LPB-Gen: Systematic Large Log-Parsing Benchmarks Generation

HETONG DAI, Department of Electrical and Computer Engineering, University of Waterloo, Canada
KUNDI YAO, Department of Electrical and Computer Engineering, University of Waterloo, Canada
FELIX LI, Department of Electrical and Computer Engineering, University of Waterloo, Canada
JIANXIN YOU, Department of Computer Science and Operational Research, University of Montreal, Canada
QIANYUN SHEN, Department of Computer Science and Operational Research, University of Montreal, Canada
WEIYI SHANG, Department of Electrical and Computer Engineering, University of Waterloo, Canada

Log parsing is a critical step in analyzing software logs by transforming unstructured raw logs into structured formats, which are essential for system monitoring, failure debugging, and anomaly detection purposes. Creating and maintaining log parsing rules for specific systems requires both domain expertise and significant time and manual effort. To address this challenge, various automated log parsers have been proposed to alleviate the parsing efforts and enhance parsing efficiency. These approaches commonly evaluate their parsing performance using off-the-shelf benchmarks (i.e., Loghub-2k and Loghub-large) that comprise manually labeled logs (i.e., oracle templates) collected from a limited number of datasets. However, such benchmarks are not designed to be extended or adapted to evaluate logs from different sources, making them less applicable to real-world scenarios. On the other hand, crafting new benchmarks requires substantial domain expertise and manual effort, rendering generating log-parsing benchmark datasets an extremely challenging task. Consequently, there is a growing need for a standardized process to assist practitioners in efficiently generating high-quality log parsing benchmarks.

In this paper, we propose LPB-Gen, a semi-automated process designed to facilitate the log parsing benchmark datasets generation for practitioners. We present a comprehensive evaluation of LPB-Gen through a user-based assessment to evaluate the correctness of the log benchmark data generated by LPB-Gen as well as the efficiency and reproducibility of LPB-Gen. The user study results not only show that LPB-Gen can generate a benchmark that outperforms the state-of-the-art benchmark in both coverage and accuracy, but also prove that LPB-Gen can provide actionable solutions that apply to generalized datasets, where practitioners may leverage our semi-automated process to produce oracle templates based on their specific datasets. Finally, we summarize seven factors that contribute to inconsistencies during developers' manual refining of logs. These factors shall provide valuable insights for future research aimed at advancing log parsing and analysis.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Log parsing, Software logs, Log parsing benchmark

Authors' Contact Information: Hetong Dai, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada, h5dai@uwaterloo.ca; Kundi Yao, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada, kundi.yao@uwaterloo.ca; Felix Li, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada, f23li@uwaterloo.ca; Jianxin You, Department of Computer Science and Operational Research, University of Montreal, Montreal, Canada, jianxin.you@umontreal.ca; Qianyun Shen, Department of Computer Science and Operational Research, University of Montreal, Montreal, Canada, qianyun.shen@umontreal.ca; Weiyi Shang, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada, wshang@uwaterloo.ca.

Manuscript submitted to ACM

ACM Reference Format:
Hetong Dai, Kundi Yao, Felix Li, Jianxin You, Qianyun Shen, and Weiyi Shang. 2018. LPB-Gen: Systematic Large Log-Parsing Benchmarks Generation. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 32 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Log data is an essential source of system runtime information that supports various software engineering activities. In practice, it provides developers with valuable insights into the runtime behaviors of software systems and assists in diagnosing failures [6, 11]. Since log files are typically large and complex, automated log analysis techniques have been proposed to facilitate tasks such as anomaly detection [18, 25, 83], failure diagnosis [17, 65], and system comprehension [17, 61], thereby improving software quality and reliability.

Log data typically consists of unstructured text produced from heterogeneous sources and represented in different formats. Since automated log analysis approaches usually rely on structured inputs, log parsing, which transfers unstructured text logs into a structured format, usually serves as the prerequisite step of log analysis. As shown in Figure 1, the raw log is generated by executing the logging statement in the source code. Taking the raw log as an input, the initial step of log parsing is to separate each line of logs into two parts, i.e., header information (e.g., datetime, log level) and content (`Block not found: blk_12`). Log parsing identifies the static text that is written by developers and dynamic variables that are produced by dynamic information generators (e.g., variable or method invocation). Accurate log parsing still faces several challenges. For instance, the source code and logging configurations shown in the left part of Figure 1 are often unavailable in practice due to issues such as security restrictions and the use of external libraries. Moreover, differences in practitioners' preferences and experiences can influence the manual refinement process illustrated in the right part, ultimately affecting the quality and consistency of the oracle templates.

Our semi-automated process can reduce the impact of problems related to practitioners' experience in the manual refinement process. Moreover, our approach does not rely on source code to generate accurate oracle templates for benchmark data. The benchmark data generated by our approach shows a high consistency with benchmark data extracted from source code (over 0.95).

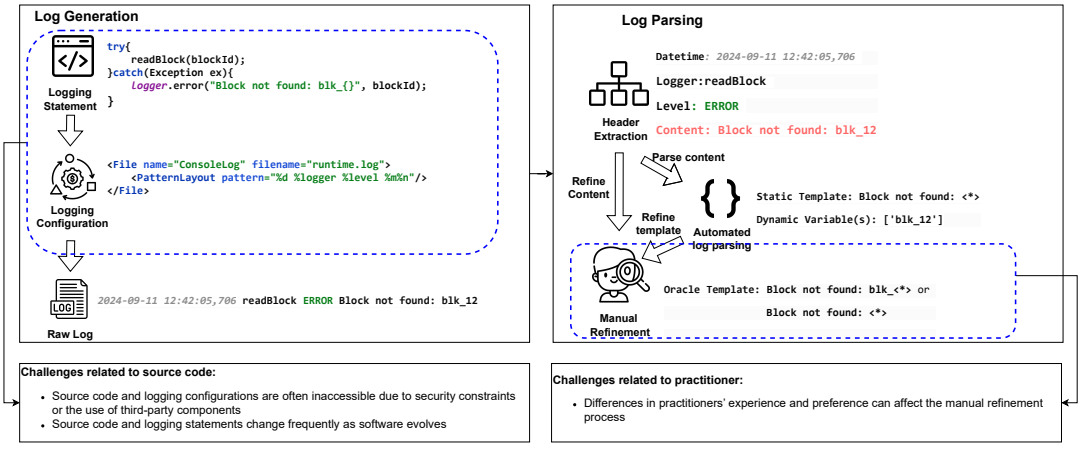


Fig. 1. An illustrative example of log generation and log parsing process.

In practice, developers usually utilize ad hoc rules like specially designed regular expressions for log parsing [23, 32, 97]. However, given the heterogeneous nature of logs and the evolving nature of log data as software updates, developing and maintaining such rules can be a complex task. To solve the problems of developing and maintaining ad hoc rules, prior research proposes various automated log parsers like Drain [24], Hue [81], and PILAR [14].

Although a large number of log parsers (e.g., ULP [60], DivLog [82], NuLog [53]) have been proposed in recent years, their direct evaluation remains largely confined to a relatively small-scale benchmark, namely Loghub-2k [97]. The version most frequently used in prior research consists of 16 log datasets, each containing only 2,000 log lines with manually created parsing results.

The limited availability of comprehensive benchmarks has become a major obstacle to further advancements in log parsing and analysis research. While some parsers are evaluated on self-constructed datasets, these datasets are still derived from widely used systems in Loghub-2k [97], such as BGL, HDFS, and OpenStack. Consequently, the evaluation of log parsers remains confined to similar environments, restricting the diversity and representativeness of assessment scenarios.

More recently, an updated version of Loghub, referred to as Loghub-large [96], was released, containing significantly more labeled log data. Nevertheless, practitioners still face challenges in directly adopting off-the-shelf automated log parsers solely based on benchmark evaluations. Prior research has shown that parser performance often varies substantially across datasets [14], meaning practitioners must first assess existing parsers on their own log data before deployment. Moreover, no existing parser achieves near-perfect accuracy [97]. In practice, customized pre- and post-processing steps are often necessary to refine automatically generated parsing results. However, both of these tasks fundamentally depend on the availability of a human-curated log parsing oracle for the specific log data under analysis.

Despite the necessity of a larger and more diverse log parsing benchmark for both research and practice, constructing such benchmarks can be a challenging task. First, the large size of logs renders manually curating oracle templates an impractical practice. Besides, practitioners' perceptions of defining log templates and dynamic variables may vary, and biased judgment can lead to discrepancies among different practitioners, thus impairing the trustworthiness and consistency of the benchmark. Although Loghub-large (namely Loghub in the rest of the paper for simplicity) keeps extending its labeled data [96], Loghub still experiences the same challenges aforementioned as 1) it only serves as a benchmark log dataset, which lacks providing an actionable approach to extending or creating new benchmarks, 2) the huge labeling efforts render logs partially labeled in large-sized datasets (only 51.1% logs labeled in Thunderbird dataset), and 3) no human feedback reported to assess the correctness of the benchmark log datasets. In fact, to the best of our knowledge, there is **no standard process that practitioners can follow to generate high-quality benchmarks for their specific log datasets**, and we argue that such a process would benefit both research and practice on software logs over the existing log benchmark datasets.

In this paper, we design a semi-automated process, LPB-Gen. LPB-Gen leverages state-of-the-art log parsers to cross-reference the clustering results to group similar logs into the same cluster. This process reduces the overwhelming number of logs to be reviewed into a single log per cluster, allowing practitioners to efficiently define oracle templates based on their expertise and perception. Furthermore, LPB-Gen adopts an iterative process by applying itself to a small sample of full log data in each iteration to avoid runtime overhead. Practitioners may adopt LPB-Gen to construct new benchmarks or extend existing benchmarks based on their specific log datasets.

To assess the practicality of adopting this process in real-world scenarios, we conducted user studies with LPB-Gen focusing on two key aspects. First, we examined the accuracy of the benchmark datasets generated by LPB-Gen. Specifically, we study: 1) the ability to group logs with different templates correctly, and 2) the ability to identify the

static text and dynamic information correctly. The findings indicate that LPB-Gen produces highly accurate benchmark data for log parsing evaluation, with both grouping and identification performance surpassing those of Loghub-large. Second, we evaluated the consistency and efficiency of LPB-Gen. The results demonstrate that LPB-Gen achieves stable grouping consistency for benchmark generation while maintaining efficiency. Evaluation also indicates that grouping consistency is insensitive to sample size, so practitioners may select an appropriate sample size freely. Finally, we summarize several factors that may lead to inconsistencies when developers manually curate log parsing data. These findings offer valuable insights for future research on improving the reliability and automation of log parsing.

In summary, our paper makes four major contributions.

- We propose a semi-automated process, LPB-Gen, which can enable practitioners to generate log parsing benchmark datasets for system logs with relatively stable formats.
- We conduct a user-based assessment to evaluate the quality of log benchmark datasets generated by LPB-Gen. The same assessment approach can be adopted for evaluating other log parsing benchmarks.
- We conduct a user study to evaluate the effectiveness of the LPB-Gen process and investigate the common pitfalls during template labeling. The findings shed light on further research directions on log parsing.
- We release our log benchmark datasets for future research and potential extension of this work for evaluation and replication purposes.

Paper organization. Section 2 introduces the background and related work of LPB-Gen. Section 3 presents our semi-automated process for generating oracle templates. Section 4 conducts comprehensive user studies to evaluate both the benchmark data and the process of LPB-Gen. Section 5 discusses the threats to the validity, and Section 6 concludes our paper.

2 Background and Related Work

In this section, we present the background and related work of LPB-Gen.

2.1 Log parsing

In general, log parsing is the process of transforming logs from unstructured text to a structured format. The main goal of this process is to identify the static information written by the developers as well as the dynamic information generated during runtime. An automated log parser typically contains two steps. In the first step, header information, such as the time stamp and log verbosity levels, is automatically identified. Such header information usually follows pre-defined formats. Often, the header format can be found in the configuration of various logging libraries [88, 94], making extracting header information less challenging. However, if the format of the headers is not readily available, this step may still be a challenging step for automated parsers. In the second step, after removing the header information, the remaining parts in the logs are typically referred to as log content. The main challenge of log parsing is to identify static information and dynamic information in log content.

Log parsers can be classified into three categories: heuristic rules, code-driven, and data-driven approaches. Heuristic rules based parsers, which parse logs with hand-crafted rules [30], usually require significant efforts in maintaining such rules as the software evolves. Code-driven log parsers leverage source code to assist in identifying the static information from raw logs [67, 84, 89]. While code-driven log parsers may achieve high accuracy, they are usually limited by the program complexity and source code accessibility [92].

A data-driven based log parser is proposed to solve the problem caused by the former two types of log parsers. This type of log parsing mainly consists of two kinds of parsing strategies: 1) cluster-based and 2) token-based. Drain [97] is an exemplary cluster-based strategy that represents logs with a fixed-depth prefix tree, where each branch represents log lines from the same template. Drain calculates the similarity between each branch and log lines to cluster the log lines with the same template together. It is needed to mention that log lines are fed to parsers in a streaming manner. Finally, the static information and dynamic variables are identified based on the clusters. PILAR [14] is an exemplar token-based strategy. Unlike cluster-based strategies, PILAR calculates the entropy on the position of each token, based on which PILAR identifies whether the token is static information or dynamic variable.

