PDF Download
3803019.pdf
26 March 2026
Total Citations: 0
Total Downloads: 0

Latest updates: https://dl.acm.org/doi/10.1145/3803019

Published: 25 March 2026
Accepted: 02 March 2026
Revised: 23 December 2025
Received: 10 September 2025

Citation in BibTeX format

RESEARCH-ARTICLE

Not-So-Pretty: Studying and Segmenting Multiline Console Logs

JIANCHEN ZHAO

KUNDI YAO

MASANARI KONDO

HETONG DAI

WEIYI SHANG

YASUTAKA KAMEI

ACM Transactions on Software Engineering and Methodology
https://doi.org/10.1145/3803019
EISSN: 1557-7392

Not-So-Pretty: Studying and Segmenting Multiline Console Logs

JIANCHEN ZHAO, University of Waterloo, Canada
KUNDI YAO, University of Waterloo, Canada
MASANARI KONDO, Kyushu University, Japan
HETONG DAI, University of Waterloo, Canada
WEIYI SHANG, University of Waterloo, Canada
YASUTAKA KAMEI, Kyushu University, Japan

Console logs play a crucial role in the development, maintenance, and operation of software systems. However, their diverse and human-oriented formatting presents significant challenges for automated analysis. Most existing log analysis benchmarks and the tools built upon them operate under the assumption that individual log messages correspond to single lines. In reality, console logs frequently span multiple lines and exhibit a wide range of structures and formats, complicating parsing and downstream analysis. In this paper, we conduct a qualitative study on a diverse dataset of console logs, identifying four structural types and ten common formats in multiline log segments. We find that although only 3% of the log messages are multiline, they account for 66% of the content by line count. To address the challenge of parsing log messages that span multiple lines, we also propose a deep learning–based preprocessing approach that splits logs into structurally and semantically consistent segments and assigns labels about their format. Our model, trained on a manually labelled dataset, achieves an F1 score of 0.88 for log segmentation and 0.84 for log format classification. We release our dataset and model to support further research in log analysis.

CCS Concepts: • **Software and its engineering** → **Maintaining software**; **Software maintenance tools**; • **Information systems** → *Clustering and classification*; • **Applied computing** → Format and notation; Annotation; • **Computing methodologies** → Neural networks.

Additional Key Words and Phrases: Console Logs, Log Segmentation, Log Format Classification, Log Preprocessing, Language Models, Deep Learning

1 Introduction

Software logs are a ubiquitous source of information for various software system development and operation-related activities. They provide valuable insights into system behavior, underpinning essential software engineering tasks such as log-based anomaly detection [2], failure prediction [20], performance monitoring [28], and automated testing [25]. As modern systems increase in complexity, so does the volume and diversity of logs, making automated log analysis tools indispensable [9, 14].

A crucial step in automated log analysis is log preprocessing, such as log parsing, which transforms raw log data into structured formats suitable for downstream analysis. However, existing log analysis benchmarks, such

Authors' Contact Information: Jianchen Zhao, jianchen.zhao@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Kundi Yao, kundi.yao@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Masanari Kondo, kondo@ait.kyushu-u.ac.jp, Kyushu University, Fukuoka, Fukuoka, Japan; Hetong Dai, h5dai@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Weiyi Shang, wshang@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Yasutaka Kamei, kamei@ait.kyushu-u.ac.jp, Kyushu University, Fukuoka, Fukuoka, Japan.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7392/2026/3-ART
https://doi.org/10.1145/3803019

Just Accepted

as LogHub [33], are composed exclusively of well-structured single-line logs. These benchmarks have shaped the development of state-of-the-art log preprocessing and analysis tools that operate under the assumption that logs are composed of single-line events. Although this assumption simplifies log analysis, it fails to account for the diversity and complexity of real-world logs.

In practice, logs are not always presented in a pretty and well-behaved single-line format; instead, they often contain multiline segments with varying structures and complexities. For example, continuous integration (CI) logs frequently include blocks of self-contained log messages that span multiple lines, such as tabular outputs, formatted reports, and pretty-printed data structures. Other examples include stack traces, recursive structures, and progress bars. These multiline logs are notably absent from commonly used benchmarks like LogHub, making it challenging for tools built on single-line assumptions to handle real-world log scenarios effectively.

The lack of support for multiline logs in existing log parsing tools and benchmarks presents significant challenges. First, logs must be segmented into meaningful chunks to capture their structure. Second, the data type of each chunk must be identified to apply the appropriate preprocessing rules. Without addressing these challenges, important information in multiline logs can be lost or misinterpreted, leading to suboptimal analysis results. To understand these challenges, we conducted a qualitative study on a diverse dataset of console logs and answered the following research questions:

RQ. 1: How are multiline log segments structured in console logs?
Motivation: The way information is organized differs from log to log. In multiline log segments, each line carries part of the information. By answering this question, we aim to understand how these pieces of information are assembled together and whether log templates remain viable for multiline logs.
Results: We found four different types of structures.

RQ. 2: How are multiline log segments formatted in console logs?
Motivation: Unlike structures, the format of a log is a finer-grained categorization of the way data is presented. Multiline logs, because they are generally longer than single-line logs, may have more complex formats that existing log parsers cannot parse. Answering this question allows us to identify challenges for log parsers due to the presentation of information in multiline logs.
Results: We found ten different formats.

RQ. 3: How prevalent are multiline log segments in console logs?
Motivation: It is unclear whether the proportion of multiline logs warrants any precautions from a log preprocessing perspective compared to conventional single-line logs.
Results: Although we found that only 3% of console logs span multiple lines, they represent 66% of the total line count.

Building on the insights from our qualitative study, we propose an automated approach that segments logs into coherent chunks and classifies the data type of each chunk. Specifically, we train a deep learning model on a large dataset of console logs and then fine-tune it to split console logs into individual segments and classify the format of the segments. We evaluate our model by answering the following research question:

RQ. 4: How accurately can we automatically identify multiline logs and their structures?
Motivation: In particular, we evaluated our approach from two perspectives. How accurately can our approach segment console logs? And how accurately can our model learn to identify the different types of structure and formats present in console logs?
Results: Our approach achieves F1-scores of 0.88 in log segmentation and 0.84 in format classification (weighted average across all formats).

