service system at Microsoft. The results show that our method can accurately find impactful service problems from large log datasets with high time performance. Log3C can precisely find out problems with an average precision of 0.877 and an average recall of 0.883. We have also successfully applied Log3C to the maintenance of many actual online service systems at Microsoft. To summarize, our main contributions are threefold:

- We propose Cascading Clustering, a novel clustering algorithm that can greatly save the clustering time while keeping high accuracy. The implementation is available on Github[1].
- We propose Log3C, which is a novel framework that integrates cascading clustering and correlation analysis. Log3C can automatically identify impactful problems from a large amount of log and KPI data efficiently and accurately.
- We evaluate our method using the real-world data from Microsoft. Besides, we have also applied Log3C to the actual maintenance of online service systems at Microsoft. The results confirm the usefulness of Log3C in practice.

The rest of this paper is organized as follows: In Section 2, we introduce the background and motivation. Section 3 presents the proposed framework and each procedure in detail. The evaluation of our approach is described in Section 4. Section 5 discusses the experiment results and Section 6 shares some success stories and experiences obtained from industrial practice. The related work and conclusion are presented in Section 7 and Section 8, respectively.

[1] https://github.com/logpai/Log3C

| | | |
|---|---|---|
| 01 | Name=Request (GET:http://AAA:1000/BBBB/sitedata.html) | t_41bx0 |
| 02 | Leaving Monitored Scope (EnsureListItemsData) Execution Time=52.9013 | t_51xi4 |
| 03 | HTTP request URL: /14/Emails/MrX(MrX@mail.com)/1c-48f0-b29.eml | t_23hl3 |
| 04 | HTTP Request method: GET | t_41bx0 |
| 05 | HTTP request URL: /55/RST/UVX/ADEG/Lists/Files/docXX.doc | t_01mu1 |
| 06 | Overridden HTTP request method: GET | t_41bx0 |
| 07 | HTTP request URL: http://AAA:1000/BBBB/sitedata.html | t_41bx0 |
| 08 | Leaving Monitored Scope (Request (POST:http://AAA:100/BBBB/ sitedata.html)) Execution Time=334.319268903038 | t_41bx0 **(Task_ID)** |

| | |
|---|---|
| E1 | Name=Request (*) |
| E2 | Leaving Monitored Scope (*) Execution Time = * |
| E3 | HTTP Request method: * |
| E4 | HTTP request URL: * |
| E5 | Overridden HTTP request method: * |

Log Parsing

**Figure 1: An Example of Log Messages and Log Events**

## 2 BACKGROUND AND MOTIVATION

Cloud-based online service systems, such as Microsoft Azure, Google Cloud, and Amazon AWS, have been widely adopted in the industry. These systems provide a variety of services and support a myriad of users across the world every day. Therefore, one system problem could cause catastrophic consequences. Thus far, service providers have made tremendous efforts to maintain high service quality. For example, Amazon AWS [2] and Microsoft Azure [25] claim to have "five nines", which indicates the service availability of 99.999%.

Although a lot of efforts have been devoted to quality assurance, in practice, online service systems still encounter many problems. To diagnose the problem, engineers often rely on system logs, which record system runtime information (e.g., states and events).

The top frame of Figure 1 shows eight real-world log messages from Microsoft (some fields are omitted for simplicity of presentation). Each log message comprises two parts: a constant part and a variable part. The constant part consists of fixed text strings, which describe the semantic meaning of a program event. The variable part contains parameters (e.g., URL) that record important system attributes. A log event is the abstraction of a group of similar log messages. As depicted in Figure 1, the log event for log message 3,5,7 is E4: **"HTTP request URL: ∗"**, where the constant part is the common part of these log messages (*"HTTP request URL:"*), and the asterisk represents the parameter part. Log parsing is the procedure that extracts log events from log messages, and we defer details to Section 3.1. A log sequence is a sequence of log events that record a system operation in the same task. In Figure 1, log message 1,4,6,7,8 are sequentially generated to record a typical HTTP request. These log messages share the same task ID (t_41bx0), and thereby the corresponding log sequence is: [E1, E3, E5, E4, E2].

For a well-deployed online service system, it operates normally in most cases and exhibits problems occasionally. However, it does not imply that problems are easy to identify. On the contrary, problems are hidden among a vast number of logs while most logs record the system's normal operations. In addition, there are various types of service problems, which may manifest different patterns, occur at different frequencies, and affect the service system in different

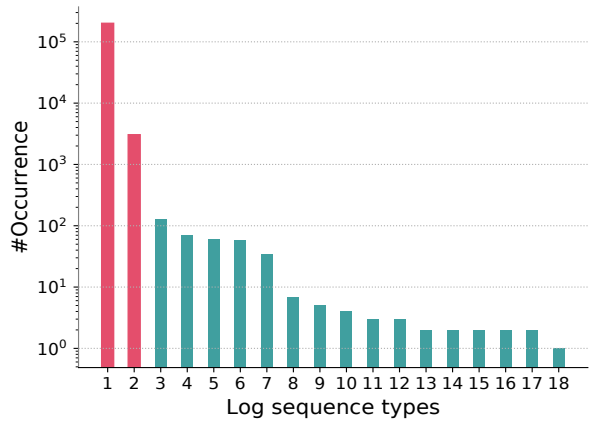


**Figure 2: Long Tail Distribution of Log Sequences**

manners. As a result, it is challenging to precisely and promptly identify the service problems from the logs.

As an example, Figure 2 shows the long tail distribution of 18 types of log sequences (in logarithmic scale for easy plotting), which are labeled by engineers from product teams. The first two types of log sequences occupy more than 99.8% of the total occurrences ("head") and are generated by normal system operations. The remaining ones indicate different problems, but they all together only take up less than 0.2% of all occurrences ("long tail"). Besides, the occurrences of distinct problem types varies significantly. For example, the first type of problem (the 3rd bar in Figure 2) is a "SQL connection problem", which shows that the server cannot connect a SQL database. The most frequent problem occurs over 100 times more often than the least frequent one. The distribution is highly imbalanced and exhibits strong long-tail property, which poses challenges for log-based problem identification.

Among all the problems, some are impactful because they can lead to the degradation of system KPIs. As aforementioned, system KPIs delineate the system's health status. A lower KPI value indicates that some system problems may have occurred and the service quality deteriorates. In our work, we leverage both log and KPI data to guide the identification of impactful problems. In practice, systems continuously generate logs, but the KPI values are periodically collected.

We use *time interval* to denote the KPI collection frequency. The value of time interval is typically 1 hour or more, which is set by the production team. In our setting, we use *failure rate* as the KPI, which is the ratio of failed requests to all requests within a time interval. In each time interval, there could be many logs but only one KPI value (e.g., one failure rate).

## 3 LOG3C: THE PROPOSED APPROACH

In this paper, we aim at solving the following problems: Given system logs and KPIs, how to detect impactful service system problems automatically? How to identify different kinds of impactful service system problems precisely and promptly?