With the popularity of large language model (LLM), many LLM-based log parsers have been proposed. LILAC [29], ULog [26], and LogBatcher [80] are the representatives of this kind of log parsers. This kind of log parser leverages the semantics in logs through fine-tuning LLMs or learning from in-context demonstrations.

In this work, we leverage existing automated log parsers, e.g., Drain and PILAR, with LPB-Gen in generating oracle templates.

2.2 Log parsing evaluation

Various log parsers have been proposed and evaluated in prior research. We conduct a literature review on prior proposed log parsers and their corresponding evaluation datasets and metrics. Tables 1, 2, 3 present a summary of our literature review.

GA (Group Accuracy), PA (Parsing Accuracy), FGA (F1 Group Accuracy), and FPA (F1 Parsing Accuracy) are among the most widely used direct evaluation metrics for assessing log parsers. These metrics have been adopted in the Loghub-large benchmark. In contrast, earlier studies generally lacked such direct evaluation measures on parsing results and instead relied on Downstream (downstream-related metrics) to indirectly assess log parser effectiveness.

Prior research on log parsing has typically relied on limited datasets, most of which were generated by the researchers themselves and lack ground truth for directly evaluating log parsers. As a result, earlier studies often assessed performance through downstream tasks or case studies. To address this limitation, Drain [24] introduced five datasets with ground truth, BGL, HPC, HDFS, Zookeeper, and Proxifier, to enable the calculation of group accuracy (GA). These datasets have since been widely adopted in subsequent research on log parsing.

Loghub-2k [97] is the first log parsing benchmark comprising 16 datasets. The benchmark samples 2,000 logs from each dataset and manually extracts the templates for these logs. The evaluation of log parsing techniques is conducted on the 2,000 logs sampled from the dataset. The evaluation of the accuracy in log parsing benchmarks is based on group accuracy, which measures whether the logs with the same template are clustered together. Logram [13] also utilizes the 2,000 logs in 16 datasets to evaluate their log parsing technique. Different from group accuracy utilized in the original benchmark, the authors propose parsing accuracy, which focuses on whether the dynamic variable and static information are identified correctly. Logram also proposes the necessity to evaluate the efficiency of the log parsing benchmark. Ma et al. proposed a benchmark to evaluate the ability of LLMs to parse logs [46]. Similar to Loghub-2k, this benchmark randomly samples 2,000 logs for evaluation. The benchmark evaluates both group accuracy and parsing accuracy of a log parsing technique.

Though the 2,000 logs are widely used in log parsing technique evaluation, it is impossible to evaluate the actual performance of log parsing techniques in practical scenarios, as the volume of logs in practice is much larger than 2,000. Recently, Zhu et al. [96] updated Loghub-2k to Loghub-large (Loghub) by incorporating significantly more labeled data. However, both Loghub-2k and Loghub-large are labeled based on the same data sources (logs collected in 16 software

Table 1. A summary of existing log parsing techniques and their evaluation

Log parser	Publication	Dataset	Metrics
LogMine [21]	2016	LogMine-6-dataset [21], Pylogabstract-5-datasets [64], Loghub-2k [97], Loghub-large [96]	GA, PA, FPA, FGA
LogCluster [41]	2016	Hadoop [41], Microsoft Service [41], log2timeline [71], Loghub-2k [97], Loghub-large [96]	Downstream, Clustering performance, GA, PA, FPA, FGA
LPV [79]	2020	BGL [97], HPC [97], HDFS [97]	GA, Precision, Recall, F1
LogTree [68]	2010	FileZilla, MySQL, PVFS2, Apache [68]	F1
LogOHC [86]	2019	BGL [97], HPC [97], HDFS [97]	Precision, Recall, F1
METING [12]	2020	Loghub-2k [97]	GA
STE [35]	2014	STE-dataset [35], Dlog-1-dataset [39], FT-Tree-1-dataset [91], Craftsman-1-dataset [90]	GA, PA, Group rand index
LTE [85]	2015	OpenSSH(1746 lines) [85]	Template words accuracy, recall
NER [56]	2023	Windows-6-datasets [56]	GA, Variable accuracy
UiLog [99]	2016	CentOS [99]	Downstream
HLAer [55]	2014	LogMine-6-dataset [21]	GA, PA, FPA, FGA
LenMa [62]	2016	LenMa-3-dataset [62], Loghub-2k [97], Loghub-large [96]	Downstream, GA, PA, FPA, FGA
StringMatch [4]	2009	StringMatch-4-datasets [4], FileZilla [69], Thunderbird [69], PVFS2 [69], Apache [69], Hadoop [69]	Downstream, F1
SLCT [73]	2003	Linux [73], Solaris [73], HPC [47], Windows [47, 73], Apache [47], OsX [47], OpenBSD [47], LFA-1-dataset [51], DB2 [100], Squid [100], CPlant [63], Baler-2-datasets [66], AEL-1-dataset [31], Loghub-2k [97], Loghub-large [96]	Downstream, Case study, GA, PA, FPA, FGA, Precision, Recall, F1
LKE [19]	2009	Hadoop [19], SILK [19], BGL [16, 24], HPC [16, 24], HDFS [24], Zookeeper [24], Proxifier [24], Loghub-2k [97], Loghub-large [96]	Downstream, GA, PA, FPA, FGA, F1
IPLoM [47]	2011	Public Security Log [1], HPC [2, 16, 24, 47], Windows [47], Apache [47, 69], OsX [47], OpenBSD [47], BGL [2, 10, 16, 24], HDFS [2, 10, 24], Zookeeper [2, 24], Proxifier [24], Thunderbird [69], OpenStack [10], Spark [2], Pylogabstract-5-datasets [64], Huapu [45], log2timeline [71], Baler-2-datasets [66], Loghub-2k [97], Loghub-large [96]	Case study, avgTokensLost, Clustering performance, GA, PA, FPA, FGA, Precision, Recall, F1
LogSig [69]	2011	FileZilla [69], Thunderbird [69], PVFS2 [69], Apache [69], Hadoop [69], Pylogabstract-5-datasets [64], Loghub-2k [97], Loghub-large [96]	GA, PA, FPA, FGA, F1
SHISO [50]	2013	Public Security Log [1], BGL [2, 24], HPC [2, 24], HDFS [2, 24], Zookeeper [2, 24], Proxifier [24], Spark [2], Presto [2], Loghub-2k [97], Loghub-large [96]	avgTokensLost, Case study, GA, PA, FPA, FGA, F1
FDiag [9]	2010	Ranger [9], Turing [9], BGL [9]	Downstream

Table 2. A summary of existing log parsing techniques and their evaluation 2

Log parser	Publication	Dataset	Metrics
LogHound [74]	2004	LogHound-8-datasets [74], DB2 [100], Squid [100], Baler-2-datasets [66]	Downstream, Case study
LFA [51]	2010	LFA-1-dataset [51], Loghub-2k [97], Loghub-large [96]	Case study, GA, PA, FPA, FGA
ENG [72]	2019	BGL [24], HPC [24], HDFS [24], Zookeeper [24], Proxifier [24]	F1
SignatureTree [57]	2010	SignatureTree-2-datasets [57], FT-Tree-1-dataset [91], Craftsman-1-dataset [90]	Downstream, Group rand index
Dlog [39]	2018	Dlog-1-dataset [39]	PA
FT-tree [91]	2017	FT-Tree-1-dataset [91], Loghub-large [96]	Group rand index
Craftsman [90]	2020	Craftsman-1-dataset [90]	Group rand index
Prefix-Graph [8]	2021	Loghub-large [96]	Group rand index
Capri [100]	2013	DB2 [100], Squid [100]	Downstream
Logram [13]	2020	Loghub-2k [97], Loghub-large [96]	GA, PA, FPA, FGA
Liu et al. [45]	2020	Huapu [45]	GA, Recall, F1
Stearley et al. [63]	2004	CPlant [63]	Case study
SwissLog [40]	2020	Loghub-large [96]	GA
Delog [2]	2019	BGL [2], HPC [2], Zookeeper [2], HDFS [2], Spark [2], Presto [2]	avgTokensLost
Logan [3]	2019	BGL [2], HPC [2], Zookeeper [2], HDFS [2], Spark [2], Presto [2]	avgTokensLost
LTMatch [76]	2021	Loghub-2k [97]	GA
OLMPT [77]	2020	Loghub-large [96]	GA
USTEP [75]	2021	Loghub-large [96]	GA
AECID-PG [78]	2019	BGL [24], HPC [24], HDFS [24], Zookeeper [24], Proxifier [24]	GA
NLP-LP [5]	2018	HDFS [24]	GA
Li et al. [38]	2005	logdump2td [38]	Downstream
Kobayashi [36]	2014	Kobayashi-3-datasets [36]	Word accuracy, Line accuracy, Template accuracy
FastLogSim [44]	2020	HDFS [24], Zookeeper [24], Proxifier [24], OpenStack [96]	GA
NuLog [52]	2020	Loghub-2k [97], Loghub-large [96]	GA, PA, FPA, FGA
LogParse [48]	2020	Loghub-2k [97]	GA
Thaler et al. [71]	2017	log2timeline [71]	Clustering performance
Lopper [33]	2018	BGL [24], HPC [24], HDFS [24], Zookeeper [24], Proxifier [24]	GA
CLF [42]	2018	Loghub-2k [97]	GA

systems), without the flexibility to extend the benchmark or possess the capability to support log parsing evaluation for

Table 3. A summary of existing log parsing techniques and their evaluation 3

Log parser	Publication	Dataset	Metrics
POP [22]	2018	BGL [24], HPC [24], HDFS [24], Zookeeper [24], Proxifier [24]	GA
Paddy [27]	2020	Loghub-large [96]	GA, PA,FPA, FGA
AEL [31]	2008	AEL-1-dataset [31], Loghub-2k [97], Loghub-large [96]	Precision, Recall, GA, PA, FPA, FGA
Zhu et al. [98]	2010	LEARNPADS [98]	Downstream
Baler [66]	2011	Baler-2-datasets [66]	Case study
MoLFI [49]	2018	BGL [24], HPC [24], HDFS [24], Zookeeper [24], Proxifier [24], Loghub-2k [97], Loghub-large [96]	GA, PA,FPA, FGA
BSG [20]	2018	BGL [24], HDFS [24], Thunderbird [69], Apache [69]	F1
Zhao et al. [93]	2018	Windows [93]	Downstream
FLP [95]	2018	Zookeeper [24], Proxifier [24], BGL [24], HDFS [24], HPC [24]	F1
One-to-one [10]	2020	BGL [10], HDFS [10], OpenStack [10]	GA, Recall, F1
Pylogabstract [64]	2020	Pylogabstract-5-datasets [64]	F1
Spell [16]	2016	BGL [2, 10, 16, 24], HPC [2, 16, 24], HDFS [2, 10], OpenStack [10], Zookeeper [2], Spark [2], Presto [2], Pylogabstract-5-datasets [64], Huapu [45], Loghub-2k [97], Loghub-large [96]	avgTokensLost, GA, PA, FPA, FGA
Drain [24]	2017	BGL [2, 10, 24], HPC [2, 24], HDFS [2, 10, 24], Zookeeper [2, 24], Proxifier [24], OpenStack [10], Spark [2], Presto [2], Pylogabstract-5-datasets [64], Huapu [45], Loghub-2k [97], Loghub-large [96]	avgTokensLost, GA, PA, FPA, FGA, F1
Rand et al. [58]	2021	Rand-1-dataset [58]	Downstream
FlexParser [59]	2022	Loghub-large [96]	F1
LogDTL [54]	2021	LogDTL-2-datasets [54]	Word Accuracy, Line Accuracy, Template Accuracy
LogStamp [70]	2022	Loghub-2k [97]	GA

developers' specific logs. Furthermore, although Loghub-large increased the number of oracle templates to expand the benchmark, not all the logs in the dataset can be covered by the provided oracle templates.

We also note that although some log parsers, such as Spell [16] and One-to-one [10], employ their own datasets for direct evaluation, these datasets are still derived from widely used systems such as BGL, HDFS, and OpenStack. As a result, the evaluation of log parsers remains constrained to similar environments, thereby limiting the diversity of assessment scenarios.

Despite the increasing availability of benchmarks for log parsing, the datasets used for evaluation remain limited in scope. Direct evaluation can only be performed on a small set of benchmarks and systems, highlighting the absence of a mature approach for systematically producing such benchmarks. Developing such an approach would be highly beneficial in practice. In this work, we propose LPB-Gen, a semi-automated process to generate oracle templates that can be used to generate log parsing benchmarks for different datasets.

Existing log parsing studies rely heavily on a small number of publicly available datasets (e.g., LogHub) and self-constructed benchmarks that lack reproducibility and ground truth. Despite significant progress in parsing algorithms, there remains no systematic and scalable approach for generating reliable log-parsing benchmarks. This gap motivates the development of a framework to enable reproducible and human-verified benchmark generation.