Using this automatic log segmentation approach, we quantify the impact of multiline log segmentation on log parsing. We answer the following research question:

RQ. 5: What is the impact of multiline log segments on log parsing?

Just Accepted

Motivation: Due to the complex structures of multiline logs, templates generated by existing log parsers for each line of multiline logs may have unintended side effects on the parsing of single-line logs. Templates of individual lines of multiline logs may be overspecific or overgeneralized.
Results: We found that the presence of multiline logs can negatively impact log parsing performance by causing log parsers to produce more overgeneric and overspecific templates. When adapting the logs by aggregating multiline segments, the impact is reduced.

Contributions. Our main contributions are as follows:

(1) We conducted, to the best of our knowledge, the first qualitative study of multiline segments in console logs.
(2) We identified the structures and potential formats present in multiline segments of logs and determined their prevalence in console logs.
(3) We propose a baseline, deep learning based automated console log segmentation and format classification approach.
(4) We quantified the impact of multiline log segments on log parsing.
(5) We publicly release our fine-tuned segmentation model as well as a manually labeled dataset of 480 log segments, including both single-line and multiline logs.[1]

Our work enables practical access to the rich information in console logs, which previously had not been fully utilized in log analysis during software engineering activities.
Paper organization. The remainder of the paper is organized as follows. In Section 2, we present the background of log preprocessing and multiline logs. In Section 3, we describe our qualitative study on multiline log segments in console logs. In Section 4, we describe our automated approach to segmenting and classifying multiline logs. In Section 5, we describe how we quantified and measured the impact of multiline log segments on log parsing. In Section 6, we discuss the implications of our studies, and in Section 7, the threats to the validity of our findings. In Section 8, we present the research related to this work. Finally, we conclude the paper in Section 9.

2 Log Preprocessing and Multiline Logs

Logs are most useful when they are abstracted into a dense representation [14, 18, 19]. In recent literature, many have proposed a variety of log parsers [18] to automatically process logs into abstract representations. Fig. 1 shows the log parsing process: log lines are processed into a log template, where the variable parts of the logs are replaced by wildcards (<*>). A log template uniquely identifies a type of event. As such, logs can be viewed as actual instances of events, with the variable parts being dynamic information conveyed in each instance. To convert logs into events, pattern-matching expressions, such as RegEx, are constructed from the log templates. Each log is matched against the most fitting template, after which an event ID representing the type of event can be assigned. Event IDs, along with the dynamic information extracted from each log line, are stored and used for downstream log analysis tasks.

Despite the variety of log preprocessing approaches that have been proposed, most are based on the assumption that each log message is self-contained within a single line, delimited by `\n` or `\r\n` characters. However, this single-line assumption does not hold for many real-world scenarios, especially in automation systems such as continuous integration (CI) and continuous deployment (CD) pipelines. In these scenarios, logs are often generated by a diverse set of third-party console applications, each with its own output style and structure. As a result, individual log events produced by console applications can span a varying number of lines, exhibit a wide range of formats, and include elements such as ANSI escape sequences for user interactivity (note that logs are not interactive and escape sequences only interfere with encoding when saved to a file), natural language

[1] The models and the manually labeled dataset are available at https://doi.org/10.5281/zenodo.15562249

Just Accepted

Fig. 1. An example of a log parsing process. The log on the top is parsed into the log template of the bottom.

Listing 1 An example of two multiline console log snippets.

```
1  phpenv versions
2    system
3    5.6
4    5.6.40
5    7.1
6    7.1.27
7    7.2
8  * 7.2.15 (set by /home/travis/.phpenv/version)
9    hhvm
10   hhvm-stable
11
12 The command "docker run -v \$PWD:/root/src/openface bamos/openface \
13        /bin/bash -l -c \
14        "source /root/torch/install/bin/torch-activate; \
15        cd /root/src/openface; \
16        ./models/get-models.sh && \
17        ./data/download-lfw-subset.sh && \
18        wget -nv http://openface-models.storage.cmusatyalab.org/nn4.v1.t7 \
19             -O ./models/openface/nn4.v1.t7 && \
20        python2 setup.py install && \
21        ./run-tests.sh"
22 " exited with 255.
```

Just Accepted

paragraphs, new-line separated lists, structured object representation, variable dumps, nested log files, code fragments, progress bars, etc. We refer to such logs as **multiline console logs**. More specifically, we define a multiline log segment as **the smallest sequence of lines sharing a consistent format and pertaining to the same event.**

Without consistent formats and clear delimitation, multiline logs are difficult to parse. Listing 1 contains examples of multiline log snippets emitted during a CI pipeline run taken from the LogChunks [1] dataset. In the first snippet (lines 1 to 10), each line corresponds to one dynamic variable; if parsed individually, the produced templates may overlap with similar-looking but semantically different events, since there are no static texts to differentiate them, thus impacting the usefulness of the parsing results. In the second snippet (lines 12 to 22), it is difficult to determine what is static and what is dynamic; one might even conclude that such messages cannot be parsed into templates, adding uncertainty to the correctness of the parsing.

To demonstrate the limitations of log parsers on multiline logs, we use Drain [8], a popular log parser, to parse the entire LogChunks dataset. We then attempt to parse the snippet in Listing 1. We present the results in Listing 2. The versions in the log snippet (lines 1 to 10) each have their own event ID and template (with no dynamic value except for event 101), which defeats the purpose of parsing them in the first place. Similarly, the templates produced for lines 12 to 22 either contain no dynamic variables (represented by <*>) or are very similar and only differ in the number of dynamic variables. In total, Drain found 72,464 different log templates in

Listing 2 Event ID and log templates obtained by parsing the snippet in Listing 1 with Drain3.

```
1    8201 phpenv versions
2      95 system
3      96 5.6
4      97 5.6.40
5      98 7.1
6      99 7.1.27
7     100 7.2
8     101 * <*> (set by /home/travis/.phpenv/version)
9     102 hhvm
10    103 hhvm-stable
11
12  30863 The command "docker run -v <*> bamos/openface \
13  30829 /bin/bash -l -c \
14  30830 "source /root/torch/install/bin/torch-activate; \
15   4905 cd <*> \
16  30831 <*> && \
17  30831 <*> && \
18  30832 wget -nv <*> \
19  22562 <*> <*> && \
20  22564 <*> <*> <*> && \
21  30833 ./run-tests.sh"
22   4219 " exited with <*>
```

the LogChunks datasets, which is a large count compared to LogHub [13] (3,488 templates), while not having as many log lines. These examples show that the templates obtained by directly applying log parsers are not satisfactory for log files containing multiline segments.