To this end, we propose Log3C, whose overall framework is depicted in Figure 3. Log3C consists of four steps: log parsing, sequence vectorization, cascading clustering, and correlation analysis. In short, at each time interval, logs are parsed into log events and vectorized into sequence vectors, which are then grouped into multiple clusters through cascading clustering. However, we still cannot extrapolate whether a cluster is an impactful problem, which necessitates the use of KPIs. Consequently, in step four, we correlate clusters and KPIs over different time intervals to find impactful problems. More details are presented in the following sections.

### 3.1 Log Parsing

As aforementioned, log parsing extracts the log event for each raw log message since raw log messages contain some superfluous information (e.g., file name, IP address) that can hinder the automatic log analysis. The most straightforward way of log parsing is to write a regular expression for every logging statement in the source code, as adopted in [41]. However, it is tedious and time-consuming because the source code updates very frequently and is not always available in practice (e.g., third-party libraries). Thus, automatic log parsing without source code is imperative.

In this paper, we use an automatic log parsing method proposed in [13] to extract log events. Following this method, firstly, some common parameter fields (e.g., IP address), are removed using regular expressions. Then, log messages are clustered into coarse-grained groups based on weighted edit distance. These groups are further split into fine-grained groups of log messages. Finally, a log event is obtained by finding the longest common substrings for each group of raw log messages.

To form a log sequence, log messages that share the same task ID are linked together and parsed into log events. Moreover, we remove the duplicate events in the log sequence. Generally, repetition often indicates retrying operations or loops, such as continuously trying to connect to a remote server. Without removing duplicates, similar log sequences with different occurrences of the same event are identified as distinct sequences, although they essentially indicate the same system behavior/operation. Following the common practice [21, 32] in log analysis, we remove the duplicate log events.

### 3.2 Sequence Vectorization

After obtaining log sequences from logs in all time intervals, we compute the vector representation for each log sequence. We believe that different log events have different discriminative power in problem identification. As delineated in Step 2 of Figure 3, to measure the importance of each event, we calculate the event weight by combining the following two techniques:

**IDF Weighting:** IDF (Inverse Document Frequency) is widely utilized in text mining to measure the importance of words in some documents, which lowers the weight of frequent words while increasing rare words' weight [30, 31]. In our scenario, events that frequently appear in numerous log sequences cannot distinguish problems well because problems are relatively rare. Hence, the event and log sequence are analogous to word and document respectively. We aggregate log sequences in all time intervals together to calculate the IDF weight, which is defined in Equation 1, where $N$ is the total number of all log sequences and $n_e$ is the number of log sequences that contain the event $e$. With IDF weighting, frequent events have low weights, while rare events are weighted high.

$$w_{idf}(e) = \log\left(\frac{N}{n_e}\right) \quad (1)$$

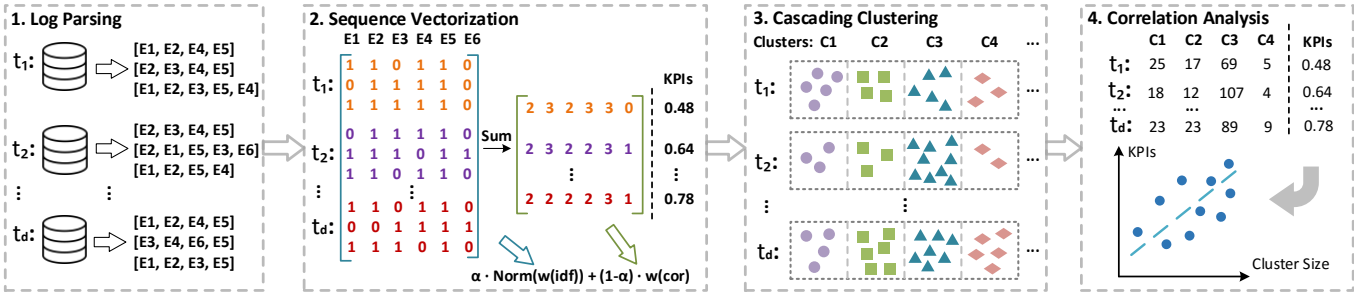


**Figure 3: Overall Framework of Log3C**

$$w(e) = \alpha * Norm(w_{idf}(e)) + (1 - \alpha) * w_{cor}(e) \quad (2)$$

**Importance Weighting:** In problem identification, it is intuitive that events strongly correlate with KPI degradation are more critical and should be weighted more. Therefore, we build a regression model between log events and KPI values to find the importance weight. To achieve so, as shown Figure 3, in each time interval, we sum the occurrence of each event in all log sequences (three in the example) as a summary sequence vector. After that, we get $d$ summary sequence vectors, and $d$ KPI values are also available as aforementioned. Then, a multivariate linear regression model is applied to evaluate the correlation between log events and KPIs. The weights $w_{cor}(e)$ obtained from the regression model serve as the importance weights for log events $e$. Note that the regression model only aims to find the importance weight for the log event.

As denoted in Equation 2, the final event weight is the weighted sum of IDF weight and importance weight. Besides, we use *Sigmoid* function [40] to normalize the IDF weight into the range of $[0, 1]$. Since the importance weight is directly associated with KPIs and is thereby more effective in problem identification, we value the importance weight more, i.e., $\alpha < 0.5$. In our experiments, we empirically set $\alpha$ to 0.2. Given the final event weights, the weighted sequence vectors can be easily obtained. For simplicity, hereafter, we use "sequence vectors" to refer to "weighted sequence vectors". Note that each log sequence has a corresponding sequence vector.

## 3.3 Cascading Clustering

Once all log sequences are vectorized, we group sequence vectors into clusters separately for each time interval. However, the conventional clustering methods are incredibly time-consuming when the data size is large [1, 12, 15, 18] because distances between any pair of samples are required. As mentioned in Section 2, log sequences follow the long tail distribution and are highly imbalanced. Based on the observation, we propose a novel clustering algorithm, *cascading clustering*, to group sequence vectors into clusters (different log sequence types) promptly and precisely, where each cluster represents one kind of log sequence (system behavior).

Figure 4 depicts the procedure of cascading clustering, which leverages iterative processing, including sampling, clustering, matching and cascading. The input of cascading clustering is all the sequence vectors in a time interval, and the output is a number of clusters. To be more specific, we first sample a portion of sequence vectors, on which a conventional clustering method (e.g., hierarchical clustering) is applied to generate multiple clusters. Then, a pattern can be extracted from each cluster. In the matching step, we match all the *original unsampled* sequence vectors with the patterns to determine their cluster. Those unmatched sequence vectors are collected and fed into the next iteration. By iterating these processes, all sequence vectors can be clustered rapidly and accurately. The reason behind is that large clusters are separated from the remaining data at the first several iterations.