3 LPB-Gen: A systematic approach to building log-parsing benchmarks

In this section, we present our semi-automated process for generating oracle templates for raw log datasets with extra-large sizes. The main idea of the pipeline is to use log parsers to assist in generating clustering results and refining the representative logs. The pipeline is composed of six steps: 1) Extracting log content from raw logs, 2) Generating log samples, 3) Generating initial parsing results with log parsers, 4) Aligning log parsing results, 5) Refining log templates, and 6) Matching datasets with refined templates. Figure 2 shows the entire pipeline of LPB-Gen's semi-automated process. LPB-Gen enables practitioners to produce oracle templates for different datasets based on their specific requirements. For evaluation, we choose the datasets from Loghub-large to generate the oracle templates as examples in this paper.

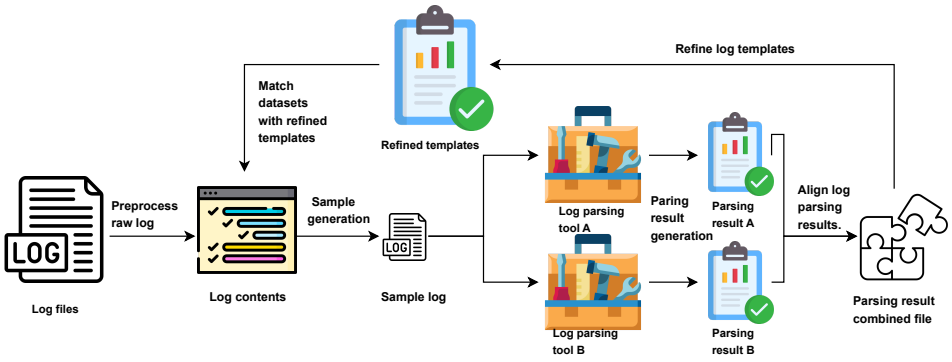


Fig. 2. A semi-automated process for generating log-parsing benchmarks.

Step 1: Extracting log content from raw logs. As shown in Figure 1, logs from the same dataset usually follow the same logging format that is defined in the configuration. In this step, we utilize regular expressions to extract the log content part from log lines. Our semi-automated process generates oracle templates for log contents extracted from log lines. In the following example, the highlighted text in the box is the log content to be extracted, and it is used as input for log parsers.

```
Jul 1 09:01:43 calvisitor-10-105-160-95 kernel[0]: PM response took 1990 ms (54, powerd)
Jul 1 09:01:00 — last message repeated 1 time —
```

Figure 3 shows a running example of the preprocess step with simplified example logs from the Thunderbird dataset.

There exist multiple log formats inside Thunderbird dataset. In the preprocess step, we utilize heuristic rules to find the corresponding format for log lines. In the running example, the first 3 logs utilize the first log format for preprocess, and the last 3 logs utilize the second log format for preprocess. Based on the format, the preprocess step will generate a

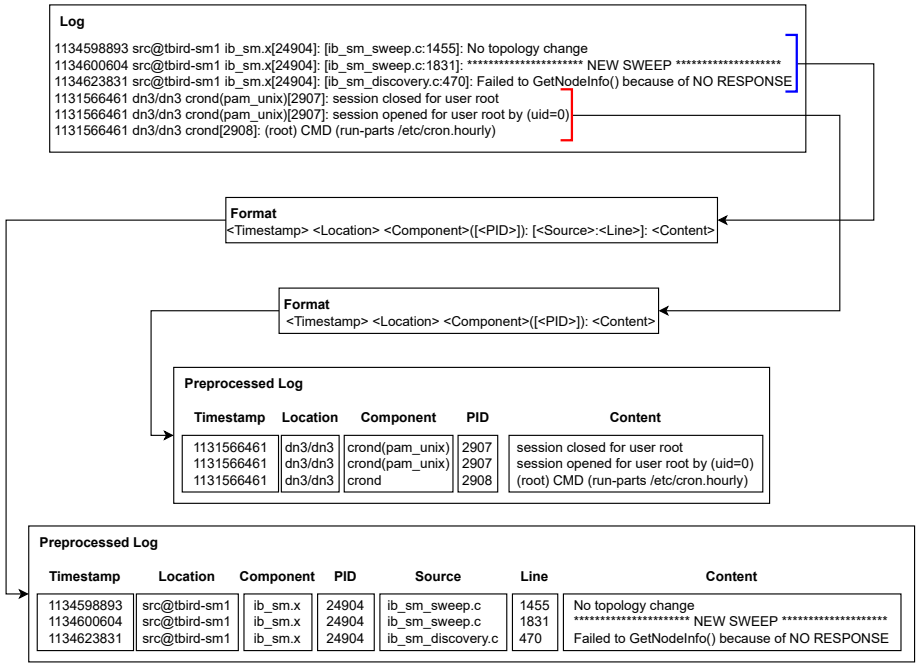


Fig. 3. A running example for preprocess step

heuristic regular expression and extract the header information (e.g., Timestamp and Component) and log content with it. In total, we summarize 4 log formats that can be applied to the Thunderbird datasets.

Prior work, such as LogHub, uses one log format for one dataset to extract header information and log content. This approach can only work on most existing log datasets, as the logs in these datasets have only one log format exactly. We also apply one log format for these kinds of datasets. We also conduct a user study to discuss this in Section 4.3.2.

In practice, there exist logs whose content part is omitted. The following logs from Thunderbird are examples that are without log content. Since logs without content would not provide much useful information, we only use logs with content to produce benchmarks. If the content of a log line does not exist, we leave out this log line from parsing.

```
- 1137433662 2006.01.16 dn587 Jan 16 09:47:42 dn587/dn587 ts_fixup:
- 1138295471 2006.01.26 bn446 Jan 26 09:11:11 bn446/bn446 kernel:
```

Step 2: Generating log samples. As the size of log data increases, the parsing time needed for a parser can increase significantly [13]. On the other hand, a small number of templates may already cover a large portion of the entire log data [43]. Therefore, we first sample a decent number of log contents (e.g., 20,000) randomly from the processed datasets after Step 1. If the number of log contents in a dataset is less than this number, we directly utilize the entire dataset. It is worth noting that we do not explicitly remove duplicate log lines in our framework, as duplication naturally occurs in real-world log data and reflects realistic operational workloads. To mitigate the potential impact of overrepresentation of frequent log templates during sampling, we adopt a stratified sampling strategy. Specifically, the dataset is divided into n equal segments, and one log line is randomly selected from each segment. This approach ensures that the sampled

Table 4. Overlapping ratio of log parsers with Drain

AEL	Brain	IPLoM	LenMa	PILAR	Spell
0.834	0.809	0.825	0.790	0.726	0.758

subset is evenly distributed across the entire dataset, thereby preserving diversity and reducing the likelihood that localized duplicates dominate the sampled data. As log data exhibits a locally repetitive nature [87], adjacent logs are likely to have similar contents, and sampling consecutive log lines may lead to fewer diverse templates included in the log sample. To ensure that the distribution of sampled logs is even among the whole dataset, we first distribute the entire dataset into n pieces equally (n is the number of log lines in the sample file). Then, we pick one line of logs from every chunk of log data randomly and align these logs to get a sample of logs.

Step 3: Generating initial parsing results with log parsers. Although automated log parsers cannot produce perfect log parsing results [97], their results are still valuable and can serve as a starting point to produce the log benchmark data. Studies show that different log parsers may have different results on different datasets [97]. In order to have more diverse log parsing results from the automated log parsers as starting points, we select log parsers from both kinds (i.e., cluster-based and token-based, as presented in Section 2.1) in our process.

We employ multiple log parsers to assist in the initial generation of parsing results. In this paper, we select Drain [24] and PILAR [14] for the following reasons.

Representativeness: Drain represents logs using a fixed-depth tree structure and clusters log lines sharing the same template based on their textual similarities. PILAR, on the other hand, estimates the entropy of each token position to determine whether a token corresponds to static or dynamic information. Both parsers have demonstrated strong performance on log parsing tasks, achieving average group accuracies of 0.88 and 0.85 on the Loghub-2k datasets, respectively [14]. We select these two parsers as representative examples of cluster-based and token-based approaches.

Complementary: We conducted an evaluation of the overlapping ratios among different log parsers to further validate this selection. Table 4 shows that PILAR exhibits the least overlap with Drain compared to other parser combinations, with an overlapping ratio of 0.726. This indicates that PILAR and Drain produce the most diverse parsing results among the parsers we examined, which we believe provides better complementation when reconciling their outputs to derive oracle templates.

Framework Flexibility: We acknowledge that practitioners may wish to incorporate additional parsers or substitute different ones based on their specific needs. Our framework is built with extensibility and modular design in mind, and can be readily adapted to support additional or alternative parser combinations.

Each parser generates two output files following the same format. 1) An event file contains the original logs and log templates, where each row in the event file consists of a log content and its corresponding log template. 2) A template file contains all log templates and their occurrences, where each row in the template file consists of a unique log template and the number of log lines that belong to this template. We leverage the output of the automated log parsers to assist in generating the oracle templates.

Step 4: Aligning log parsing results. It is impractical to define templates manually for all logs in a dataset, as the size of log data is usually too large. Therefore, prior research like Loghub-large clusters similar logs together and selects representative logs from the clusters for defining templates manually [96]. The clustering strategy of Loghub-large is based on top-k frequent tokens, which may lead to noisy data in the cluster. The noisy data can further produce

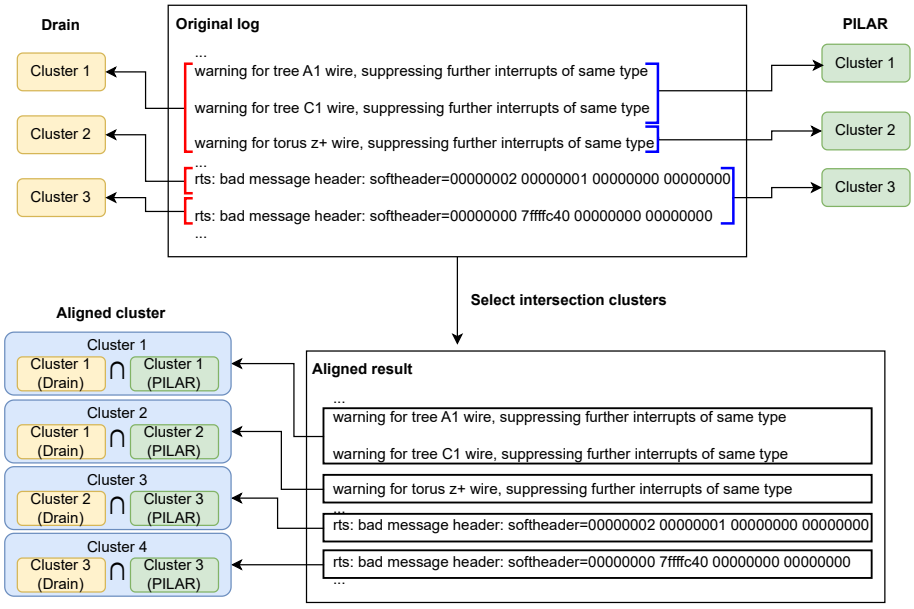


Fig. 4. A running example for our aligned strategy

misleading information for practitioners, which can increase labeling effort and complexity. To overcome this, we combine parsing results from two log parsers, aligning their results to identify the minimal overlap between clusters. This alignment allows for a more precise clustering, filtering out noise and improving the overall quality of clustering logs having the same template.

Our aligned strategy utilizes the clustering result produced by the two log parsers to generate a more precise clustering result, which will be used to extract representative logs. To avoid a cluster containing logs from different templates, we select the intersection of clusters from multiple log parsers as a minimum cluster. In other words, we choose the clusters that are agreed upon by different log parsers. The reason that we choose the intersection of the clusters is that though one single parser can have mistakes in clustering results, we can mitigate these mistakes by cross-referencing clustering results with another parser. The logs in the intersect clusters are considered to have the same log template by all of the log parsers. Though logs with the same template may sometimes be split into different clusters when using the intersect, the minimum cluster aims to enhance the similarity within each cluster and minimize noise. This approach enables LPB-Gen to produce clean and more consistent log clusters for practitioners to start working on, improving the overall accuracy of the labeling step. We would like to note that, due to the choice of using intersects between log parsers for alignment, this step may generate repetitive templates. These repetitive templates will be removed in the next step of our process (refining step).

Figure 4 illustrates a running example of our alignment strategy. The running example illustrates how the aligned file is generated based on the differing clustering results from Drain and PILAR. The clusters on the left represent the results from Drain, while those on the right correspond to PILAR. For the four log lines shown, the two parsers differ in how they group the last log line: Drain places the last two logs in the same cluster, whereas PILAR separates them into two distinct clusters. We take the intersection of clusters produced by multiple log parsers as the minimum cluster. As

shown in figure 4, the aligned cluster represents this minimum cluster, which is used for representative log extraction. A representative log is then randomly selected from each cluster for the manual refinement process.