Despite these challenges, extracting structured information from multiline logs is crucial. In the previous examples, the version information in the first snippet is only meaningful when all related lines are processed together as a cohesive unit, while the complex command structure in the second snippet demonstrates that multiline logs exhibit fundamentally different characteristics from typical system event logs. These observations lead to two key insights:

1) **Multiline logs should be processed together to preserve event boundaries and maintain semantic coherence.**
2) **Multiline logs require specialized processing approaches that account for their unique structures and formats.**

We propose to address these challenges by introducing a segmentation step that groups related lines into coherent events, and a classification step that identifies the structural characteristics of each segment to enable appropriate downstream processing.

Just Accepted

3 Qualitative Study of Multiline Logs in Console Logs

We first conducted a qualitative study of multiline logs found in console logs to understand the structures and data formats they contain through a manual coding process. We identify two interesting dimensions to the shape of multiline log segments. On the one hand, "structure" designates the general ways in which information is inherently organized. On the other hand, "format" dictates how the data is presented, including, for example, the use of marker words.

3.1 Dataset

We chose the LogChunks [1] dataset for two main reasons. **Relevance:** While not all console logs are relevant for software engineering (SE), CI pipeline logs are direct artifacts of automated SE tasks; the analysis of these

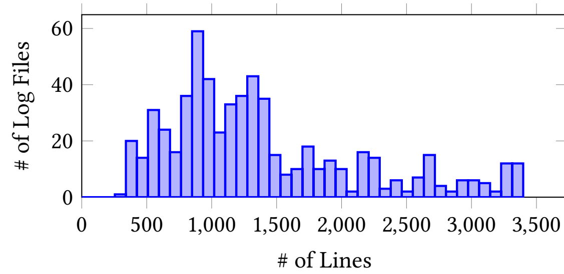


Fig. 2. Distribution of the number of lines in each log file in the LogChunks dataset. Outliers are omitted.

Just Accepted

logs further enables SE-related downstream activities. **Generalizability:** CI pipeline logs are produced by a wide range of console applications and scripts and are generated from realistic workflows, and thus our findings can be generalized to other CI logs. LogChunks is particularly interesting for our study, since it contains a relatively large proportion of logs related to failures, which are normally less common but hold important information for log analysis. These logs may contain different structures because the nature of the information differs from normal logs. Projects based on different programming languages require different tools for their build processes. The logs in this dataset are generated not only from common tools, but also from language-specific ones.

LogChunks is a dataset of build logs extracted from Travis CI pipelines of 80 GitHub projects. It contains 797 log files of failed builds from 80 open-source projects, spanning 29 main programming languages. On average, there are 3,283 lines of logs per file and 32,704 lines per project. As shown in Fig. 2, the log files have different total numbers of log lines, giving us a diverse dataset. The three projects with log files with the largest volume of logs are large applications: *ejabberd* (an XMPP server), *php-src* (the interpreter for the PHP language), and *nginx* proxy (a Docker container running the Nginx proxy server).

An important detail about this dataset is that timestamps are not always available. Although log aggregation systems can automatically generate timestamps for each line of logs, they are not always present due to different configurations and may not be accurate due to buffering of the logs. In LogChunks, certain log segments come with their own timestamps in various formats. However, for most of the logs, timestamps are not recorded. For these reasons, we do not consider timestamps in our approach.

Because many applications running in CI pipelines are console applications that are often designed with human users in mind, the emitted logs may contain console escape sequences that are normally used to improve readability and user interactions. However, for automated log analysis, we consider them as noise as they contain invisible characters and are not semantically useful. We remove such invisible console escape sequences from the data with simple regular expressions (details shared in our replication package). We also find that some files had extra empty space between each line. Thus, for consistency with normal log files, we remove all empty lines. Additionally, we also normalize line endings to Unix-style line endings.

3.2 Approach

In this section, we present our approach to the qualitative study we conducted on multiline segments in console logs, which we also outline in Fig. 3a. After an initial data exploration, we sampled the dataset, formulated a taxonomy through many rounds of labeling and discussion, and finally manually labeled the entire sample.

Step 1: Initial Data Exploration. To obtain an initial idea of the structures and formats of multiline log segments in the build log files, we manually explored the dataset. We first randomly picked one log file from each of the 80 projects in the dataset. We then read through each of the files, identifying each multiline log segment

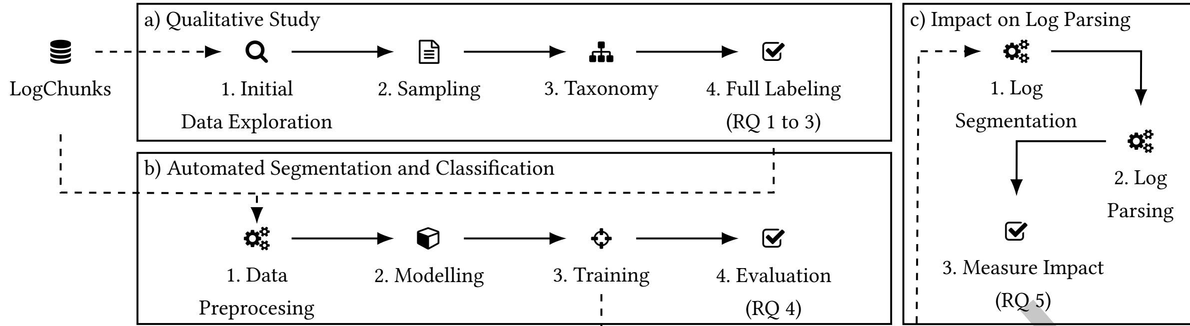


Fig. 3. Overview of our study approach.

encountered. We assigned a meaningful label to the format from a set of labels. When a new, distinct format is encountered, we add it to the set of labels. However, we note that these initial labels serve only as a basis for the manual labeling effort in the next steps and are improved during the manual coding process (step 3) through multiple rounds of labeling and discussions.