*3.3.1 Sampling.* Given numerous sequence vectors in each time interval, we first sample a portion of them through Simple Random Sampling (SRS). Each sequence vector has an equal probability $p$ (e.g., 0.1%) to be selected. Suppose there are $N$ sequence vectors in the input data, then the sampled data size is $M = \lceil p * N \rceil$. After sampling, log sequence types (clusters) that dominate in the original input data are still dominant in the sampled data.

*3.3.2 Clustering.* After sampling $M$ sequence vectors from the input data, we group these sequence vectors into multiple clusters and extract a representative vector (pattern) from every cluster. To do so, we calculate the distance between every two sequence vectors and apply an ordinary clustering algorithm.

**Distance Metric:** During clustering, we use Euclidean distance as the distance metric, which is defined in Equation 3: $u$ and $v$ are two sequence vectors, and $n$ is the vector length, which is the number of log events. $u_i$ and $v_i$ are the $i$-th value in vector $u$ and $v$, respectively.

$$d(u, v) = \sqrt{\|u - v\|} = \sqrt{\sum_{i=1}^{n}(u_i - v_i)^2} \quad (3)$$

$$D(A, B) = max\{d(a, b), \quad \forall a \in A, \forall b \in B\} \quad (4)$$

$$\mu = \min\{d(x, P_j), \quad \forall j \in \{1, 2, ..., k\}\} \quad (5)$$

**Clustering Technique:** We utilize *Hierarchical Agglomerative Clustering (HAC)* to conduct clustering. At first, each sequence vector itself forms a cluster, and the closest two clusters are merged into a new one. To find the closest clusters, we use the complete linkage [38] to measure the cluster distance. As shown in Equation 4, $D$ is the cluster distance between two clusters $A$ and $B$, which is defined as the longest distance between any two elements (one in each cluster) in the clusters. The merging process continues until reaching a distance threshold of $\theta$. That is, the clustering stops when all the distances between clusters are larger than $\theta$. In Section 4.4, we also study the effect of different thresholds. After clustering, similar sequence vectors are grouped into the same cluster, while dissimilar sequence vectors are separated into different clusters.

**Pattern Extraction:** After clustering, a representative vector is extracted for each cluster, which serves as the pattern of a group of similar log sequences. To achieve so, we compute the mean

vector [3] of all sequence vectors in a cluster. Assume that there are $k$ clusters, then $k$ mean vectors (patterns) can be extracted to represent those clusters respectively.

*3.3.3 Matching.* As illustrated in Figure 4, we match each sequence vector in the *original unsampled* input data (of size $N$) to one of the $k$ patterns, which are obtained by clustering $M$ sampled sequence vectors. To this end, for each sequence vector $x$, we calculate the distance between it and every pattern. Furthermore, we compute the minimum distance $\mu$ as denoted in Equation 5, where $P$ is a set of all patterns. If the minimum distance $\mu$ is smaller than the threshold $\theta$ defined in the clustering step, the sequence vector $x$ is matched with a pattern successfully and thereby can be assigned to the corresponding cluster. Otherwise, this sequence vector is classified as mismatched. Note that those mismatched sequence vectors would proceed to the next iteration.

**Algorithm 1:** Cascading Clustering

**Input :** Sequence vector data $D$, Sample rate $p$, Clustering threshold $\theta$
**Output :** Sequence clusters *globalClusters*, Pattern set *gloablPatList*

```
1  misMatchData = D;
2  globalPatList = ∅; globalClusters = ∅;
3  while misMatchData! = ∅ do
4      SampleData = ∅;
5      /* Sampling sequence vectors                    */
6      foreach seqVec ∈ D do
7          if random(0, 1) <= p then
8              SampleData.append(seqVec);
9          end
10     end
11     /* Hierarchical clustering                      */
12     localClusters = HAC(SampledData, θ);
13     localPatList = patternExtraction(clusters);
14     /* Matching and finding mismatched data         */
15     newMismatchData = ∅;
16     foreach seqVec ∈ misMatchData do
17         foreach pat ∈ localPatList do
18             distList.append(dist(seqVec, pat));
19         end
20         if min(distList) > θ then
21             newMismatchData.append(seqVec);
22         end
23     end
24     misMatchData = newMismatchData;
25     globalPatList.extend(localPatList);
26     globalClusters.extend(localClusters);
27 end
28 Return globalClusters, globalPatList
```

*3.3.4 Cascading.* After going through the above three processes (i.e., sampling, clustering, and matching), the majority of the data in a time interval can be grouped into clusters, while some sequence vectors may remain unmatched. Hence, we further process the unmatched data by repeating the abovementioned procedures. That is, during each iteration, new clusters are grouped based on current mismatched data, new patterns are extracted, and new mismatched data are produced. In our experiments, we cascade these repetitions until all the sequence vectors are successfully clustered.

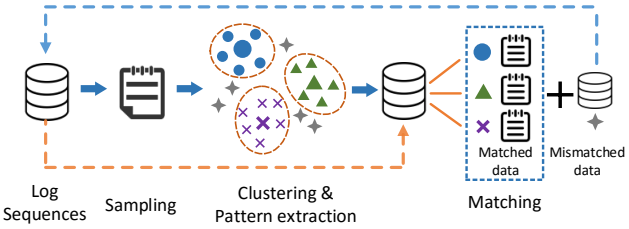


**Figure 4: Overview of Cascading Clustering Algorithm**

*3.3.5 Algorithm and Time Complexity Analysis.* The pseudo code of *Cascading Clustering* is detailed in Algorithm 1. The algorithm takes sequence vectors in a time interval as input data, with sample rate and clustering distance threshold as hyper-parameters. After cascading clustering, the algorithm outputs all sequence vector clusters and a set of patterns. To initialize, we assign all sequence vectors $D$ to the mismatched data. Besides, we define two global variables (lines 1-2) to store the clusters and patterns. Then, the sampled data is obtained with a sampling rate of $p$ (lines 3-10). In lines 12 and 13, we perform the hierarchical agglomerative clustering (HAC) on sampled data with threshold $\theta$ and extract the cluster patterns. In fact, other clustering methods (e.g., K-Means) are also applicable here. During the matching process (line 16-23), we use *distList* (line 17-19) to store the distances between a sequence vector and every cluster pattern. The sequence vector is allocated to the closest cluster if the distance is smaller than the threshold $\theta$. The remaining mismatched data is updated (lines 24) and processed in the next cascading round.