The aligned file consists of the following information: 1) different log templates generated from different log parsers (Drain and PILAR in our case of evaluation, while other parsers can be included without changing our process), and 2) the log lines. Each row in the aligned file includes one line of the log, which is randomly selected from the intersection clusters and its corresponding templates from the automated parsers (Drain and PILAR). Noted that we sample one log line per cluster by default to minimize the labeling effort for practitioners. Our pipeline also enables one to choose to sample multiple log lines per cluster to capture greater variability and further enhance the template refining accuracy.

Step 5: Refining log templates. LPB-Gen adopts a two-step approach to refining templates, namely: 1) manual refining and 2) automatic refining. In the first part, we refine the templates for representative logs manually. In the second part, an automated approach is utilized to remove duplicated templates and re-sort the sequence of log templates.

Manual refining log templates: Based on the aligned file that is generated in the prior step, we perform a manual refining process to derive oracle templates based on sampled log lines. For each row in the aligned file, we leverage the parsing results as references to assist in defining the template of the representative log. Specifically, we preserve the static parts of the log line and replace dynamic parts with a wildcard (i.e., <*>). By labeling directly on a log line, we ensure that errors caused by human factors (e.g., typos) are minimized. The template extracted from the representative log is treated as the template for the logs in the cluster.

Although log templates are aligned with corresponding log lines, the parsing result serves as a reference for practitioners during the manual refining process. A major reason that templates generated by log parsers cannot be leveraged directly is because of wrong tokenization. Furthermore, log parsers may fail to derive meaningful or informative templates from broken logs and special logs. We discuss these special cases that we encountered during the manual refining process as follows.

Wrong tokenization: In practice, developers usually include symbols in logging (e.g., =, &), such as placeholders like *user* = $*username*$, for easier tracking and debugging system activities. However, most log parsers only rely on white spaces for tokenization, which can result in valuable information being overlooked in parsed templates. In the former example, the placeholder text is treated as a single token, and the log parser identifies the tokens as a whole variable (<*>) rather than recognizing the individual static text with a corresponding dynamic variable (user=<*>). Improper handling of cases like the separation of meaningful variables and overlooking symbols will potentially impair the accuracy of log parsing and the effectiveness of log analysis. To address this issue, we compare each log template generated by log parsers with its corresponding raw logs and repair the log templates generated by log parsers.

Broken logs and special logs: In some datasets, there exist some special logs with non-informative or incomplete information. We observe logs that only contain dynamic information in their content parts, and some logs contain special cases that do not provide meaningful information for parsing. Meanwhile, some logs contain incomplete information in their content parts because there exist some bugs when the logs are generated by their logging statements. The following logs from the Thunderbird dataset are examples of broken logs and logs with only dynamic information.

```
- 1148652114 2006.05.26 an827 May 26 07:01:54 an827/an827 kernel: <1ump iobuf at 00000101331d0780 :
- 1148590109 2006.05.25 bn74 May 25 13:48:29 bn74/bn74 00000101b9ab9e88 0000000000000046 000000351a0bcdb5 ffffff000000007a
```

For this kind of log, we remove them from the datasets and put them into a special log file to ensure that irrelevant or non-informative logs are excluded from the parsing process.

Once the manual refining is completed, we conduct a verification process to verify that each template can successfully match with its original corresponding logs. If any unmatched case is detected, the mismatching templates and corresponding log lines are prompted to practitioners to fix. The manual refining is marked completed until no mismatching case is observed, ensuring that all templates are valid and error-free.

Automatic refining log templates: After refining the templates manually, many repeated templates may exist inside the refined file. There may also exist templates that are contained by other templates, this can lead to an overmatch problem. For example, there are template A (read block: <*> locally) and template B (read block <*>) and we can find that logs can be matched by template A (*Logs_A*) can also be matched by template B (*Logs_B*). If we place template B ahead of template A when generating log benchmark data, both *Logs_A* and *Logs_B* will be matched by template B, and there will not be any logs of template A. Such a case is known as an overmatch in the log parsing process. In the automatic refining step, we eliminate the repeated templates and sort the log templates automatically. We put the position of template B after template A to make sure template A can be used in the dataset matching process.

Step 6: Matching datasets with refined templates. After obtaining the refined templates of the sample logs, we utilize the refined templates to match all log contents that are unmatched in the prior iteration. It needs to be mentioned that in the first iteration, the unmatched log contents are the whole dataset since none of the logs were matched before. Then, we output the log contents that are not matched by templates into a file. The unmatched file is treated as the log file in step 2, and we repeat the iteration until all logs in the target dataset can be matched with corresponding templates. This iterative process ensures both comprehensive coverage and efficient log labeling, as log templates generated from small log samples tend to match most logs within the first few iterations [43]. We aggregate templates from all iterations into one template file and conduct the automatically refined step to get an oracle template file for the whole dataset. Then we can utilize this oracle template file to match the whole dataset and produce the log parsing results as benchmarks.

4 Evaluation

In this section, we conduct comprehensive user studies to evaluate the correctness of the log benchmark data that is produced by LPB-Gen, as well as the efficiency and reproducibility of LPB-Gen.

4.1 User study on the correctness of benchmark data

The oracle templates in existing log parsing benchmarks (e.g., Loghub) are usually treated as ground truth and widely adopted in prior research in evaluating the performance and accuracy of log parsers. Due to the difficulties in analyzing source code to gain the ground truth of these datasets (e.g., unknown version of software, external libraries), we cannot exactly know which benchmark is more accurate for evaluation. The correctness of oracle templates in these benchmarks also raised doubts among recent research [34]. Moreover, practitioners may have different preferences regarding what should be considered static or dynamic information. To address these challenges, we introduce human feedback to verify the reliability of the benchmarks in this study.

High quality log parsing should meet the following two goals: 1) the ability to group logs from the same template together, and 2) the ability to correctly separate static and dynamic information. Therefore, benchmark data for log parsing should also serve the same goals. To assess the correctness of the produced log benchmark data by LPB-Gen, we evaluate the benchmark data correspondingly in two steps, each targeting one goal of the log parsing.

User study setups. We invite 20 participants in this user study. All participants have software engineering backgrounds and hold either a Master's or a Ph.D. degree in software engineering or related fields. When crafting questions

for the user studies, we ensure that each question is answered by at least two participants to enhance the reliability of the feedback and ensure a more robust evaluation of our results. Furthermore, we conceal the key information (e.g., which benchmark this template is from) and randomly shuffle the order of options to prevent any biased preferences.

4.1.1 The ability to group logs from the same template together. In this part, we assess the ability of LPB-Gen to group logs from the same template together. Intuitively, we do not have the capacity to have participants in our user study to manually verify all log benchmark data. More importantly, with only log lines and their templates, participants often do not have enough context to determine whether the results are fully correct. This factor directly affects the reliability of evaluations based on GA and FGA, which are the two most widely used metrics for assessing log parsers. Therefore, we design two sub-studies to strategically evaluate our benchmark data.

Different grouping results between LPB-Gen and Loghub. Our first strategy targets the different grouping results between LPB-Gen and Loghub. There exists a considerable amount of grouping results between the two datasets that are exactly the same. Although having the same results does not guarantee that the results are correct, we opt to strategically not leverage human participants on these results. Instead, we focus on the logs that are grouped differently in the two benchmarks.

In particular, we first put log lines with the same template into the same group based on the template files provided by LPB-Gen and Loghub separately. Then, we compare the groups generated by our templates and Loghub's templates. We do not study logs in groups from the two benchmarks where there exist no overlaps (meaning that the two groups are indeed different), nor exactly the same (meaning that the clustering results are identical). If the groups from the two benchmarks have overlaps but are not identical, we take the logs that exist in one group from one benchmark but not the group in the other benchmark as the log line to study. In order to evaluate whether the log line should be grouped with either of the groups, we randomly select another log line from the group and simply ask the participant whether the two log lines belong to the same group.

Figure 5 shows an illustrative example of how we choose logs to be evaluated by the participants. Template A and Template B are from LPB-Gen where log lines 1 and 3 are grouped together with Template A, while log line 2 is with Template B. On the other hand, Loghub only has one template, i.e., Template X, with all three log lines grouped together. Therefore, log line 2 is selected in the user study. Since log lines 1 and 3 are in the same group of Loghub but not in the same group of LPB-Gen, we randomly select one line, e.g., log line 1, and ask the participant whether log line 1 and log line 2 belong to the same group. For this particular example, if the participant chooses 'yes', it means that the grouping results on these log lines from Loghub are more accurate, and vice versa. We would like to emphasize that we design our user study as easily as possible for the participants to avoid the need to learn much contextual information about the logs. In this particular case, from the participants' point of view, all they need to determine is whether two log lines belong to the same group.

In our survey, we provide the two log lines to participants and ask them the following question: In this part of the user study, we show the log content from two different log lines: log 0 and log 1. You need to decide whether the two log lines belong to the same group and share the same log template.

Log lines that are more likely to be mis-grouped. Our second strategy targets the log lines that are more likely to be mis-grouped. In particular, there may exist log lines from different templates, but with similar textual content; while there may also exist log lines from the same template, but with dissimilar textual content. Therefore, based on the data from LPB-Gen, we aim to select log lines from different templates that are most similar to each other, as well as the log lines from the same templates that are the most different from each other. Due to the huge size of data in

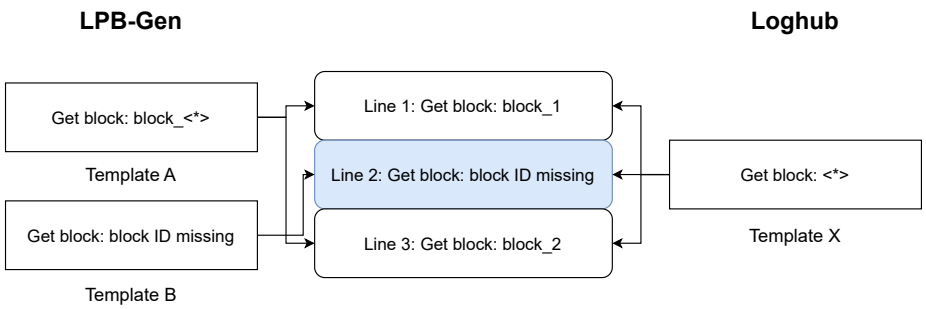


Fig. 5. Example to set up survey questions based on clustering result different from Loghub

the benchmarks, we cannot compare all the possible pairs of log lines. Instead, for each dataset, we select the top 10 most similar templates based on Levenshtein distance. Levenshtein distance measures the minimum number of single-character edits required to transform one string into another [37]. It has been widely adopted in previous research to assess text-based similarity [7, 15]. The reason for selecting similar templates is that logs with these templates tend to have similar structures. Then, we calculate the similarity based on Levenshtein distance for each log inside the same group and the similarity of each log from different groups. We select the top five most dissimilar pairs of log lines from the same group and the top 10 most similar pairs of log lines from different groups. These selected pairs of log lines exhibit either high inner-group heterogeneity or high inter-group homogeneity, both of which indicate potential misclassification within the groups.

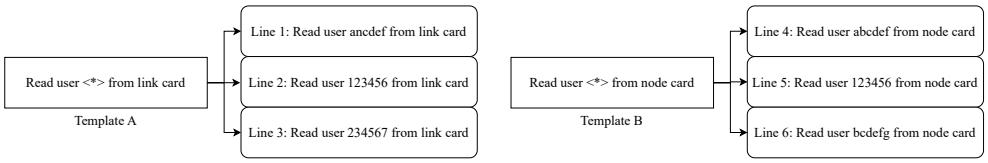


Fig. 6. Example to set survey questions based on calculating Levenshtein based similarity

Figure 6 shows an example of how we choose similar and dissimilar logs. Template A and Template B are similar log templates. For example, line 1 and line 2 are dissimilar logs from template A, and line 2 and line 4 are similar logs from different templates. We provide dissimilar and similar pairs of logs to user study participants and let them decide whether the two log lines belong to the same group. We count the number of questions on which the decision given by the participants is the same as our benchmark data. Similar to the first strategy, the participants only need to answer 'yes' or 'no' for each question.

In our survey, we provide the two log lines to participants and ask them the following question: In this part of the user study, we show the log content from two different log lines: log 0 and log 1. You need to decide whether the two log lines belong to the same group and share the same log template.

4.1.2 The ability to correctly separate static and dynamic information. In this part, we present how we design user studies to assess the ability of LPB-Gen to correctly separate static and dynamic information. In order to evaluate the separation of static and dynamic information while minimizing the influence of the grouping of logs, we would like to conduct this evaluation on the logs that are more likely to be grouped correctly. In other words, if logs are already

grouped together correctly, it determines whether the separation of static and dynamic information is correct. This factor directly affects the reliability of evaluations based on PA and FPA, which are the two most widely used metrics for assessing log parsers.

In the last part, when we evaluate LPB-Gen's ability to group logs from the same template together, we identify groups of logs where LPB-Gen and Loghub have exactly the same log lines (cf. Section 4.1.1). Since these groups of logs are more likely to be grouped correctly, we leverage this data to evaluate the separation of static and dynamic information. Intuitively, we compare the templates that are generated by the group of the same logs from LPB-Gen and Loghub. If the templates are different, the templates and one randomly selected log line will be selected for the user study.