From this initial manual exploration, we found that there is no clear line between all formats. In discussion sessions about the initial set of labels, we found that the exact line where a log message starts and ends is subjective and depends largely on the perspective of the person reading the logs. Depending on how precisely the format is defined, more formats may be needed to code all multiline logs. To resolve this issue, we adopted a definition that is suitable in the context of log parsing to guide our qualitative analysis: **A multiline log segment is the smallest sequence of lines sharing a consistent format and pertaining to the same event.** An event can only contain the information of a single state of the system at a specific time. However, because we do not consider timestamps in our analysis, we only used the semantics of the log messages to distinguish events.

Step 2: Sampling. We sampled individual lines from each log file. To ensure that the samples are uniform, independent, and nonoverlapping, we sampled only one line per log file. This is because if we uniformly sample more than one line in the same log file, those lines could be part of the same multiline log. By only sampling one line in each log file, we ensure that there is no overlap, and each multiline log we encounter in the coding process is only ever counted once.

Step 3: Finding a Taxonomy. We adopted a manual coding approach similar to [3]. Fig. 4 shows our manual labeling process. We proceeded as follows: First, we tasked two authors with finding an appropriate taxonomy from the initial set of labels as a basis. In multiple rounds, the two authors individually inspected 20 randomly sampled lines in each round. For each line, the authors determined whether it is part of a multiline log segment. In the case where it was, they found the starting and ending line numbers and assigned a label about the structure type and a label about the format from the set of labels. In the other case, where a line is not part of a multiline segment, the authors marked it as a single-line log. After 20 examples had been labeled, the two authors compared their results and computed an agreement score for the format labels. Since the structure type labels are dependent on the formats, there is no need to compute the agreement for the structure types. If the agreement is under a predetermined threshold, the two authors continued to label another 20 examples to refine the set of labels. When the agreement threshold is exceeded, we proceed to the next step.

We used Cohen's Kappa agreement score [4] and a threshold, to be conservative, of 0.8, which can be interpreted as almost perfect [15]. The labeling spanned three rounds and ended with an agreement of 0.86. Later, two additional authors joined the labeling effort, so we proceeded with another round of 20 examples. The four authors achieved a Fleiss' Kappa coefficient [6] of 0.59. Given the increased difficulty of achieving unanimous

Just Accepted

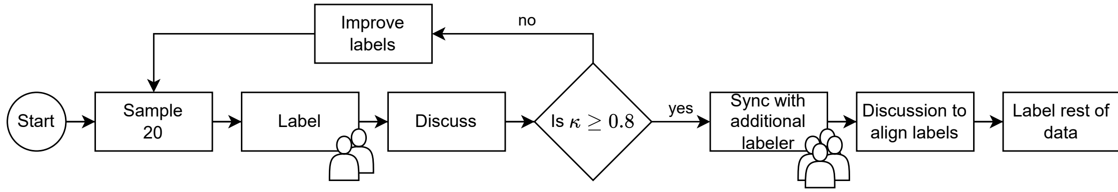


Fig. 4. Flowchart of the manual labeling process.

Table 1. Overview of the structures in multiline logs. Examples are taken from LogChunks [1].

Structures	Example
Simple Simple structure where the format doesn't repeat.	[C#/NancyFx@Nancy/failed/377324945.log#L95:98] Packer v1.0.2 Your version of Packer is out of date! The latest version is 1.1.2. You can update by downloading from www.packer.io
Sequential Structure in which each line has the same format, such as list of items.	[PowerShell/pester@Pester/failed/562381289.log#L386:396] [-] - mock defined in beforeall is counted independently -> Expected $true but got 'False'. at Verify-True, /Users/travis/build/pester/Pester/Dependencies/Axiom/ Verify-True.ps1:8 at <ScriptBlock>, /Users/travis/build/pester/Pester/new-runtimepoc/ Pester.Mock.RSpec.ts.ps1:95 ... at <ScriptBlock>, <No file>:1 at <ScriptBlock>, <No file>:1
Tabular Structure in which lines are rows of items. It can have a header.	[C#/NancyFx@Nancy/failed/377324945.log#L286:314] Huge version without GUI. Features included (+) or not (-): +acl +farsi +mouse_netterm +syntax +arabic +file_in_path +mouse_sgr +tag_binary +autocmd +find_in_path -mouse_sysmouse +tag_old_static ...
Recursive Structure in which elements are formatted recursively, such as for nested object representations.	[Haskell/jgm@pandoc/failed/480326509.log#L1138:1249] [(DefiniteUnitId (DefUnitId {unDefUnitId = UnitId "base-4.8.2.0-0d6d1084fbc041e1cded9228e80e264d"}),DefaultRenaming),(DefiniteUnitId ... (DefUnitId {unDefUnitId = UnitId "file-embed-0.0.11-5eblKl2yNd7K74FblwfEQk"}),DefaultRenaming)]

Just Accepted

agreement with four labelers on 10 classes compared to a two-rater two-class scenario, in which case 0.59 borderlines moderate and substantial agreement, we consider our agreement to be meaningfully consistent beyond chance. Then, we held a discussion session to correct the labels where the authors disagreed.

Step 4: labeling the Rest of the Data. We split the four authors into two groups of two. We tasked each group with labeling a different set of randomly selected 200 examples, for a total of 400 examples. This sample size is chosen based on the estimated labeling effort. In each group, the authors proceeded individually. During the labeling process, we also ensure that each label is reviewed by at least two authors to avoid bias and to guarantee the quality of the labels. After labeling, a discussion is held to compare the results and align the labels where there were differences. Then, we aggregated the results of the two groups into a single dataset. In total, the authors labeled a total of 480 examples.