We now analyze the time complexity of the proposed algorithm. Note that only the core parts of the cascading clustering algorithm are considered, i.e., distance calculation and matching, because they consume most of the time. We set the data size to $N$, which is a large number (e.g., larger than $10^6$). The sample rate $p$ is usually a user-defined small number (e.g., less than 1%). For standard hierarchical agglomerative clustering, the distance calculation takes $O(N^2)$ time complexity, and no matching is involved. For cascading clustering, suppose that $pN$ data instances are selected and clustered into $k_1$ groups firstly, and further $N_1$ instances are mismatched. Therefore, the time complexity of the first round is $T_1 = p^2N^2 + k_1N$. After $t$ iterations, the total number of clusters is $K = \sum_{i=1}^{t} k_i$. Therefore, the overall time complexity $T(cc)$ is calculated as:

$$\begin{aligned} T(cc) &= p^2N^2 + k_1N + p^2N_1^2 + k_2N_1 + \ldots + p^2N_t^2 + k_tN_{t-1} \\ &= p^2N^2 + \sum_{i=1}^{t} p^2N_i^2 + k_1N + \sum_{i=1}^{t-1} k_{i+1}N_i \\ &< p^2N^2 + tp^2N_1^2 + KN < (pN + p\sqrt{t}N_1 + \sqrt{KN})^2 \end{aligned}$$

Since $N$ is a large number and $K$ is the total number of clusters, we have $K \ll N$ and $\sqrt{KN} \ll N$. Because the data follows the long tail distribution and the "head" occupies most of the data (e.g., more than 80%). After several iterations, most data can be successfully clustered and matched. Recall that $p \ll 1$, we then have $p\sqrt{t}N_1 \ll N$

and $pN \ll N$. Therefore, the inequality $pN + p\sqrt{t}N_1 + \sqrt{KN} < N$ holds and the left-hand side is much smaller. Given that $f(X) = X^2$ is a monotonic increasing function ($X \geq 0$), where $f(X)$ decreases with the decreasing of $X$. We then have $(pN+p\sqrt{t}N_1+\sqrt{KN})^2 \ll N^2$ satisfied, which indicates that cascading clustering consumes much less time than standard clustering in terms of distance calculation and matching. In our experiments, we empirically evaluate the time performance of cascading clustering, and the results support our theoretical analysis.

### 3.4 Correlation Analysis

As described in Figure 3, log sequence vectors are grouped into multiple clusters separately in each time interval. These clusters only represent different types of log sequences (system behaviors) but may not necessarily be problems. From the clusters, we identify the impactful problems that lead to the degradation of KPI. Intuitively, KPI degrades more if impact problems occur more frequently. Hence, we aim to identify those clusters that highly correlate with KPI's changes. To do so, we model the correlation between cluster sizes and KPI values over multiple time intervals. Unlike the importance weighting in Section 3.2 that discriminates the importance of different log events, this step attempts to identify impactful problems from clusters of sequence vectors.

More specifically, we utilize a multivariate linear regression (MLR) model (Equation 6), which correlates independent variables (cluster sizes) with the dependent variable (KPI). Among all independent variables, those have statistical significance make notable contributions to the dependent variable. Moreover, the corresponding clusters indicate impactful problems, whose occurrences contribute to the change of KPI. Statistical significance is widely utilized in the identification of important variables [17, 34].

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} & \dots & c_{1n} \\ c_{21} & c_{22} & c_{23} & \dots & c_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{d1} & c_{d2} & c_{d3} & \dots & c_{dn} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_d \end{bmatrix} = \begin{bmatrix} KPI_1 \\ KPI_2 \\ \vdots \\ KPI_d \end{bmatrix} \quad (6)$$

In Equation 6, suppose there are $n$ clusters generated during $d$ time intervals, $c_{ij}$ represents the cluster size (the number of sequence vectors) of the $j$-th cluster at time interval $i$. $KPI_i$ is the system KPI value at time interval $i$. $\beta_i$ and $\varepsilon_i$ are coefficients and error terms that would be learned from data.

To find out which clusters are highly correlated with the KPI values, we adopt the *t-statistic*, which is a widely used statistical hypothesis test. In our MLR model, important clusters (indicating impactful problems) make major contributions to the change of KPIs, and their coefficients should not be zero. Therefore, we make a null hypothesis for each independent variable that its coefficient is zero. Then, a two-tailed t-test is applied to measure the significant difference of each coefficient, i.e., the probability $p$ that the null hypothesis is true. A lower *p-value* is preferred since it represents a higher probability of the null hypothesis being rejected. If *p-value* is less than or equal to a given threshold $\alpha$ (significance level), the corresponding cluster implies an impactful problem that affects the KPI. In this paper, we set $\alpha$ to 0.05, which is a common setting in hypothesis test.

## 4 EXPERIMENTS

In this section, we evaluate our approach using real-world data from industry. We aim at answering the following research questions:

RQ1: How effective is the proposed Log3C approach in detecting impactful problems?

RQ2: How effective is the proposed Log3C approach in identifying different types of problems?

RQ3: How does cascading clustering perform under different configurations?

### 4.1 Setup

**Datasets:** We collect the real-world data from an online service system X of Microsoft. Service X is a large scale online service system, which serves hundreds of millions of users globally on a 24/7 basis. Service X has been running over the years and has achieved high service availability. The system is operating in multiple data centers with a large number of machines, each of which produces a vast quantity of logs every hour. Service X utilizes the load balancing strategies, and end user requests are accepted and dispatched to different back-ends. There are many components at the application level, and each component has its specific functionalities. Most user requests involve multiple components on various servers. Each component generates logs and all the logs are uploaded to a distributed HDFS-like data storage automatically. Each machine or component has a probability to fail, leading to various problem types. We use *failure rate* as the KPI, which shows the percentage of failed requests in a time interval.

**Table 1: Summary of Service X Log Data**

| Data | Snapshot starts | #Log Seq (Size) | #Events | #Types |
|---|---|---|---|---|
| Data 1 | Sept 5th 10:50 | 359,843 (722MB) | 365 | 16 |
| Data 2 | Oct 5th 04:30 | 472,399 (996MB) | 526 | 21 |
| Data 3 | Nov 5th 18:50 | 184,751 (407MB) | 409 | 14 |

Service X produces a large quantity of log data consisting of billions of log messages. However, it is unrealistic to evaluate Log3C on all the data due to the lack of labels. The labeling difficulties origin from two aspects: first, the log sequences are of huge size. Second, various problem types can exist, and human labeling is very time-consuming and error-prone. Therefore, we extract logs that were generated during a specified period[2] on three different days. In this way, three real-world datasets (i.e., Data 1, 2, 3) are obtained, as shown in Table 1. Besides the log data, we also collect the corresponding KPI values. During labeling, product team engineers utilize their domain knowledge to identify the normal log sequences. Then, they manually inspect the rest log sequences from two aspects: 1) Does the log sequence indicate a problem? 2) What is the problem type? Table 1 shows the number of problem types identified in the evaluation datasets. Note that the manual labels are only used for evaluating the effectiveness of Log3C in our experiments. Log3C is an unsupervised method, which only requires log and KPI data to identify problems.

**Implementation and Environments:** We use Python to implement our approach for easy comparison, and run the experiments on a Windows Server 2012 (Intel(R) Xeon(R) CPU E5-4657L v2 @ 2.40GHz 2.40 with 1.00TB Memory).

[2]The actual period is anonymous due to data sensitivity.