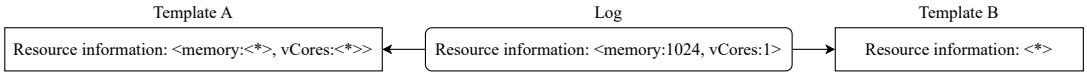
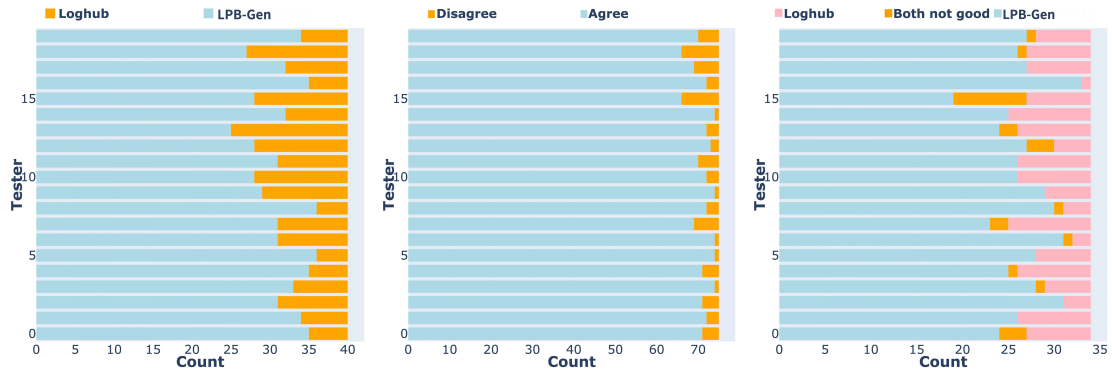


Fig. 7. Example to set survey questions based on different identification of dynamic information in templates

Figure 7 is an example of the user study in this section. We first randomly select a log from the group and provide the templates from LPB-Gen and Loghub that are different. The participants decide which of the two templates can better describe the log and can extract the dynamic information in a more accurate way. We realize that both results may be wrong. Therefore, we also provide the 'Both of the templates are wrong' option for the participants.

In our survey, we provide two log templates and a log to participants and ask them the following questions: The templates in this part of the user study can match the same logs from the dataset. In this part, we provide original log content and its corresponding template 0 and template 1. You need to choose a template that can better describe the original log content. If you think both of the templates are not accurate, you can also choose 'Both of the templates are wrong' option.

4.1.3 Result of the user studies. Figure 8 shows the overview results of the user studies.



(a) Study based on different grouping results (b) Study based on logs that are more likely to be mis-grouped (c) Study for ability to correctly separate static and dynamic information

Fig. 8. Results of the user studies

The ability to group logs from the same template together. Figure 8a shows the result of the user study on the grouping of logs compared with Loghub. There are 39 cases in this part of the user study for each participant to finish.

For all the participants, LPB-Gen can achieve a higher agreement with the decision given by the participants. The agreement of LPB-Gen and participants' decision ranges from 64.1% (25 out of 39) to 92.3% (36 out of 39). LPB-Gen's results agree with participants in 80.9% of the cases on average. The results demonstrate that the ground truth generated by LPB-Gen achieves higher accuracy in grouping logs from the same template than Loghub, thereby enabling more precise measurement of GA and FGA for log parsers.

Figure 8b shows the result of the user study on the log lines that are more likely to be mis-grouped. There are 39 questions in this part of the user study for each participant to finish. For all the participants, the agreement of LPB-Gen and participants' decisions is higher than 85%. The agreement of LPB-Gen and participants' decision ranges from 88% (66 out of 75) to 98.7% (74 out of 75). Five participants have the highest agreement (98.7%) with LPB-Gen. The average agreement of this part of the survey is 95.1% (71.3 out of 75). In total, 18 out of 20 participants can achieve an agreement higher than 90%. The results demonstrate that the ground truth generated by LPB-Gen achieves an accurate result in grouping logs that are more likely to be mis-grouped, ensuring a precise measurement of GA and FGA for log parsers.

The ability to correctly separate static and dynamic information. Figure 8c shows the result of the user study on the ability to correctly separate static and dynamic information. There are 34 questions in this part of the user study for each participant to finish. All the participants find that the templates from LPB-Gen are better in more than half of the cases. The worst result appears in the study conducted by user #15. In total, the user considers templates from our benchmark accurate in 54.3% (19 out of 34) of the cases. In 23.5% (8 out of 34) of the cases, the user considers that both of the templates from LPB-Gen and Loghub are not a good choice for the corresponding log line. On average, the users consider that the templates from LPB-Gen are more accurate in 78.7% of the cases. The results show that the ground truth generated by LPB-Gen achieves higher accuracy in distinguishing static and dynamic information compared to Loghub. This ensures a more precise evaluation of PA and FPA.

> The user study confirms that LPB-Gen produces benchmark templates that closely match human judgment and outperform existing LogHub data in both grouping accuracy and the separation of static versus dynamic log content. These results demonstrate that LPB-Gen provides a more reliable and precise ground truth for evaluating log-parsing performance.

4.2 User study on the process of LPB-Gen

In the previous subsection, we conducted user studies demonstrating that LPB-Gen can generate high-quality oracle templates, enabling practitioners to construct customized log parsing benchmarks. However, the efficiency of this process and the amount of effort required to construct a benchmark remain unclear. A process that demands excessive time or manual intervention could limit its practical applicability. Moreover, since LPB-Gen includes a manual refinement step, the interpretation of templates may vary across practitioners with different levels of expertise, potentially introducing inconsistencies into the generated benchmarks and affecting subsequent log parsing and analysis tasks. We also examine whether varying the sample size influences the outcomes and workflow of our semi-automated process.

Therefore, in this section, we conduct a user study to evaluate the efficiency and consistency of LPB-Gen, investigate potential inconsistencies in the oracle templates produced by different practitioners, and analyze the effect of varying sample sizes. Specifically, we perform a replication study to verify that participants with different experience levels can efficiently use LPB-Gen to generate accurate benchmark data. We further summarize the factors that may lead to inconsistencies to help practitioners produce more reliable results and conduct an additional sensitivity analysis with a single participant to provide guidance on selecting appropriate sample sizes.

User study setups. We invite participants in the software engineering domain to reproduce the oracle template generation process using LPB-Gen. Since users' experience on logging and log analysis may impact the ease of adoption and results of following LPB-Gen, we invite two participants who have different experiences in logging and log analysis. Specifically, the experienced participant had extensive prior hands-on experience and background knowledge in log analysis and log parsing before this study, while the inexperienced participant only acquired the necessary background knowledge during the user study itself. Our semi-automated process is designed to minimize human errors, ensuring that the benchmark data stem from individual preference in interpreting log structure and deciding how logs should be parsed, rather than from mistakes. We intentionally selected participants with different experience levels to evaluate whether LPB-Gen can produce consistent benchmark data regardless of users' prior expertise. These results prove that our framework can reduce human subjectivity and support reproducible benchmark generation across varying practitioner backgrounds.

In addition, since this user study is rather extensive in its workload (essentially, each participant needs to reproduce log parsing benchmarks), we cannot let the participants reproduce all the log parsing data. Instead, we choose three representative log datasets (i.e., Apache, BGL, Hadoop) based on their different sizes (i.e., small, medium, and large) to represent the real-world scenarios of constructing log parsing benchmarks. As LPB-Gen operates as a semi-automated process, we track the elapsed time for both the manual refining process and the automated steps for all participants. The evaluation results on the selected logs shall provide references for practitioners to estimate the time and effort required to construct dedicated benchmarks on their specific dataset by size and complexity.

To examine the inconsistencies observed during the template refining process by different practitioners, we use the generated benchmark by the authors (as demonstrated in Table 8) as a baseline. Each new participant's generated benchmark is then compared against this baseline to identify differences. We follow the same approach as Section 3 to locate logs that are matched by different templates from different benchmarks. We then choose a log line for each unique pair of templates for further investigation. Next, the authors of this paper employ a manual labeling process to identify and categorize the differences, with each author independently conducting the labeling process. After the labeling process, the authors gather to discuss the observed differences, ensuring consistency in categorization and refining the categories based on their observations and insights. The labeling process is repeated until the results converge and all authors reach a consensus. This iterative process helps to categorize and summarize the observed discrepancies into a unified set of **inconsistency factors**, which is presented in Section 4.2.3.

To examine the impact of varying sample sizes on the results and workflow of our semi-automated process, we invited one participant with expertise in software engineering to reproduce the oracle template generation process using LPB-Gen with different sample sizes (5,000, 20,000, and 50,000). We selected three representative log datasets (Apache, BGL, and Spark) for this sensitivity analysis. During the experiment, we recorded the total time required to complete the generation process and calculated the group consistency and parsing consistency between the generated benchmark data and our ground truth.

4.2.1 Efficiency of the replication test. Table 5 shows the total amount of time that participants take to reproduce the benchmarks. The results demonstrate that both participants are able to independently reproduce log parsing benchmarks with a manageable effort (2 to 4 iterations), regardless of the size of logs and their experience level. We also observe that while the experience level does significantly impact the manual refining time on small datasets, the experienced participant tends to complete the manual refining process more quickly than the less experienced participant when handling larger datasets. While experience could be an impactful factor, our observation shows that both participants

Table 5. The efficiency and consistency evaluation of replication user study.

Dataset		Experiment Details		Efficiency Metrics		Consistency Metrics	
Logs	**Size**	**Participant**	**#Iterations**	**Manual refining time (HH:MM:SS)**	**Automated process time (HH:MM:SS)**	**Group consistency**	**Parsing consistency**
Apache	4.90 MB (small)	Inexperienced	2	00:08:24	00:00:05	1.00	0.87
		Experienced	2	00:08:18	00:00:03	1.00	0.87
BGL	708.76 MB (medium)	Inexperienced	2	01:51:20	00:02:29	0.96	0.77
		Experienced	3	01:21:49	00:01:01	0.97	0.79
Spark	2.75 GB (large)	Inexperienced	4	01:20:29	00:10:40	0.91	0.71
		Experienced	3	01:08:22	00:04:29	0.89	0.57

of different experience levels can still efficiently complete the benchmarking process within a reasonable time frame (less than 2 hours). Such results underscore the efficiency and adaptability of LPB-Gen, showing that it can be used by practitioners with different expertise and across logs of various scales, making it suitable for real-world benchmarking scenarios.

Specifically, we observe that log size is not a deterministic factor that impacts the total efforts in benchmarking. As shown Table 5, the medium-sized dataset (i.e., BGL) takes both participants more time and iterations to complete as compared to a much larger dataset (i.e., Spark). The underlying rationale behind this observation is that although BGL is smaller than Spark in size, the logs in the BGL dataset are more diverse and complex than the Spark dataset, leading to a larger amount of unique templates to be examined. Furthermore, the complex and lengthy logs consume extra time and effort in the manual refining process for both participants. Practitioners may leverage our evaluation results as references to estimate the time and effort of a benchmarking process according to the size and characteristics of their specific log dataset.

4.2.2 Consistency of the replication test. To evaluate the consistency of the replication test, based on the group accuracy and parsing accuracy as introduced in Sections 2.2, we propose two metrics: 1) group consistency and 2) parsing consistency.

Group consistency measures whether the grouping result generated with LPB-Gen by different practitioners is consistent. Parsing consistency measures whether the identification of dynamic information and static information with LPB-Gen by practitioners is exactly the same. We calculate group consistency based on the group accuracy that is defined in prior research [97] between each participant's benchmark and the baseline. Similarly, parsing consistency is calculated based on the portion of different templates between each participant's benchmark and the baseline.

The consistency evaluation results are shown in Table 5. Specifically, we observe that benchmarks produced by the two new participants demonstrate a generally high group consistency with the original benchmark that is produced by the authors, with values ranging from 0.89 to 1.00. The high group consistency demonstrates that LPB-Gen can effectively produce consistent and reliable grouping results.

On the other hand, we observe variations within the parsing consistency (ranging from 0.57 to 0.87) shown in Table 5. Such a variation indicates that although LPB-Gen can produce convincing grouping results, practitioners have their personal perspective in deriving templates from a group of similar logs, which can be subjective. The reason for the varying parsing consistency is that this type of consistency needs participants to achieve exactly the same common sense at the character level, while such decisions are often personal preferences. For example, sometimes the only difference between two templates is merely a white space. It is also needed to mention that the inconsistency part leads to group inconsistency, which also leads to parsing inconsistency. The inconsistency part leads to parsing inconsistency, which may not lead to group inconsistency. For example, when we set up the user study for the ability to correctly

separate static and dynamic information, all the cases have the same clustering result but have different templates. In Section 4.2.3, we further discuss the reasons that lead to the inconsistent results of the replication test.

4.2.3 Factors leading to inconsistent log templates. In this part, we summarize the seven main factors leading to the inconsistency in the replication of the benchmarking. We start by aggregating the log templates curated by various participants and identifying those with differing content across their annotations. Each participant then independently reviews these differences and compiles a list of factors contributing to the inconsistencies. Subsequently, all participants meet to discuss their findings and collaboratively establish a standardized set of inconsistency factors. Following this, participants revisit the inconsistencies, apply labels based on the agreed-upon standard, and discuss any remaining discrepancies in their labeling. This iterative process continues until a consensus is achieved among all participants, ensuring both consistency and comprehensive coverage during the labeling process. We would like to note that we call these "inconsistencies" since for most of these factors (except for the human-made mistake), there is no clear distinction of right or wrong on the parsing results, but rather personal preferences. The summarized inconsistency factors serve as a reference, enabling practitioners to develop their labeling standards based on these factors to ensure consistency when curating templates tailored to their specific log datasets.