3.3 Results

RQ. 1: How are multiline log segments structured in console logs? **After manual coding, we identified four types of structure in multiline logs.** We present an overview of the types and examples for each in Table 1. We differentiate types of structure by how the information is organized structurally.

- **Simple.** A structure that is not sequential, tabular, or recursive is a simple structure. That is, the information is not in a format that repeats or recurses. Usually, as shown in the example in Table 1, simple structures tend to be a paragraph of text that spans multiple lines. If lines in a multiline log do not share any noticeably similar formats, they are considered to have a simple structure. It is possible to extract important information using a regular expression, but constructing a log template may not always be possible.
- **Sequential.** In this type of structure, information is organized into smaller but similar parts where their formats are repeated. We make an important distinction between sequential structures in multiline logs and repeated single-line logs. In sequential multiline logs, each line is difficult to parse on its own, e.g., because they only contain dynamic values. For consistency, sequences of lines sharing the same format and part of the same event are considered to be a sequential multiline log. Repeated single-line logs must have a more distinct format that does not break any line boundaries and must be part of different events of the system. A parser can iterate through each line to parse them individually, while in the context of the multiline log.
- **Tabular.** In this type of structure, similar to sequential structures, information is organized into smaller items, but in a grid. The grid uses whitespace characters to separate items in the same row, so that the items align in each column. However, sometimes the formatting of the grid can break due to a wrong assumption about the length of items, and columns fail to align. Although we consider grid-like structures to be their own type, the line between sequential and tabular types is blurry. Consider the sequential example in Table 1: each repeated line has a specific format and contains two dynamic values. It is entirely possible to consider this as a table of two columns, and the column separator would be the comma, and `at` would serve as a line prefix in the format. In the tabular example, each item can be considered part of a list, and thus be considered to be a sequential structure semantically. To make the distinction clear, items in a tabular structure can not use words as delimiters, prefixes, or suffixes; only symbols or whitespace can be used.
- **Recursive.** In this type of structure, information is organized in nested structures. While both sequences and tables can be constructed recursively, we make a distinction for specifically nested structures that cannot be described by a regular grammar. This type of structure particularly poses a problem to log parsers. Conventionally, log parsers produce log templates, which only use wildcards as placeholders for dynamic values. Log templates can encode at most a regular grammar. However, it is not clear whether it is needed to parse a recursive multiline log: a regular expression can be used to extract important parts only and avoid the need for a specific parser.

> **Finding 1.**
> There are four distinct types of structure in which multiline information can be organized. Although there might be some overlap, each type has different requirements for a parser. A log template may not be able to recognize these structures.

Just Accepted

RQ. 2: How are multiline log segments formatted in console logs? **We find 10 different multiline formats**, listed in Table 2 with an example for each. We found that formats were especially difficult to formulate because there are many variations for each format, and there are overlaps due to the variations. To balance the accuracy of the format labels and the number of them, we formulated the format such that each contains enough information to allow them to be parsed with a generic parser. We describe the formats as follows:

- **Code.** This format encompasses fragments of any programming language code in the logs. It is impossible to parse all possible code fragments using a single generic parser, due to the complexity of the languages and

Table 2. Overview of the formats in multiline logs. Examples are taken from LogChunks [1].

<table>
<tr><th>Structures</th><th>Example</th></tr>
<tr><td>Code
Code fragment that has been directly included.</td><td>[C/vim@vim/failed/555378073.log#L790:792]
$ if [["$(which python3)" =~ ^/opt/python/]]; then
export PATH=$(py3=$(which python3); echo $\{PATH//$\{py3%/python3
\}:/\})
fi</td></tr>
<tr><td>Diff Hunk
The output of the <code>diff</code> command. A hunk is a file fragment containing a change.</td><td>[C/php@php-src/failed/529279089.log#L1913:1916]
001+ Warning: PDOStatement::fetch(): SQLSTATE[HY000]: ...
001- done!
002+ [000 + 1] Expecting array('one' => 1), got false
003+</td></tr>
<tr><td>Paragraph
Paragraph containing long messages.</td><td>[C#/NancyFx@Nancy/failed/377324945.log#L95:98]
Packer v1.0.2
Your version of Packer is out of date! The latest version
is 1.1.2. You can update by downloading from www.packer.io</td></tr>
<tr><td>Stack Trace
A stack trace, often includes error message, source location and relevant code.</td><td>[Haskell/purescript@purescript/failed/549434310.log#L2469:2475]
happy > ...: warning: [-Wdeprecations]
happy > In the use of `rawSystemProgramConf'
happy > (imported from Distribution.Simple.Program):
happy > Deprecated: "use runDbProgram instead. ...
happy > |
happy > 48 | let runProgram p = rawSystemProgramConf (.
happy > | ^^^^^^^^^^^^^^^^^^^^</td></tr>
<tr><td>Items
Single word or row of words, of individual values, part of a list or table.</td><td>[C#/NancyFx@Nancy/failed/433641108.log#L420:423]
Pre-installed Ruby versions
ruby-2.2.7
ruby-2.3.4
ruby-2.4.1</td></tr>
<tr><td>Progress Bar
Interactive progress bar, duplicated in log files.</td><td>[C/vim@vim/failed/555378073.log#L815]
61% | | 512kB 40.3MB/s eta 0:00:01
62% | | 522kB 40.1MB/s eta 0:00:01
64% | | 532kB 41.8MB/s eta 0:00:01</td></tr>
<tr><td>Variable Assignments
List of variable assignments often in the format <code>variable=value</code>.</td><td>[VimL/mhinz@vim-galore/failed/374778890.log#L439:441]
GORACE=""
GOROOT="/home/travis/.gimme/versions/go1.8.linux.amd64"
GOTOOLDIR="/home/travis/.gimme/versions/go1.8.linux.amd64/pkg/tool/
linux_amd64"</td></tr>
<tr><td>JSON Like
Nested Object representation similar to JSON, where brackets <code>{}</code> and <code>[]</code> surround objects.</td><td>[CSS/twbs@bootstrap/failed/566160487.log#L566:863]
{ Error: Request failed with status code 500
at createError (...)
at settle (...)
...
[Symbol(isCorked)]: false,
[Symbol(outHeadersKey)]: [Object] },
data:
... },
isAxiosError: true,
toJSON: [Function] }</td></tr>
<tr><td>Tree Like
Nested Directory listing, similar to the output of the <code>tree</code> command.</td><td>[C#/powershell@PowerShell/failed/535985298.log#L688:761]
/home/travis/.nvm/versions/node/v6.4.0/lib
└─┬ markdown-spellcheck@0.11.0
├── async@1.5.2
├─┬ chalk@1.1.3
...
│ └── unique-concat@0.2.2
└── native-promise-only@0.8.1</td></tr>
<tr><td>YAML Like
Nested Object representation similar to YAML, where a colon character (<code>:</code>) is used to denote values.</td><td>[C#/NancyFx@Nancy/failed/377324945.log#L34:44]
Client:
Version: 17.09.0-ce
API version: 1.32
...
Server:
Version: 17.09.0-ce
API version: 1.32 (minimum version 1.12)</td></tr>
</table>