**Table 2: Accuracy of Problem Detection on Service X Data**

| Data | Data 1 | | | Data 2 | | | Data 3 | | |
|---|---|---|---|---|---|---|---|---|---|
| Metrics | Precision | Recall | F1-measure | Precision | Recall | F1-measure | Precision | Recall | F1-measure |
| PCA | 0.465 | 0.946 | 0.623 | 0.142 | 0.834 | 0.242 | 0.207 | 0.922 | 0.338 |
| Invariants Mining | 0.604 | 1 | 0.753 | 0.160 | 0.847 | 0.269 | 0.168 | 0.704 | 0.271 |
| **Log3C** | **0.900** | 0.920 | **0.910** | **0.897** | 0.826 | **0.860** | **0.834** | 0.903 | **0.868** |

**Evaluation Metrics:** To measure the effectiveness of Log3C in problem detection, we use the Precision, Recall, and F1-measure. Given the labels from engineers, we calculate these metrics as follows:

*Precision:* the percentage of log sequences that are correctly identified as problems over all the log sequences that are identified as problems: Precision = $\frac{TP}{TP+FP}$.

*Recall:* the percentage of log sequences that are correctly identified as problems over all problematic log sequences: Recall = $\frac{TP}{TP+FN}$.

*F1-measure:* the harmonic mean of *precision* and *recall*.

*TP* is the number of problematic log sequences that correctly detected by Log3C, *FP* is the number of non-problematic log sequences that are wrongly identified as problems by Log3C. *FN* is the number of problematic log sequences that are not detected by Log3C, *TN* is the number of log sequences that are identified as non-problematic by both engineers and Log3C.

To measure the effectiveness of clustering, we use the *Normalized Mutual Information* (NMI), which is a widely used metric for evaluating clustering quality [35]. The value of NMI ranges from 0 to 1. The closer to 1, the better the clustering results. To measure the efficiency of cascading clustering, we record the total time (in seconds) spent on clustering.

## 4.2 RQ1: Effectiveness of Log3C in Detecting Impactful Problems

To answer RQ1, we apply Log3C to the three datasets collected from Service X and evaluate the precision, recall, and F1-measure. The results are shown in Table 2. Log3C achieves satisfactory accuracy, with recall ranging from 0.826 to 0.92 and precision ranging from 0.834 to 0.9. The F1-measures on the three datasets are 0.91, 0.86, and 0.868, respectively.

Furthermore, we compare our method with two typical methods: PCA [41] and Invariants Mining [23]. All these three methods are unsupervised, log-based problem identification methods. PCA projects the log sequence vectors into a subspace. If the projected vector is far from the majority, it is considered as a problem. Invariants Mining extracts the linear relations (invariants) between log event occurrences, which hypothesizes that log events are often pairwise generated. For example, when processing files, "File A is opened" and "File A is closed" should be printed as a pair. Log sequences that violate the invariants are regarded as problematic.

Log3C achieves good recalls (similar to those achieved by two comparative methods) and surpasses the comparative methods concerning precision and F1-measure. The absolute improvement in F1-measure ranges from 15.7% to 61.8% on the three datasets. The two comparative methods all achieve low precision (less than 0.61), while the precisions achieved by Log3C are greater than 0.83. We also explore the reasons for the low precision of the competing methods. In principle, PCA and Invariants Mining aim at finding the abnormal log sequences from the entire data. However, some rare user/system behaviors can be wrongly identified as problems. Thus, many false positives are generated, which result in high recall and low precision. More details are described in Section 6.2.1.

Regarding the time usage of problem detection, on average, it takes Log3C 223.93 seconds to produce the results for each dataset, while PCA takes around 911.97 seconds and invariants mining consumes 1830.78 seconds. The time performance of Log3C is satisfactory considering the large amount of log sequence data.

## 4.3 RQ2: Effectiveness of Log3C in Identifying Different Types of Problems

In Log3C, we propose a novel cascading clustering algorithm to group the log sequences into clusters that represent different types of problems. For the ease of evaluation, clusters that represent normal system behaviors are considered as special "non-problem" types. In this section, we use NMI to evaluate the effectiveness of Log3C in identifying different types of problems. We also compare the performance of cascading clustering (denoted as CC) with the standard clustering method hierarchical agglomerative clustering (denoted as SC). To compare fairly, we implement a variant of Log3C that replaces CC with SC, denoted as *Log3C-SC*. All the other settings (e.g., distance threshold, event weight) remain the same.

**Table 3: NMI of Clustering on Service X Data**

| | Size | 10k | 50k | 100k | 200k |
|---|---|---|---|---|---|
| Data 1 | Log3C-SC | 0.659 | 0.706 | 0.781 | 0.822 |
| | **Log3C** | **0.720** | **0.740** | **0.798** | **0.834** |
| | Size | 10k | 50k | 100k | 200k |
| Data 2 | Log3C-SC | 0.610 | 0.549 | 0.600 | 0.650 |
| | **Log3C** | **0.624** | 0.514 | **0.663** | **0.715** |
| | Size | 10k | 50k | 100k | 180k |
| Data 3 | Log3C-SC | 0.601 | 0.404 | 0.792 | 0.828 |
| | **Log3C** | **0.680** | **0.453** | **0.837** | **0.910** |

Table 3 presents the NMI results, in which data size refers to the number of log sequences. We sample four subsets of each dataset with size ranging from 10k to 200k (for Data 3, 180k is used instead of 200k as its total size is around 180k). From the table, we can conclude that Log3C (with cascading clustering) is effective in grouping numerous log sequences into different clusters and outperforms Log3C-SC on all three datasets. Besides, with the increase of data size, clustering accuracy increases. This is because more accurate event weights can be obtained with more data during sequence vectorization. For instance, when 200k data is used, the NMI values achieved by Log3C range from 0.715 to 0.91 (180k for Data 3).

We also evaluate the time performance of cascading clustering. Table 4 shows that our cascading clustering (CC) dramatically save

**Table 4: Time Performance (in Seconds) of Clustering**

| | Size | 10k | 50k | 100k | 200k |
|---|---|---|---|---|---|
| Data 1 | SC | 127.6 | 2319.2 | 9662.3 | 38415.5 |
| | **CC** | 1.0 | 4.3 | 9.2 | 20.7 |
| | Size | 10k | 50k | 100k | 200k |
| Data 2 | SC | 80.6 | 2469.1 | 8641.2 | 38614.0 |
| | **CC** | 0.7 | 3.8 | 9.5 | 18.9 |
| | Size | 10k | 50k | 100k | 180k |
| Data 3 | SC | 81.5 | 2417.2 | 8761.2 | 33728.3 |
| | **CC** | 0.8 | 4.0 | 8.8 | 18.3 |

the time in contrast to the standard HAC clustering (SC), and the comparison is more noticeable when the data size grows. For example, our approach is around 1800x faster than standard clustering on dataset 1 with a size of 200k.

## 4.4 RQ3: Cascading Clustering under Different Configurations

In Section 3.3, we introduced two important hyper-parameters that are used in cascading clustering: the sample rate $p$ and the distance threshold $\theta$. In this section, we evaluate clustering accuracy and time performance under different configurations of parameters. We conduct the experiments on Data 1, but the results are also applicable to other datasets.