Human-made mistake refers to the inconsistencies between templates caused by human mistakes during manual refining. This factor is usually inevitable in manual processes. The following example showcases inconsistencies of this type, with three lines representing the original log content, the template defined by the participant, and the template from the baseline, respectively. In this example, the log line is accidentally omitted by the participant during the manual labeling process, rendering dynamic information such as `status` and `exitCode` unlabeled.

(1) **Original:** `Final app status: FAILED, exitCode: 1, (reason: User application exited with status 1`
(2) **Participant:** `Final app status: FAILED, exitCode: 1, (reason: User application exited with status 1)`
(3) **Baseline:** `Final app status: <*>, exitCode: <*>, (reason: User application exited with status <*>)`

Option refers to one category of information that has a limited number of options. In the following example, the log content in the first two lines differs in terms of `bytes` and `values`, which is likely one of the only few options for describing the data storage method in this context. Such a difference can be seen as dynamic if viewed as varying components that change along the text or can be considered static if they represent fixed options of categories in the template. The distinction is subjective and depends on how individuals interpret logs.

(1) **Original:** `Block broadcast_4_piece247 stored as bytes in memory (estimated size 4.0 MB, free 3.0 GB)`
(2) **Original:** `Block broadcast_4_piece247 stored as values in memory (estimated size 4.0 MB, free 3.0 GB)`
(3) **Participant:** `Block <*> stored as bytes in memory (estimated size <*>, free <*>)`
(4) **Participant:** `Block <*> stored as values in memory (estimated size <*>, free <*>)`
(5) **Baseline:** `Block <*> stored as <*> in memory (estimated size <*>, free <*>)`

Long variable indicates a long token typically composed of multiple subtokens connected by symbols, often displaying a distinct pattern. Practitioners may have different perspectives on whether to treat each subtoken individually or consider the entire long token as a single dynamic variable. As shown in the following example, the inconsistency arises from whether `13901` should be separated from the long variable `core.13901`.

(1) **Original:** `ciod: for node 77, incomplete data written to core file core.13901`
(2) **Participant:** `ciod: for node <*>, incomplete data written to core file core.<*>`
(3) **Baseline:** `ciod: for node <*>, incomplete data written to core file <*>`

Multi-dynamic factor refers to a group of consecutive tokens that can be grouped together to represent one dynamic information entity. In the following example, for consecutive tokens like *1044.8 MB*, practitioners may opt for different levels of granularity of templates, deciding whether to treat these tokens separately or aggregate them into a single token.

(1) **Original:** `Finished task 2.0 in stage 3.0 (TID 12). Result is larger than maxResultSize (1044.8 MB > 1024.0 MB), dropping it.`
(2) **Participant:** `Finished task <*> in stage <*> (TID <*>). Result is larger than maxResultSize (<*> MB > <*> MB), dropping it.`
(3) **Baseline:** `Finished task <*> in stage <*> (TID <*>). Result is larger than maxResultSize (<*> > <*>), dropping it.`

Parameter refers to scenarios in which practitioners disagree on whether a parameter-like token is a static text or dynamic variable. The file path (`/usr/sbin/suexec`) in the following example can be interpreted as a dynamic variable that is printed at runtime or the static text that developers placed in logging statements for traceability concerns.

(1) **Original:** `suEXEC mechanism enabled (wrapper: /usr/sbin/suexec)`
(2) **Participant:** `suEXEC mechanism enabled (wrapper: /usr/sbin/suexec)`
(3) **Baseline:** `suEXEC mechanism enabled (wrapper: <*>)`

Debug information refers to context that is related to debug information, like exception type and its corresponding description. In the following example, the participant perceives the exception class information `java.io.IOException` as static text to explain the causes of the connection closure failure. In the baseline, however, the exception class information is treated as dynamic information.

(1) **Original:** `Ignored failure: java.io.IOException: Connection from mesos-master-1/10.10.34.11:34641 closed`
(2) **Participant:** `Ignored failure: java.io.IOException: Connection from <*> closed`
(3) **Baseline:** `Ignored failure: <*>: Connection from <*> closed`

Punctuation inconsistency refers to cases where punctuation in log lines cannot be determined to be part of dynamic variables or static text. In the following example, whether the single quote character `'` is embedded in the static text or printed from a dynamic object at runtime raises inconsistencies among participants.

(1) **Original:** `Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 34607.`
(2) **Participant:** `Successfully started service '<*>' on port <*>.`
(3) **Baseline:** `Successfully started service <*> on port <*>.`

4.2.4 Proportion of different inconsistency factors. We present the statistics of the inconsistency factors that are observed during the replication user study, as shown in Figure 9. First, the number of inconsistencies is proportional to the size of the logs, meaning an increasing number of inconsistencies are likely to occur when refining larger-sized datasets. Additionally, we observe that the benchmark produced by the experienced participant tends to contain fewer inconsistencies compared to those generated by the inexperienced participant. Although LPB-Gen is accessible to practitioners with varying levels of experience, we recommend that experienced practitioners collaborate on the benchmarking process when handling large and complex logs to minimize inconsistencies in the final benchmarks.

We observe that parameter-related discrepancies account for a significant proportion of the inconsistencies in most of the studied cases. As developers sometimes embed parameter values inside static text when they write logging statements, distinguishing between static or dynamic information based solely on logs can be highly subjective and challenging. Even during an offline discussion after the user study, neither the authors nor the participants can

Table 6. Impact of efficiency with different sample sizes

Dataset		Efficiency Metrics	
Logs	**Sample Size**	**Manual refining time (HH:MM:SS)**	**Automated process time (HH:MM:SS)**
Apache	5,000	00:04:44	00:00:03
	20,000	00:04:49	00:00:04
	50,000	00:04:44	00:00:01
BGL	5,000	01:10:17	00:00:38
	20,000	01:11:22	00:00:53
	50,000	01:12:48	00:01:01
Spark	5,000	00:45:21	00:06:27
	20,000	00:51:16	00:06:51
	50,000	00:56:05	00:08:01

reach a consensus on the proper affiliation of the parameters. Multi-dynamic inconsistent factors are also commonly observed across the studied cases. Specifically, practitioners may opt for different levels of granularity when refining logs, particularly when determining whether consecutive parameters should be merged. This flexibility is subject to practitioners' preferences and judgments, as it involves a tradeoff between the informativeness of the oracle template and its overall conciseness.

We also observe types of consistencies that mostly occur in certain datasets. For example, Debug and Option factors majorly occur when comparing benchmarks that are based on Spark and BGL datasets. The occurrences of such factors are highly dependent on the nature of the target dataset. The BGL dataset comprises system logs that are collected from a BlueGene/L supercomputer. Such logs usually include system-specific parameters such as cache levels (e.g., L1, L3) that can be interpreted by participants as static content, thus leading to the Option typed inconsistency. On the other hand, the Spark dataset contains application logs produced by the Apache Spark system. Due to the nature of the specific system, the corresponding dataset comprises more debug information (e.g., system runtime exception) in the log data, leading to an increasing number of debug information related inconsistencies observed in this user study.

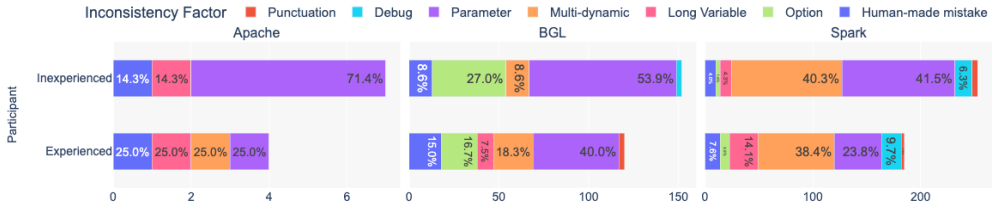


Fig. 9. The proportion of inconsistency factors that are observed in the replication user study.

4.2.5 Impact of varying sample sizes. We present the time required to generate oracle templates with different sample sizes. Table 6 summarizes the recorded time for completing the oracle template generation process under each sample size setting.

The results show that varying the sample size has only a limited impact on the efficiency of our semi-automated process. For the Apache and BGL datasets, the difference in processing time remains minimal (within approximately two minutes). The largest variation occurs in the Spark dataset, where the difference between sample sizes is around five minutes.

Table 7. Group and parsing consistency of benchmark data with different sample sizes

Dataset	Sample size	Group consistency	Parsing consistency
Apache	5,000	1.0	1.0
	50,000	1.0	1.0
BGL	5,000	0.991	0.946
	50,000	0.966	0.955
Spark	5,000	0.999	0.905
	50,000	1.0	0.926

We also calculate the group consistency and parsing consistency of the generated benchmark data. The benchmark data with a sample size of 20,000 is used as the baseline, as it serves as the default setting in this paper. We then compute the group consistency and parsing consistency between this baseline and the benchmark data obtained with sample sizes of 5,000 and 50,000. Table 7 presents the group consistency and parsing consistency results across the three datasets.

The results indicate that varying the sample size has only a limited impact on the benchmarking outcomes. Across all datasets, the group consistency remains high (above 0.95). In three out of six cases, the group consistency reaches 1.0, indicating that the clustering results are completely identical across benchmarks generated with different sample sizes. The parsing consistency also remains high (above 0.9) across all datasets, indicating that practitioners' preferences for identifying dynamic and static information are consistent under different sample sizes.

The results indicate that varying the sample size has only a limited impact on the benchmark data. Therefore, practitioners using LPB-Gen can flexibly choose their preferred sample size without significantly affecting the outcomes.

> Our replication study demonstrates that the LPB-Gen methodology allows for efficient and consistent benchmark generation across practitioners of all expertise levels. Both inexperienced and experienced participants completed the process in under 2 hours, achieving high group consistency (0.89-1.00). Although parsing consistency varied more (0.57-0.87) due to subjective interpretation, we identified seven main inconsistency factors that can guide practitioners in establishing their labeling standards. Furthermore, the minimal impact of sample size variation on efficiency and consistency offers practitioners flexibility in their workflow.

4.3 Quality validation of the log benchmark dataset

In this subsection, we discuss two additional concerns about the quality of the generated benchmark data for log parsing. In particular, we are concerned 1) whether the log parsing results in the benchmark data can cover the entire raw log data and 2) whether the log content extraction step provided by the benchmark is correct.

4.3.1 Coverage of the benchmark data. Not all text lines in the raw logs are appropriate to be parsed. As discussed in Section 3, some log lines have empty content, which is not suitable for parsing. In addition, there are log lines that may contain only illegal characters or information that is not meaningful to the application. In these cases, the log parsing benchmarks may not contain corresponding parsing results for these lines. In particular, we find that there exists a considerable amount of log lines that are not covered by Loghub.

We compare all the log lines that are covered by the benchmark data generated by LPB-Gen and Loghub. Table 8 shows the covered log lines in each log dataset by Loghub and LPB-Gen. We find that LPB-Gen covers a larger amount of logs than Loghub. In extreme cases like HealthApp and Thunderbird, there exists a considerable amount of log lines

in the raw logs that are not covered by Loghub, while covered by LPB-Gen. In addition, we find that the covered log lines in all projects from LPB-Gen are always supersets of those from Loghub.

Table 8. Log matching statistics between Loghub and LPB-Gen

LogName	Apache	BGL	Hadoop	HDFS	HealthApp	HPC	Linux
Total lines (Loghub)	51,977	4,631,261	179,993	11,167,740	212,394	429,987	23,921
Total lines (LPB-Gen)	52,004	4,688,579	180,960	11,175,629	253,395	433,487	25,456
Percentage Increase	0.05%	1.24%	0.54%	0.07%	19.30%	0.81%	6.42%

LogName	Mac	OpenSSH	OpenStack	Proxifier	Spark	Thunderbird (Sample)	Zookeeper
Total lines (Loghub)	100,314	638,946	207,632	21,320	16,075,117	177,644	74,273
Total lines (LPB-Gen)	102,768	655,147	207,632	21,320	27,411,538	355,390	74,380
Percentage Increase	2.45%	2.54%	0.00%	0.00%	70.52%	100.06%	0.14%

4.3.2 Correctness of log content extraction . In the log parsing process, developers usually utilize regular expressions to extract the header information and log content directly from raw logs. This step, although extremely important, is often less focused since one may utilize the configuration of the corresponding logging library to generate a regular expression to extract header information and log content for each log dataset. The availability of such a regular expression makes the content extraction step a trivial problem. This is also the case for all datasets in Loghub except for Thunderbird. In Thunderbird, as well as logs analyzed in practice, there exist cases where multi-formatted logs are placed inside one dataset. Therefore, LPB-Gen utilizes a mixture of regular expressions to extract log content of different formats within the same dataset. Whether the log contents are extracted correctly can influence the quality of the log benchmark dataset.