Just Accepted

their recursive nature. Thus, if there is important information, regular expressions or language models can be used to extract it and forgo the construction of a log template.

- **Diff Hunk.** This format encodes the difference between two corpora of text. The format is similar to that of the output of the `diff` command. Usually, in this format, each line starts with a `+` or `-`, optionally followed by a line number, indicating whether the line has been removed or added; the modified text follows. The `+` or `-` can be considered static, whereas the line number and the modified text are two dynamic values. Interestingly, diff

hunks may contain important information. In the example in Table 2, the output of a test is "diff'ed" against an expected output; in that case, the hunk contains the reason why the test failed.

- **Paragraph.** This format contains text similar to normal system event logs, with the difference that it spans multiple lines. A regular log parser should be able to parse logs of this format, provided that the boundaries of the multiline segment are known.
- **Stack Trace.** Stack traces may have different formats, but they contain similar elements: file paths, line numbers, code, and symbols pointing to the code element of interest. Different languages or compilers generate different stack traces. Sometimes they contain a single stack frame, and at other times they contain an entire list of stack frames. Log parsers can potentially leverage the repeated frames to discover a pattern automatically.
- **Items.** In this format, the content is simply a single dynamic value. In the dataset we analyzed, they are often version specifiers, as shown in the example in Table 2. Events in this format will share the same log templates, and thus, log parsers will have to rely on the surrounding context to infer the event.
- **Progress Bar.** Progress bars are often left in log files. Normally, they serve as an interactive indicator of a long-running process. However, tools do not offer the option to disable interactivity or are not configured to do so. Progress bars make extensive use of ANSI console escape sequences. Thus, it is necessary to preprocess them. In the console, progress bars should be displayed on the same line, for example, using carriage return characters, but when recorded into log files, all lines are present. When normalizing the log files in our datasets, carriage returns are treated as new lines. Due to the bars having a variable length, log templates may have to include special wildcards to denote them.
- **Variable Assignments.** This is a sequential format where each element in the sequence is composed of two dynamic values: the name of a variable and the value assigned to it. There is a symbol between the two to indicate the assignment action. Similar to the Items format, log templates can not be used to reliably identify the event, and log parsers will have to rely on the surrounding context.
- **JSON-Like.** In this format, a nested object is represented textually using paired brackets to surround each object, depending on its type. The paired brackets are `{}`, `[]`, `()` and `<>`. Symbols such as `=` or `:` can also be present to denote an assignment or naming of a nested object. We note that the format is JSON-like, but does not always conform to the JSON syntax. Log parsers can leverage the symbols to find text in the object that are able to be automatically parsed.
- **Tree-Like.** In this format, each line has an indentation to represent the nesting level of individual items. For example, in Table 2, it can be an output of the `tree` command. The symbols used to draw edges of the "tree" can be ignored and replaced by whitespace. Then, log parsers can automatically construct the log template of each item, while leaving the indentation as is.
- **YAML-Like.** Similar to the Tree-Like format, but in this case, each item is a nested object and is formatted similarly to a YAML syntax. Symbols from the JSON-Like format can be used, with the addition of - to represent items in a sequence.

> **Finding 2.** We identify 10 distinct formats in the log files of the LogChunks dataset. We find that existing log parsers have to be extended and that log templates cannot always be used.

RQ. 3: How prevalent are multiline log segments in console logs? **We find that multiline logs occupy a large proportion of log files while only accounting for a small number of log events.** Of the 480 samples, 318 are multiline logs. The median length of a multiline segment is 60 lines. We can see from Fig. 5 that the length distribution is spread over a wide range of values. Their lengths go as high as 2,677 lines. There are two outliers, one segment of 31,288 lines and another of 31,178 lines; both are sequential segments in the Progress Bar format. We estimate that approximately 66% of the lines in the log files of the LogChunks dataset are part of a multiline

Just Accepted

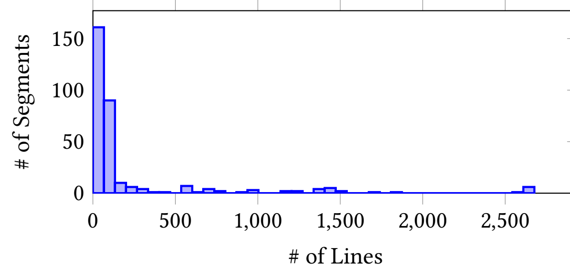
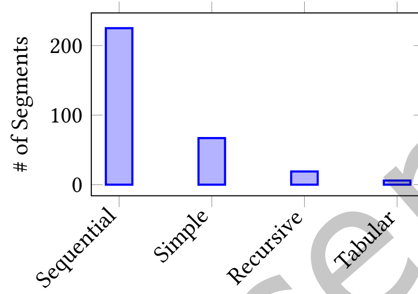
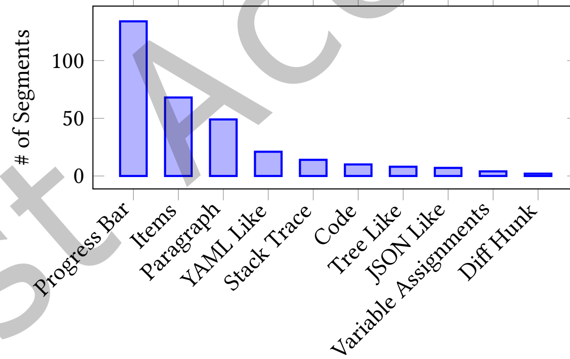


Fig. 5. Distribution of number of lines in multiline log segments. The median number of lines is 60. Outliers are not shown.