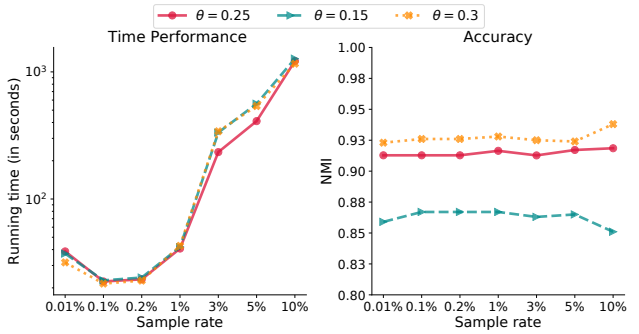


**Figure 5: Cascading Clustering on Service X Data under Different Configurations**

**Distance threshold $\theta$:** We first fix the sample rate (1%) and vary the distance threshold $\theta$ for cascading clustering. The clustering accuracy (NMI) is given in Table 5. When $\theta$ is 0.30, the highest NMI value (0.928) is achieved. However, we also observe that NMI changes slightly when the threshold changes within a reasonable range. The results show that our proposed cascading clustering algorithm is insensitive to the distance threshold to some degree.

**Sample rate $p$:** It is straightforward that sample rate can affect the time performance of cascading clustering because it takes more time to do clustering on a larger dataset. To verify it, we change the sample rate while fixing the distance threshold. Figure 5 depicts the results. We conduct the experiments with different sample rates under three distance thresholds (0.15, 0.25, and 0.3). The left sub-figure shows that a higher sample rate generally causes more time usage. However, when the sample rate is very small, e.g., 0.01%, a little more time is required. This is because, in cascading clustering, a small sample rate leads to more iterations of clustering, which hampers the efficiency of cascading clustering. Besides, we also

**Table 5: NMIs of Cascading Clustering under Different Distance Thresholds $\theta$**

| $\theta$ | 0.10 | 0.15 | 0.20 | 0.25 | 0.30 | 0.35 | 0.40 | 0.45 |
|---|---|---|---|---|---|---|---|---|
| NMI | 0.848 | 0.867 | 0.912 | 0.916 | 0.928 | 0.898 | 0.898 | 0.887 |

evaluate the clustering accuracy under different sample rates. As shown in the right sub-figure of Figure 5, clustering accuracy (NMI) is relatively stable when the sample rate changes. It shows that the NMI value floats slightly with a small standard deviation of 0.0071. In summary, we can conclude that generally, a small sample rate does not affect clustering accuracy and cost much less time. This finding can guide the setting of the sample rate in practice.

# 5 DISCUSSIONS

## 5.1 Discussions of Results

*5.1.1 Performance of Cascading Clustering with Different Numbers of Clusters.* In Section 4, we explored the efficiency and effectiveness of cascading clustering on real-world datasets. In this section, we evaluate our cascading clustering algorithm on some synthetic datasets. More specifically, we generate some synthetic datasets with different number of clusters.

To simulate a scenario that is similar to problem identification, we synthesize several synthetic datasets consisting of multiple clusters, where the cluster sizes follow the long tail distribution. In more detail, 1) we firstly synthesize a dataset of multiple clusters, and the data sample dimension is fixed at 200. The data samples in each cluster follow the multivariate normal distribution [39]. 2) Then, we use the pow law function (i.e., $f(x) = \alpha x^{-k}$) to determine the size of each cluster with some Gaussian noises added, as noises always exist in real data. In this way, we can generate multiple datasets with different data sizes (from 20k to 600k) and various numbers of clusters (from 20 to 200).

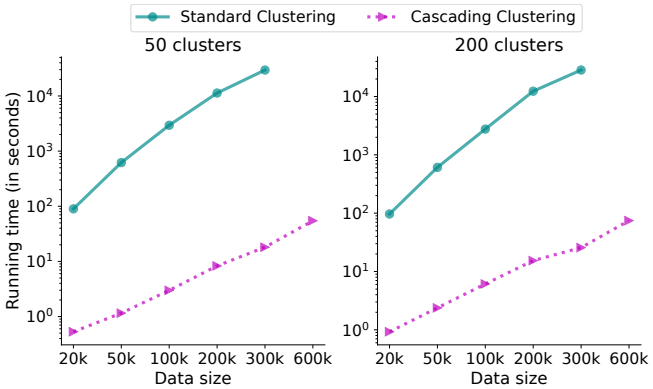


**Figure 6: Time Performance of Clustering Methods on Synthetic Data with 50 Cluster (Left) and 200 Clusters (Right)**

Figure 6 shows the time performance (in logarithmic scale for easy plotting) of standard clustering and cascading clustering where the number of clusters is 50 and 200. We also vary the synthetic data size from 20k to 600k. It is clear that cascading clustering requires much less time than standard clustering (hierarchical agglomerative clustering), with different cluster numbers. For example, standard clustering takes 11512.2 seconds (around 3.19 hours) on 200k data

**Table 6: NMIs of Standard Clustering (SC) and Cascading Clustering (CC) on Synthetic Data**

| Size | 20k | | 50k | | 100k | | 200k | | 300k | | 600k | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #Clusters | SC | CC | SC | CC | SC | CC | SC | CC | SC | CC | SC | CC |
| 20 | 0.958 | 0.947 | 0.885 | 0.918 | 0.837 | 0.850 | 0.786 | 0.791 | 0.758 | 0.774 | - | 0.725 |
| 50 | 0.971 | 0.961 | 0.937 | 0.960 | 0.906 | 0.916 | 0.864 | 0.883 | 0.837 | 0.854 | - | 0.811 |
| 100 | 0.976 | 0.995 | 0.956 | 0.979 | 0.927 | 0.939 | 0.903 | 0.911 | 0.889 | 0.892 | - | 0.859 |
| 200 | 0.988 | 0.990 | 0.973 | 0.973 | 0.955 | 0.953 | 0.929 | 0.937 | 0.914 | 0.925 | - | 0.896 |

with 50 clusters, while cascading clustering (sample rate is 1%) only takes 10.3 seconds on the same dataset. Our cascading clustering is more than 1000× faster than standard clustering.

In Table 6, we measure the clustering accuracy (in terms of NMI) under different data sizes and cluster numbers. We can conclude from the table that, overall, cascading clustering leads to equal or slightly better accuracy when compared with the standard clustering. The main reason is that our cascading clustering algorithm is specially designed for long-tailed data. The small clusters can be precisely clustered. Moreover, the evaluation results of standard clustering on 600k data are not available due to the out-of-memory (more than 1TB) computation. From Table 6, we can also observe that clustering accuracy increases with the increase of cluster number and decreases when the data size increases.