In order to assess the correctness of the content extraction, similar to our user studies presented before, we invited all 20 participants to choose the correct extraction of log header information and log content for a sampled eight unique cases of differences between LPB-Gen and Loghub. The participants also have the option of both benchmarks being wrong. We would like to note that, although there are only eight unique cases since logs follow the same templates, there exists a large number of log lines that are impacted by these different content extraction results. We present one example as follows, where the first line is the original log in the Thunderbird dataset. The second and third lines are log content extracted by Loghub and LPB-Gen. We observe that *VFS:* is categorized into header information by Loghub, while it is recognized as part of log content by LPB-Gen.

(1) **Raw Log**: `- 1131682200 2005.11.10 tbird-admin1 Nov 10 20:10:00 local@tbird-admin1 VFS: file-max limit 65536 reached`
(2) **Content by Loghub**: `file-max limit 65536 reached`
(3) **Content by LPB-Gen**: `VFS: file-max limit 65536 reached`

We find that for 123 out of 160 (77%) questions, participants think LPB-Gen extracts more accurate log content than Loghub, while content extracted by Loghub is preferred in 31 (19%) questions. Notably, two participants consistently choose LPB-Gen as the better option across all eight questions. On average, LPB-Gen's content is chosen as the better option in 6.2 out of 8 questions, demonstrating a consistent preference among participants. In summary, LPB-Gen is perceived to provide notable advancements in content extraction, providing a strong basis for further log parsing and analytics tasks.

4.3.3 Comparison with source code extracted benchmark data. Although we conduct a survey to evaluate the accuracy of the benchmark data generated by LPB-Gen, a direct evaluation using a more objective ground truth is still needed to ensure the reliability of the generated benchmarks. To perform this evaluation, we require benchmark data extracted directly from the source code. However, the datasets provided by Loghub [96] lack version information about the systems used to generate the log data. Therefore, we deployed a Hadoop 3.2.1 environment consisting of one server and three worker nodes to generate logs for comparison.

We executed several example tasks provided by Hadoop, including pi, terasort, distbbp, and wordcount. For the wordcount task, we ran it on the log data from Loghub [96]. We then collected the logs through Hadoop's log aggregation mechanism and used them as the dataset for this part of our evaluation. As some logs are generated by external libraries such as Jetty, we remove these entries and obtain the final dataset containing 82,304 log lines.

To extract the benchmark data from the source code, we leverage the package information contained in the log header to locate the source file where each log line is generated. We then extract the logging statements from the corresponding file and match their text content to the log lines. Finally, we verify the matched log lines and their corresponding logging statements to extract the log templates. The log templates can be used to generate more objective benchmark data from the source code of Hadoop 3.2.1.

Similar to the procedure described in Section 3, we invited two participants with different levels of experience to use LPB-Gen to generate benchmark data for this dataset. We then calculated the group consistency between the benchmark data generated by LPB-Gen and those extracted from the source code to evaluate whether participants with different levels of expertise could produce accurate benchmark data using LPB-Gen.

Table 9. Group consistency of benchmark data from participants with the source code

Dataset	Inexperienced	Experienced
Hadoop	0.974	0.905

Table 9 presents the group consistency results. As shown, the benchmark data generated by both the inexperienced and experienced participants achieve a high level of consistency with that extracted from the source code. This finding indicates that LPB-Gen can produce benchmark data with high accuracy, thereby ensuring the reliability of evaluations conducted on benchmarks generated by LPB-Gen.

As extracting benchmark data directly from source code can be challenging due to issues such as unavailable source code, unknown system versions, and system evolution, LPB-Gen provides a more general and practical strategy for generating reliable benchmark data for log parsing evaluation.

> LPB-Gen demonstrates high quality across multiple dimensions. It achieves significantly greater log line coverage than LogHub (up to 100% increase for Thunderbird), and its content extraction was preferred by participants in 77% of cases. Furthermore, validation against source code-derived benchmarks from Hadoop 3.2.1 shows high group consistency (>0.9), confirming that LPB-Gen produces reliable benchmark data comparable to an objective ground truth while remaining practical for scenarios where source code analysis is infeasible.

5 Threats to Validity

External Validity. The evaluation of our approach is conducted on a limited number of datasets. While it is impossible to encompass every log parsing scenario, the evaluation results indicate that our approach is applicable and generalizable

to various types of log data, enabling practitioners to adapt our approach to conduct the benchmarking process efficiently on their specific datasets. Furthermore, although the evaluation of LPB-Gen is conducted only on logs collected from Loghub [96], the results demonstrate that our framework can be seamlessly integrated into existing log collection and processing pipelines. This study thus highlights a promising direction toward developing an end-to-end solution that bridges log collection and benchmark creation.

Internal Validity. In this paper, we choose two popular, lightweight yet effective log parsers to validate the effectiveness of our approach. Practitioners may use a different selection of log parsers with our framework according to their specific requirements. The manual refining process relies on the user's subjective judgment, which can cause variances in templates when generating benchmarks. It is based on the person's expertise and perception of the system and how the parsing results will be used for further analysis. During the replication user study, the participants conduct the benchmarking independently on their own devices, which may cause differences in automated process execution time. However, as discussed in Section 4.1, both participants are capable of reproducing the benchmarking process within a reasonable time frame, where the automated process takes less than 5 minutes to complete, even for large-sized log datasets. Therefore, the variances in hardware devices shall not impact the overall efficiency of the benchmarking process. Our process primarily supports system logs with relatively stable formats, which allows us to cover a broad range of widely used system datasets. However, we acknowledge that it does not yet support terminal logs or multi-line logs, which present additional challenges (e.g., lack of consistent structure, interleaving of user inputs/outputs).

Construct Validity. In this study, we use two metrics (e.g., group consistency and parsing consistency) to assess whether the benchmark data produced by different participants are consistent. However, these metrics may not fully capture all aspects of consistency in log parsing results. In addition, we strategically select a limited number of cases for the user study to ensure participants are not overwhelmed. These selected cases may not fully represent all the data in the benchmarks. A few studies, like SemParser [28], have begun to explore the semantics of variable annotation. Our framework can support human annotations of semantic information with relatively low effort. However, scaling such annotations across diverse systems would require significant domain expertise for each system to correctly identify and label variables that could be further leveraged for log analysis tasks.

6 Conclusion

In this paper, we propose a semi-automated process, LPB-Gen, to generate oracle templates for system log datasets. LPB-Gen leverages state-of-the-art log parsers to generate an aligned parsing result file, and practitioners can utilize this file to refine oracle templates. To evaluate LPB-Gen, we design and conduct user studies to assess the correctness of the generated benchmarks, as well as the efficiency and reproducibility of the process of benchmarking with LPB-Gen. Furthermore, we summarize factors that may cause inconsistencies in log parsing results. Our analysis of the inconsistency factors indicates that the definition of an effective log template may vary between individuals, as it depends on practitioners' interpretations and the specific downstream tasks where these templates are utilized. Future log parsing techniques should adopt approaches tailored to individual needs and downstream task requirements. Finally, we release our generated log benchmark datasets for future research and potential extension of this work.

Given the heterogeneity of log data and the domain expertise required for log interpretation, manual labeling is essential to ensure the accuracy and meaningfulness of the produced log templates; thus, it is challenging to eliminate the manual process when curating oracle log templates. In future research, we will leverage the reasoning and inference capabilities of Large Language Models (LLMs) to alleviate the human effort required in template identification and to enhance the efficiency of the log parsing benchmark construction process.

7 Data Availability

The source code and dataset generated and evaluated in this study are available and shared with the public in a supplementary material package [1].

References

[1] [n. d.]. Public Security Log Sharing. https://log-sharing.dreamhosters.com/ [Online; accessed 2025-10-01].

[2] Amey Agrawal, Abhishek Dixit, Namrata A Shettar, Darshil Kapadia, Vikram Agrawal, Rajat Gupta, and Rohit Karlupia. 2019. Delog: A high-performance privacy preserving log filtering framework. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 1739–1748.

[3] Amey Agrawal, Rohit Karlupia, and Rajat Gupta. 2019. Logan: A distributed online log parser. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1946–1951.

[4] Michal Aharon, Gilad Barash, Ira Cohen, and Eli Mordechai. 2009. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 227–243.

[5] Nicolas Aussel, Yohan Petetin, and Sophie Chabridon. 2018. Improving performances of log mining for anomaly prediction through nlp-based log parsing. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 237–243.

[6] Titus Barik, Robert DeLine, Steven M. 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, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 92–101.

[7] Zhichao Chen, Junjie Chen, Weijing Wang, Jianyi Zhou, Meng Wang, Xiang Chen, Shan Zhou, and Jianmin Wang. 2023. Exploring Better Black-Box Test Case Prioritization via Log Analysis. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 72:1–72:32. https://doi.org/10.1145/3569932

[8] Guojun Chu, Jingyu Wang, Qi Qi, Haifeng Sun, Shimin Tao, and Jianxin Liao. 2021. Prefix-graph: A versatile log parsing approach merging prefix tree with probabilistic graph. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2411–2422.

[9] Edward Chuah, Shyh-hao Kuo, Paul Hiew, William-Chandra Tjhi, Gary Lee, John Hammond, Marek T Michalewicz, Terence Hung, and James C Browne. 2010. Diagnosing the root-causes of failures from cluster log files. In *2010 International Conference on High Performance Computing*. IEEE, 1–10.

[10] Zhang Chunyong and Xiaojing Meng. 2020. Log parser with one-to-one markup. In *2020 3rd International Conference on Information and Computer Technologies (ICICT)*. IEEE, 251–257.

[11] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. 2015. The making of cloud applications: an empirical study on software development for the cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 393–403.

[12] Oihana Coustié, Josiane Mothe, Olivier Teste, and Xavier Baril. 2020. METING: A Robust Log Parser Based on Frequent n-Gram Mining. In *2020 IEEE International Conference on Web Services (ICWS)*. 84–88. https://doi.org/10.1109/ICWS49710.2020.00018

[13] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2022. Logram: Efficient Log Parsing Using *n*n-Gram Dictionaries. *IEEE Transactions on Software Engineering* 48, 3 (2022), 879–892. https://doi.org/10.1109/TSE.2020.3007554

[14] Hetong Dai, Yiming Tang, Heng Li, and Weiyi Shang. 2023. PILAR: Studying and mitigating the influence of configurations on log parsing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 818–829.

[15] Zishuo Ding, Yiming Tang, Xiaoyu Cheng, Heng Li, and Weiyi Shang. 2024. *LoGenText-Plus*: Improving Neural Machine Translation Based Logging Texts Generation with Syntactic Templates. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2024), 38:1–38:45. https://doi.org/10.1145/3624740

[16] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[17] Qiang Fu, Jian-Guang Lou, Qingwei Lin, Rui Ding, Dongmei Zhang, and Tao Xie. 2013. Contextual analysis of program logs for understanding system behaviors. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. 397–400.

[18] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*. IEEE, 149–158.

[19] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*. IEEE, 149–158.

[20] Shuting Guo, Zheng Liu, Wenyan Chen, and Tao Li. 2018. Event extraction from streaming system logs. In *International Conference on Information Science and Applications*. Springer, 465–474.

[1] https://github.com/senseuwaterloo/BigLogBench

[21] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: Fast Pattern Recognition for Log Analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (Indianapolis, Indiana, USA) *(CIKM '16)*. Association for Computing Machinery, New York, NY, USA, 1573–1582. https://doi.org/10.1145/2983323.2983358

[22] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. 2018. Towards Automated Log Parsing for Large-Scale Log Data Analysis. *IEEE Transactions on Dependable and Secure Computing* 15, 6 (2018), 931–944. https://doi.org/10.1109/TDSC.2017.2762673

[23] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017*. 33–40.

[24] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*. IEEE, 33–40.

[25] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 207–218.

[26] Junjie Huang, Zhihan Jiang, Zhuangbin Chen, and Michael R Lyu. 2024. ULog: Unsupervised Log Parsing with Large Language Models through Log Contrastive Units. *arXiv preprint arXiv:2406.07174* (2024).

[27] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. 2020. Paddy: An Event Log Parsing Approach using Dynamic Dictionary. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. 1–8. https://doi.org/10.1109/NOMS47738.2020.9110435

[28] Yintong Huo, Yuxin Su, Cheryl Lee, and Michael R. Lyu. 2023. SemParser: A Semantic Parser for Log Analytics. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 881–893. https://doi.org/10.1109/ICSE48619.2023.00082

[29] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2023. LILAC: Log Parsing using LLMs with Adaptive Parsing Cache. *arXiv preprint arXiv:2310.01796* (2023).

[30] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper). In *Proceedings of the Eighth International Conference on Quality Software, QSIC 2008, 12-13 August 2008, Oxford, UK*, Hong Zhu (Ed.). IEEE Computer Society, 181–186. https://doi.org/10.1109/QSIC.2008.50

[31] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting Execution Logs to Execution Events for Enterprise Applications (Short Paper). In *2008 The Eighth International Conference on Quality Software*. 181–186. https://doi.org/10.1109/QSIC.2008.50

[32] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2008. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance* 20, 4 (2008), 249–267.