(a) Types of Structure



(b) Formats

Fig. 6. Distribution of types of structures and formats we identified.

Just Accepted

log segment. Using the median file lengths and the median multiline log segment length, we find that there are approximately a median of 1,108 single-line logs and 36 multiline logs per file in our dataset. Thus, we estimate that the ratio between single-line logs and multiline logs is around 30 to 1; that is, approximately only 3% of the log events are multiline.

In Fig. 6a we see that **the most common type of structure is the sequential type, with 225 labeled instances.** We find 67 instances of simple structures. Recursive and tabular types are relatively rare, as we only count 19 and 6 instances, respectively.

In Fig. 6b, consistent with the count of structure types, **the most common format is Progress Bar, with 124 instances labeled.** We count 68 instances of Items, 49 instances of Paragraph, and 21 instances of YAML-Like. Other formats are relatively rare, as we count fewer than 15 instances for Stack Trace, Code, Tree-Like, JSON-Like, and Variable Assignments. We only count two instances of Diff Hunk. Initially, in our set of labels, we had Hex Dump as a format, output from the `hexdump` command, but we did not encounter any instances of it in your samples, and thus it is not included in our results.

We show the median length of multiline logs by type of structure and format in Fig. 7a and 7b respectively. While multiline logs with recursive structures are not as prevalent as those with simple and sequential structures, they are longer on average; their median line count is 252. Multiline logs with simple structures are the shortest, with a median of only two lines. The longest format is Tree-Like; its median length is 1,184 lines. This is because logs in this format are usually output from the `tree` command that lists all files in a directory. The second most lengthy format is the Diff Hunk format, with a median length of 477 lines. The content in this format are sometimes files that are completely different, hence the long hunks. For the rest of the formats, the median lengths are between 20 and 100 lines. The YAML-Like and Paragraph formats are particularly short, with a median length of 8 and 2 lines, respectively.

> **Finding 3.** 3% of the console logs span multiple lines, but they account for 66% of the lines in the LogChunks dataset. While some multiline logs with sequential and simple structure types are more common, those with the more complex recursive type are lengthier. They can have lengths from 2 to up to 2,677 lines, with a few exceptions going as high as 31,288 lines.

4 Automated Segmentation and Classification of Multiline Logs

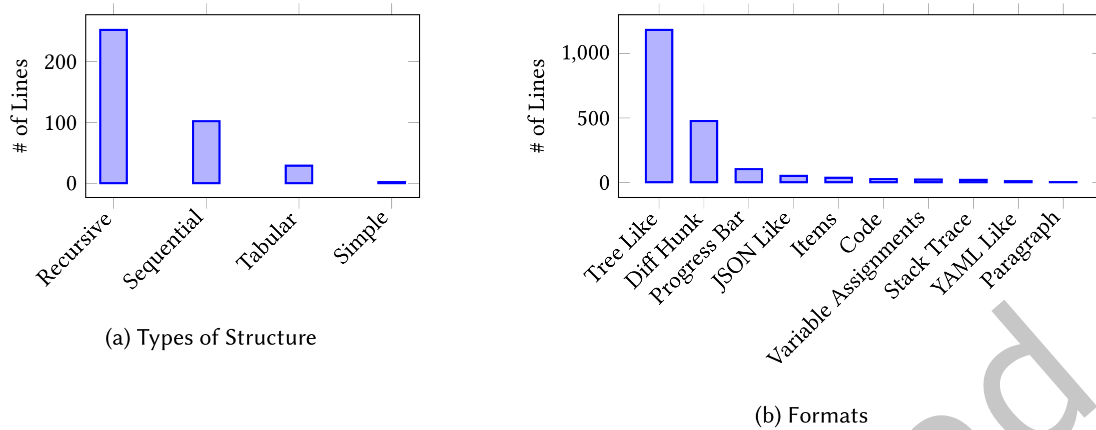
In this section, we present a deep-learning-based automated multiline log segmentation approach and the evaluation of its performance (RQ. 4).

4.1 Motivation

As shown in Section 3, multiline logs are prevalent in console logs and possibly other classes of logs. However, because existing log parsers assume that logs are single-line only, the presence of these logs will degrade the log parsing accuracy and increase the noise, making downstream log analysis results unreliable. We show the impact of multiline logs on log parsers in Section 5. Thus, a dedicated step is needed to identify the boundaries of multiline segments and classify their format, enabling specialized parsing of complex formats.

To generalize log parsing to more classes of log data, we present a preprocessing approach for log files prior to parsing. The first challenge in using existing automatic log parsers for multiline console logs is identifying the boundaries of each log segment. Most of these parsers currently assume that these boundaries are the start and end of each line. However, as we have shown in Section 3, this is not the case due to the prevalence of multiline logs in console logs. Without this assumption, these boundaries must be determined in advance. We propose preprocessing console log files by splitting them into segments, each containing exactly one single-line or one multiline log entry. A format label can be assigned to each segment to aid downstream parsing.

Just Accepted



(a) Types of Structure

(b) Formats

Fig. 7. Median length of multiline log segments by their type of structure and by their format.

4.2 Approach

To identify multiline segments and their format, we used a semantic-aware deep-learning model. We present an overview of our approach in Fig. 3b. Among alternatives, naive methods such as heuristic-based methods and regex pattern matching cannot be relied on due to two main factors. First, as shown in Section 3, multiline logs can contain complex structures and varying formats. They would require significant manual effort to construct and maintain a robust set of heuristics or regex patterns to reliably identify the boundaries. Second, some formats could potentially require advanced parsing techniques to be identified. For example, multiline logs can contain code or formatted messages that have recursive structures, increasing the difficulty of manually constructing heuristics. We also investigated the use of LLMs to split the multiline log segments and describe their format. However, we found, with simple prompts, that the results obtained do not align with our definition of multiline logs. In addition, naively using LLM is inefficient in the context of log processing due to the large volume. Instead, we fine-tune a small language model (127M parameters), pretrained in natural language, code, and console logs.