*5.1.2 Impact of Log Data Quality.* The quality of log data is crucial to log-based problem identification. For a large-scale service system, logs are usually generated on local machines, which are then collected and uploaded to a data center separately. During the process, some logs may be missed or duplicated. For duplicated logs, they do not affect the accuracy of Log3C as duplicates are removed from log sequences as described in Section 3.2. To evaluate the impact of missing log data, we randomly remove a certain percentage (missing rate) of logs from Data 1 and then evaluate the accuracy of Log3C. We use three different missing rates 0.1%, 0.5%, and 1%. The resulting F1-measures are 0.877, 0.834, 0.600, respectively. It can be concluded that a higher missing rate could lead to a lower problem identification accuracy. Therefore, we suggest ensuring the log data quality before applying Log3C in practice.

## 5.2 Threats to Validity

We have identified the following threats to validities:

**Subject Systems**: In our experiment, we only collect log data from one online service system (Service X). This system is a typical, large-scale online system, from which sufficient data can be collected. Furthermore, we have applied our approach to the maintenance of actual online service systems of Microsoft. In the future, we will evaluate Log3C on more subject systems and report the evaluation results.

**Selection of KPI**: In our experiments, we use *failure rate* as the KPI for problem identification. *failure rate* is an important KPI for evaluating system service availability. There are also other KPIs such as mean time between failures, average request latency, throughput, etc. In our future work, we will experiment with problem identification concerning different KPI metrics.

**Noises in labeling:** Our experiments are based on three datasets that are collected as a period of logs on three different days. The engineers manually inspected and labeled the log sequences. Noises (false positives/negatives) may be introduced during the manual labeling process. However, as the engineers are experienced professionals of the product team who maintain the service system, we believe the amount of noise is small (if it exists).

# 6 SUCCESS STORY AND LESSONS LEARNED

## 6.1 Success Story

Log3C is successfully applied to Microsoft's Service X system for log analysis. Service X provides online services to hundreds of millions of global end users on $7 * 24$ basis. For online service systems like Service X, inspecting logs is the only feasible way for fault diagnosis. In Service X, more than one Terabyte of logs (around billions) are generated in a few hours, and it is a great challenge to process the great volume of logs. A distributed version of Log3C is developed and employed in Service X. Billions of logs can be handled within hours using our method, which helps the service team in identifying different log sequence types and detecting system problems. For example, in April 2015, a severe problem occurred to one component of Service X on some servers. The problem was caused by an incompatibility issue between a patch and a previous product version during a system upgrade. The service team received lots of user complains regarding this problem. Our Log3C successfully detected the problem and reported it to the service team. The service team also utilized Log3C to investigate the logs and precisely identified the type of the problem. With the help of Log3C, the team quickly resolved this critical issue and redeployed the system.

Log3C is also integrated into Microsoft's Product B, an integrated environment for analyzing the root causes of service issues. Tens of billions of logs are collected and processed by Product B every day, in which Log3C is the log analysis engine. Using Log3C, Product B divides the log sequences into different clusters and identifies many service problems automatically. Log3C greatly reduces engineers' efforts on manually inspecting the logs and pinpointing root causes of failures. Furthermore, fault patterns are also extracted and maintained for analyzing similar problems in the future.

## 6.2 Lessons Learned

*6.2.1 Problems != Outliers.* Recent research [23, 41] proposed many approaches to detect system anomalies using data mining and machine learning techniques. These approaches work well for relatively small systems. Their ideas are mainly based on the following hypothesis: systems are regular most of the time and problems are "outliers". Many current approaches try to detect the "outliers" from a huge volume of log data. For example, PCA [41] attempts to map all data to a normal subspace, and those cannot be projected to the normal space are considered as anomalies.

However, the outliers are not always real problems. Some outliers are caused by certain infrequent user behaviors, e.g., rarely-used system features. Our experiences with the production system reveal that there are indeed many rare user behaviors, which are not real problems. A lot of effort could be wasted by examining these false positives. In our work, we utilize system KPI to guide the identification of real system problems.

*6.2.2 The Trend of Problem Reports Is Important.* In production, engineers not only care about the occurrence of a problem but also about the number of problem reports (i.e., the instances of problems) over time (which reflects the number of users that are affected by the problem over time). Through our communication with a principal architect of a widely-used service in Microsoft, we conclude three types of important trends: 1) Increasing. When the size of one certain problem continuously increases for a period, the production team should be notified. This is because the number of problem reports may accumulate and cause even serious consequences. 2) The appearance of new problems: when a previously unknown problem appears, it is often a sign of new bugs, which may be introduced by software updates or a newly launched product feature. 3) The disappearance of problems: The disappearing trend is very interesting. In production, after fixing a problem, the scale of the problem is expected to decrease. However, sometimes the disappearing trend may stop at a certain point (the service team continues to receive reports for the same problem), which often indicates an incomplete bug-fix or a partial solution. More debugging and diagnosis work are needed to identify the root cause of the problem and propose a complete bug-fixing solution.

# 7 RELATED WORK

## 7.1 Log-based Problem Identification

Logs have been widely used for the maintenance and diagnosis of various software systems with the abundant information they provide. Log-based problem identification has become a very popular area [13, 26, 28, 43] in recent years. Typically, based on information extracted from logs, these work employ machine learning and data mining techniques to analyze logs for anomaly detection and problem diagnosis [22, 23, 27, 41].

The target of anomaly detection is to find system's abnormal behaviors and give feedback to engineers for further diagnosis. Lou et al. [23] mined the invariants (linear relationships) among log events to detect anomalies. Liang et al. [20] trained a SVM classifier to detect failures. However, anomaly detection can only determine whether a log sequence is abnormal or not; it cannot determine if there is a real problem. Our Log3C can not only detect system problems but also cluster them into various types.

Problem identification aims at categorizing different types of problems by grouping similar log sequences together. Lin et al. [21] proposed a clustering-based approach for problem identification. Based on testing environment logs, a knowledge base is built firstly and updated in production environment. However, it requires manual examination when new problems appear. Yuan et al. [43] employed a classification method to categorize system traces by calculating the similarity with traces of existing and known problems. Beschastnikh et al. [5] inferred system behaviors by utilizing logs, which can support anomaly detection and bug finding. Ding et al. [10, 11] correlated logs with system problems and mitigation solutions when similar logs appear. Shang et al. [32] identified problems by grouping the same log sequences after removing repetition and permutations. However, they ignored different importance of log events and the similarity between two log sequences. In our work, we use cascading clustering to quickly and precisely cluster log sequences into different groups, and correlate clusters with the KPIs to identify problems. Our approach does not require manual examination, nor a knowledge base of known problems.

## 7.2 Log Parsing and Logging Practice

Logs cannot be utilized towards automatic analysis before being parsed into log events. Many log parser (e.g. LogSig [36], SLCT [37], IPLoM [24]) have been proposed in recent years, and the parsing accuracy can significantly affect the downstream log analysis tasks [16]. Xu et al. [41] extracted log events from console logs using the source code. Fu et al. [13] parsed logs by clustering with weighted edit distance. Makanju et al. [24] proposed a lightweight log parsing method by iteratively partitioning logs into subgroups and extracting log event.