[33] Ying Li Jiawei Liu, Zhirong Hou. 2018. Lopper: An Efficient Method for Online Log Pattern Mining Based on Hybrid Clustering Tree | SpringerLink. https://link.springer.com/chapter/10.1007/978-3-030-27615-7_5#citeas [Online; accessed 2025-10-02].

[34] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel Briand. 2022. Guidelines for assessing the accuracy of log message template identification techniques. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1095–1106. https://doi.org/10.1145/3510003.3510101

[35] Tatsuaki Kimura, Keisuke Ishibashi, Tatsuya Mori, Hiroshi Sawada, Tsuyoshi Toyono, Ken Nishimatsu, Akio Watanabe, Akihiro Shimoda, and Kohei Shiomoto. 2014. Spatio-temporal factorization of log data for understanding network events. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 610–618. https://doi.org/10.1109/INFOCOM.2014.6847986

[36] Satoru Kobayashi, Kensuke Fukuda, and Hiroshi Esaki. 2014. Towards an NLP-based log template generation algorithm for system log analysis. In *Proceedings of the ninth international conference on future internet technologies*. 1–4.

[37] VI Lcvenshtcin. 1966. Binary coors capable or 'correcting deletions, insertions, and reversals. In *Soviet Physics-Doklady*, Vol. 10.

[38] Tao Li, Feng Liang, Sheng Ma, and Wei Peng. 2005. An integrated framework on mining logs files for computing system management. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 776–781.

[39] Teng Li, JianFeng Ma, and Cong Sun. 2018. Dlog: diagnosing router events with syslogs for anomaly detection. *The Journal of Supercomputing* 74, 2 (2018), 845–867.

[40] Xiaoyun Li, Pengfei Chen, Linxiao Jing, Zilong He, and Guangba Yu. 2020. Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 92–103.

[41] 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* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 102–111. https://doi.org/10.1145/2889160.2889232

[42] Kunpeng Xie Zhi Wang Ye Lu Yujun Zhang Lin Zhang, Xueshuo Xie. 2018. An Efficient Log Parsing Algorithm Based on Heuristic Rules | SpringerLink. https://link.springer.com/chapter/10.1007/978-3-030-29611-7_10#citeas [Online; accessed 2025-10-02].

[43] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R Lyu. 2019. Logzip: Extracting hidden structures via iterative clustering for log compression. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 863–873.

[44] Weiyou Liu, Xu Liu, Xiaoqiang Di, and Binbin Cai. 2020. FastLogSim: A quick log pattern parser scheme based on text similarity. In *International Conference on Knowledge Science, Engineering and Management*. Springer, 211–219.

[45] Xiaojian Liu, Yi Zhu, and Shengwei Ji. 2020. Web log analysis in genealogy system. In *2020 IEEE International Conference on Knowledge Graph (ICKG)*. IEEE, 536–543.

[46] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. Llmparser: An exploratory study on using large language models for log parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[47] Adetokunbo Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2011. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering* 24, 11 (2011), 1921–1936.

[48] Weibin Meng, Ying Liu, Federico Zaiter, Shenglin Zhang, Yihao Chen, Yuzhe Zhang, Yichen Zhu, En Wang, Ruizhi Zhang, Shimin Tao, et al. 2020. Logparse: Making log parsing adaptive through word classification. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 1–9.

[49] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on Program Comprehension*. 167–177.

[50] Masayoshi Mizutani. 2013. Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing*. IEEE, 595–602.

[51] Meiyappan Nagappan and Mladen A Vouk. 2010. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 114–117.

[52] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2020. Self-supervised log parsing. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 122–138.

[53] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2021. Self-supervised log parsing. In *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track: European Conference, ECML PKDD 2020, Ghent, Belgium, September 14–18, 2020, Proceedings, Part IV*. Springer, 122–138.

[54] Thieu Nguyen, Satoru Kobayashi, and Kensuke Fukuda. 2021. Logdtl: Network log template generation with deep transfer learning. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 848–853.

[55] Xia Ning, Geoff Jiang, Haifeng Chen, and Kenji Yoshihira. 2014. HLAer: A system for heterogeneous log analysis. In *SDM Workshop on Heterogeneous Learning*. 1.

[56] Prabhat Pokharel, Roshan Pokhrel, and Basanta Joshi. 2023. A hybrid approach for log signature generation. *Applied Computing and Informatics* 19, 1/2 (2023), 108–121.

[57] Tongqing Qiu, Zihui Ge, Dan Pei, Jia Wang, and Jun Xu. 2010. What happened in my network: mining network events from router syslogs. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 472–484.

[58] Jared Rand and Andriy Miranskyy. 2021. On automatic parsing of log records. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 41–45.

[59] Nadine Rücker and Andreas Maier. 2022. FlexParser—The adaptive log file parser for continuous results in a changing world. *Journal of Software: Evolution and Process* 34, 3 (2022), e2426.

[60] Issam Sedki, Abdelwahab Hamou-Lhadj, Otmane Ait-Mohamed, and Mohammed A Shehab. 2022. An effective approach for parsing large log files. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 1–12.

[61] 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 *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 402–411.

[62] Keiichi Shima. 2016. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213* (2016).

[63] John Stearley. 2004. Towards informatic analysis of syslogs. In *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935)*. IEEE, 309–318.

[64] Hudan Studiawan, Ferdous Sohel, and Christian Payne. 2020. Automatic event log abstraction to support forensic investigation. In *Proceedings of the Australasian computer science week multiconference*. 1–9.

[65] Yicheng Sui, Yuzhe Zhang, Jianjun Sun, Ting Xu, Shenglin Zhang, Zhengdan Li, Yongqian Sun, Fangrui Guo, Junyu Shen, Yuzhi Zhang, Dan Pei, Xiao Yang, and Li Yu. 2023. LogKG: Log Failure Diagnosis Through Knowledge Graph. *IEEE Transactions on Services Computing* 16, 5 (2023), 3493–3507. https://doi.org/10.1109/TSC.2023.3293890

[66] Narate Taerat, Jim Brandt, Ann Gentile, Matthew Wong, and Chokchai Leangsuksun. 2011. Baler: deterministic, lossless log message clustering tool. *Computer Science-Research and Development* 26, 3 (2011), 285–295.

[67] Byung-Chul Tak, Shu Tao, Lin Yang, Chao Zhu, and Yaoping Ruan. 2016. LOGAN: Problem Diagnosis in the Cloud Using Log-Based Reference Models. In *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*. IEEE Computer Society, 62–67. https://doi.org/10.1109/IC2E.2016.12

[68] Liang Tang and Tao Li. 2010. LogTree: A Framework for Generating System Events from Raw Textual Logs. In *2010 IEEE International Conference on Data Mining*. 491–500. https://doi.org/10.1109/ICDM.2010.76

[69] 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*. 785–794.

[70] Shimin Tao, Weibin Meng, Yimeng Cheng, Yichen Zhu, Ying Liu, Chunning Du, Tao Han, Yongpeng Zhao, Xiangguang Wang, and Hao Yang. 2022. Logstamp: Automatic online log parsing based on sequence labelling. *ACM SIGMETRICS Performance Evaluation Review* 49, 4 (2022), 93–98.

[71] Stefan Thaler, Vlado Menkonvski, and Milan Petkovic. 2017. Towards a neural language model for signature extraction from forensic logs. In *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*. IEEE, 1–6.

[72] Daniel Tovarňák and Tomáš Pitner. 2019. Normalization of unstructured log data into streams of structured event objects. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 671–676.

[73] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. Ieee, 119–126.

[74] Risto Vaarandi. 2004. A breadth-first algorithm for mining frequent patterns from event logs. In *International Conference on Intelligence in Communication Systems*. Springer, 293–308.

[75] Arthur Vervaet, Raja Chiky, and Mar Callau-Zori. 2021. USTEP: Unfixed search tree for efficient log parsing. In *2021 IEEE international conference on data mining (ICDM)*. IEEE, 659–668.

[76] Xiaodong Wang, Yining Zhao, Haili Xiao, Xiaoning Wang, and Xuebin Chi. 2021. Ltmatch: A method to abstract pattern from unstructured log. *Applied Sciences* 11, 11 (2021), 5302.

[77] Pengyu Wen, Zhizhong Zhang, and Bingguang Deng. 2020. OLMPT: Research on online log parsing method based on prefix tree. In *Proceedings of the 3rd International Conference on Information Technologies and Electrical Engineering*. 55–59.

[78] Markus Wurzenberger, Max Landauer, Florian Skopik, and Wolfgang Kastner. 2019. Aecid-pg: A tree-based log parser generator to enable log analysis. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 7–12.

[79] Tong Xiao, Zhe Quan, Zhi-Jie Wang, Kaiqi Zhao, and Xiangke Liao. 2020. LPV: A Log Parser Based on Vectorization for Offline and Online Log Parsing. In *2020 IEEE International Conference on Data Mining (ICDM)*. 1346–1351. https://doi.org/10.1109/ICDM50108.2020.00175

[80] Yi Xiao, Van-Hoang Le, and Hongyu Zhang. 2024. Stronger, Faster, and Cheaper Log Parsing with LLMs. *arXiv preprint arXiv:2406.06156* (2024).

[81] Junjielong Xu, Qiuai Fu, Zhouruixing Zhu, Yutong Cheng, Zhijing Li, Yuchi Ma, and Pinjia He. 2023. Hue: A User-Adaptive Parser for Hybrid Logs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 413–424. https://doi.org/10.1145/3611643.3616260

[82] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2024. DivLog: Log Parsing with Prompt Enhanced In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[83] 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*. 117–132.

[84] Wei Xu, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. 2010. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, Johannes Fürnkranz and Thorsten Joachims (Eds.). Omnipress, 37–46. https://icml.cc/Conferences/2010/papers/902.pdf

[85] Jing Ya, Tingwen Liu, Haoliang Zhang, Jinqiao Shi, and Li Guo. 2015. An automatic approach to extract the formats of network and security log messages. In *MILCOM 2015 - 2015 IEEE Military Communications Conference*. 1542–1547. https://doi.org/10.1109/MILCOM.2015.7357664

[86] Ruipeng Yang, Dan Qu, Yekui Qian, Yusheng Dai, and Shaowei Zhu. 2019. An online log template extraction method based on hierarchical clustering. *EURASIP Journal on Wireless Communications and Networking* 2019, 1 (2019), 135.

[87] Kundi Yao, Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2020. A study of the performance of general compressors on log files. *Empirical Software Engineering* 25, 5 (2020), 3043–3085. https://doi.org/10.1007/s10664-020-09822-x

[88] Kundi Yao, Mohammed Sayagh, Weiyi Shang, and Ahmed E. Hassan. 2022. Improving State-of-the-Art Compression Techniques for Log Management Tools. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2748–2760. https://doi.org/10.1109/TSE.2021.3069958

[89] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, James C. Hoe and Vikram S. Adve (Eds.). ACM, 143–154. https://doi.org/10.1145/1736020.1736038

[90] Shenglin Zhang, Ying Liu, Weibin Meng, Jiahao Bu, Sen Yang, Yongqian Sun, Dan Pei, Jun Xu, Yuzhi Zhang, Lei Song, et al. 2020. Efficient and robust syslog parsing for network devices in datacenter networks. *IEEE access* 8 (2020), 30245–30261.

[91] Shenglin Zhang, Weibin Meng, Jiahao Bu, Sen Yang, Ying Liu, Dan Pei, Jun Xu, Yu Chen, Hui Dong, Xianping Qu, et al. 2017. Syslog processing for switch failure diagnosis and prediction in datacenter networks. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.

[92] Tianzhu Zhang, Han Qiu, Gabriele Castellano, Myriana Rifai, Chung Shue Chen, and Fabio Pianese. 2023. System Log Parsing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 35, 8 (2023), 8596–8614. https://doi.org/10.1109/TKDE.2022.3222417

[93] Yining Zhao, Xiaodong Wang, Haili Xiao, and Xuebin Chi. 2018. Improvement of the log pattern extracting algorithm using text similarity. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 507–514.

[94] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. 2019. An Exploratory Study of Logging Configuration Practice in Java. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 459–469. https://doi.org/10.1109/ICSME.2019.00079

[95] Ya Zhong, Yuanbo Guo, and Chunhui Liu. 2018. FLP: a feature-based method for log parsing. *Electronics letters* 54, 23 (2018), 1334–1336.

[96] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. 2023. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. arXiv:2008.06448 [cs.SE] https://arxiv.org/abs/2008.06448

[97] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.

[98] Kenny Q Zhu, Kathleen Fisher, and David Walker. 2010. Incremental learning of system log formats. *ACM SIGOPS Operating Systems Review* 44, 1 (2010), 85–90.

[99] De-Qing Zou, Hao Qin, and Hai Jin. 2016. Uilog: Improving log-based fault diagnosis by log analysis. *Journal of computer science and technology* 31, 5 (2016), 1038–1052.

[100] Farhana Zulkernine, Patrick Martin, Wendy Powley, Sima Soltani, Serge Mankovskii, and Mark Addleman. 2013. Capri: A tool for mining complex line patterns in large log data. In *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*. 47–54.