Recently, there are also research on logging practice [4, 6, 14, 29, 32, 44, 45]. For example, Fu et al. [14], Yuan et al. [44], and Shang et al. [33] provide suggestions to developers by exploring the logging practice on some open-source systems and industry products. Zhu et al. [45] proposed a method to guide developers on whether to write logging statements. Kabinna1 et al. [19] examined the stability of logging statements and suggested to developers the unstable ones. The above research improves the quality of logs, which could in turn help with log-based problem identification.

# 8 CONCLUSION

Large-scale online service systems generate a huge number of logs, which can be used for troubleshooting purpose. In this paper, we propose Log3C, a novel framework for automated problem identification via log analysis. At the heart of Log3C is cascading clustering, a novel clustering algorithm for clustering a large number of highly imbalanced log sequences. The clustered log sequences are correlated with system KPI through a regression model, from which the clusters that represent impactful problems are identified. We evaluate Log3C using the real-world log data. Besides, we also apply our approach to the maintenance of actual online service systems. The results confirm the effectiveness and efficiency of Log3C in practice.

In the future, we will apply Log3C to a variety of software systems to further evaluate its effectiveness and efficiency. Also, the proposed Cascading Clustering algorithm is a general algorithm, which can be applied to a wide range of problems as well.

# ACKNOWLEDGEMENT

The work described in this paper was supported by the National Natural Science Foundation of China (Project No. 61332010, 61472338, and 61828201), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14210717 of the General Research Fund), the National Basic Research Program of

China (973 Project No. 2014CB347701), and the Microsoft Research Asia (2018 Microsoft Research Asia Collaborative Research Award).

We thank the principal architect Anthony Bloesch and the intern student Yanxia Xu from product teams for their valuable work. We also thank partners at Microsoft product teams for their collaboration and suggestions on the application of Log3C in practice.

## REFERENCES

[1] Osama Abu Abbas. 2008. Comparisons Between Data Clustering Algorithms. *International Arab Journal of Information Technology (IAJIT)* 5, 3 (2008).
[2] Amazon. 2018. Amazon Web Service. https://aws.amazon.com/. [Online; accessed July-2018].
[3] Theodore Wilbur Anderson, Theodore Wilbur Anderson, Theodore Wilbur Anderson, Theodore Wilbur Anderson, and Etats-Unis Mathématicien. 1958. *An introduction to multivariate statistical analysis*. Vol. 2. Wiley New York.
[4] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. 2016. The bones of the system: a case study of logging and telemetry at microsoft. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 92–101.
[5] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 468–479.
[6] Marcello Cinque, Domenico Cotroneo, Raffaele Della Corte, and Antonio Pecchia. 2014. What logs should you look at when an application fails? insights from an industrial case study. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 690–695.
[7] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 1084–1093.
[8] William Dickinson, David Leon, and Andy Podgurski. 2001. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society, 339–348.
[9] Nicholas DiGiuseppe and James A Jones. 2012. Software behavior and failure clustering: An empirical study of fault causality. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 191–200.
[10] Rui Ding, Qiang Fu, Jian-Guang Lou, Qingwei Lin, Dongmei Zhang, Jiajun Shen, and Tao Xie. 2012. Healing online service systems via mining historical issue repositories. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 318–321.
[11] Rui Ding, Qiang Fu, Jian Guang Lou, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Mining historical issue repositories to heal large-scale online service systems. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 311–322.
[12] Adil Fahad, Najlaa Alshatri, Zahir Tari, Abdullah Alamri, Ibrahim Khalil, Albert Y Zomaya, Sebti Foufou, and Abdelaziz Bouras. 2014. A survey of clustering algorithms for big data: Taxonomy and empirical analysis. *IEEE transactions on emerging topics in computing* 2, 3 (2014), 267–279.
[13] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 149–158.
[14] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 24–33.
[15] Jiawei Han, Jian Pei, and Micheline Kamber. 2011. *Data mining: concepts and techniques*. Elsevier.
[16] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2016. An evaluation study on log parsing and its use in log mining. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 654–661.
[17] James D. Herbsleb and Audris Mockus. 2003. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Trans. Softw. Eng.* 29, 6 (June 2003), 481–494. https://doi.org/10.1109/TSE.2003.1205177
[18] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. 1999. Data clustering: a review. *ACM computing surveys (CSUR)* 31, 3 (1999), 264–323.
[19] Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E Hassan. 2016. Examining the stability of logging statements. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 326–337.
[20] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in ibm bluegene/l event logs. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*. IEEE, 583–588.
[21] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 102–111.
[22] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. 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*. ACM, 557–566.
[23] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. 2010. Mining Invariants from Console Logs for System Problem Detection.. In *USENIX Annual Technical Conference*.
[24] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2012. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering* 24, 11 (2012), 1921–1936.
[25] Microsoft. 2018. Microsoft Azure. https://azure.microsoft.com/en-us/. [Online; accessed July-2018].
[26] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 26–26.
[27] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. 2015. Detection of early-stage enterprise infection by mining large-scale log data. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 45–56.
[28] Karthik Pattabiraman, Giancinto Paolo Saggese, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2011. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing* 8, 5 (2011), 640–655.
[29] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 2. IEEE, 169–178.
[30] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523.
[31] Hinrich Schütze. 2008. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*.
[32] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E Hassan, and Patrick Martin. 2013. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 402–411.
[33] Weiyi Shang, Meiyappan Nagappan, and Ahmed E Hassan. 2015. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering* 20, 1 (2015), 1–27.
[34] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2011. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 300–310.
[35] Stanford. 2008. Evaluation of Clustering, NMI. https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html. [Online; accessed July-2018].
[36] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on information and knowledge management*. ACM, 785–794.
[37] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *IP Operations & Management, 2003.(IPOM 2003). 3rd IEEE Workshop on*. IEEE, 119–126.
[38] Wikipedia. 2018. Complete linkage clustering. https://en.wikipedia.org/wiki/Complete-linkage_clustering. [Online; accessed July-2018].
[39] Wikipedia. 2018. Multivariate normal distribution. https://en.wikipedia.org/wiki/Multivariate_normal_distribution. [Online; accessed July-2018].
[40] Wikipedia. 2018. Sigmoid Function. https://en.wikipedia.org/wiki/Sigmoid_function. [Online; accessed July-2018].
[41] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 117–132.
[42] Li Xuan, Chen Zhigang, and Yang Fan. 2013. Exploring of clustering algorithm on class-imbalanced data. In *2013 8th International Conference on Computer Science Education*. 89–93.
[43] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. 2006. Automated known problem diagnosis with event traces. In *ACM SIGOPS Operating Systems Review*, Vol. 40. ACM, 375–388.
[44] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael Mihn-Jong Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging.. In *OSDI*, Vol. 12. 293–306.
[45] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 415–425.