Step 1: Labeling and Data Preprocessing. In the LogChunks dataset, there are more lines of multiline logs than lines of single-line logs (Finding 3). We also need a sufficient number of counterexamples to multiline logs, that is, sequences of single-line logs and examples of the intersection between a single-line log and a multiline log. Thus, during our labeling process, detailed in Section 3.2, when we encountered a single-line log in the sample, we also labeled the lines that follow it. To put a limit on the labeling effort, we stopped after 10 lines for each example or when we encountered a multiline log. In the latter case, we also labeled the multiline log. This has the effect that, in the training data, there would be sequences of lines from multiline logs, but also sequences of single-line logs, which would benefit the model in learning the difference.

From the manually labeled samples, we constructed sequences of lines that include the borders of multiline logs. Since we do not label the lines outside of multiline log segments, not all lines will have a label; however, from the construction of the segments, we can infer that the line preceding a segment will have the label EOL. We obtained 897 sequences of labeled lines. We randomly left out 140 (roughly 15%) sequences for evaluation. We ensured that there was at least one sample of each format in this evaluation dataset. The other 757 sequences were used for training.

Step 2: Modelling. We define the training task as a supervised multi-class classification problem. Let $S := \langle x_i \rangle_{i=0}^{n}$ be the tokenized segment of lines of logs where x_i is the ith token in the segment, and let $\mathbf{Y}$ be the set of labels.

Just Accepted

For each `\n` (end-of-line) token x_i we assign a label $\tilde{y}_i$ where

$$\tilde{y}_i = \arg\max_{y \in \mathbf{Y}} \mathbb{P}\left(Y_i = y \mid S_{i-b}^{i+a}\right) \tag{1}$$

$$= \arg\max_{y \in \mathbf{Y}} f_\theta\left(y, S_{i-b}^{i+a}\right) \tag{2}$$

and b and a are the number of lines to include as context during labeling, respectively, before and after the current line. During training, these two values are dynamic and depend on the position of new-line characters in each training sample. y is a potential label that can be assigned to the `\n` token. The predicted label is the potential label with the largest estimated probability.

To estimate these probabilities, in Eq. (2), we used a model f_θ parameterized by the trainable parameters $\theta \in \Theta$. The labels in $\mathbf{Y}$ are the format labels manually found labels described in Section 3. We also introduced the end-of-log (EOL) label. The EOL label is assigned to the lines that end the log messages, to identify the boundaries of log segments. That is, a line that is assigned the EOL label is either a single-line log or the last line of multiline logs.

Similar to previous work leveraging a pretrained model for log analysis [10, 17, 34], we used a transformer encoder model. Since console logs contain code elements and snippets, we initialized our weights from those of CodeBERT [5], a model pretrained not only on natural language but also code, reducing our pretraining effort considerably [32]. We pretrained our model on log data from the LogChunks dataset [1]. We then fine-tuned the model on the manually labeled dataset to specialize it in the task of classifying log lines.

Step 3: Training. To find θ, we used gradient descent with the AdamW [22] optimizer to train a transformer encoder model [5] to learn an efficient encoding of the structures in multiline logs and perform the classification task to assign labels. We performed a two-step training process: 1) pretrain the model on a large dataset of multiline logs, then 2) fine-tune the model to perform classification using a small manually labeled dataset.

Due to the limited availability of labeled datasets and the complexity of the structures in logs, the model needs to learn the structures separately from learning the labels. To do so, we decomposed the model of Eq. (2) as follows:

$$f_\theta(y, S) = h_\theta(y, \tilde{e}_S) \tag{3}$$

$$f'_\theta(S) = h'_\theta(\tilde{e}_S) \tag{4}$$

$$\tilde{e}_S = g_\theta(S) \tag{5}$$

$\tilde{e}_S$ is a learned encoding of the tokens of the logs, computed with the encoding model, and which we input into the classification and masked language modeling (MLM) heads. Before fine-tuning the model on the classification task, we pretrained f'_θ on a larger dataset of multiline logs. Specifically, given a sequence of tokens, we randomly mask 15% of the tokens. To increase the robustness of the pretrained model, we replaced 80% of masked tokens with random tokens. We use the default masking and replacement probabilities of the training utility.[2]

During training, given a transformed sequence, the model predicts the probability distribution of the original tokens for each token in the sequence. The model parameters are optimized by minimizing the cross-entropy between the prediction and the actual distribution using the AdamW optimizer [22]. The result is a model whose latent states can effectively encode the structures of multiline logs, such that the training of the final task would require fewer pretraining data.

The final model f_θ predicts the labels of each line while also taking into account the surrounding context (Eq. (2)). We therefore replace the last layer of the model with the classification head h_θ and fine-tune the model, using a manually labeled dataset, to output probabilities of the labels for each token. Since the task is to classify

[2]https://github.com/huggingface/transformers/blob/main/src/transformers/data/data_collator.py#L764

Just Accepted

Table 3. Summary of the heuristics used in the baseline approach.

Heuristic Type	Description	Formats
Off-the-shelf parser	A parser that we can directly apply and verify if it can parse a segment successfully.	YAML-Like, Diff-Like
RegEx	A set of RegEx found by sampling and iterative refining	Stack Trace, Items, Progress Bar, Variable Assignments, Tree Like
Custom Rules	A set of more complex rules that match our definition of the format	Code, Paragraph, JSON Like

lines, we only considered the labels assigned to the `\n` (new line) tokens and ignored all other tokens. During training, the inputs to the model are sequences of tokens surrounding the start and the end of each labeled snippet in the dataset. That is, let i be the position of the starting or ending token of a snippet, the input to the model would be the token sequence $\langle x_{i-b}, x_{i-b+1}, \dots, x_i, \dots, x_{i+a-1}, x_{i+a} \rangle$. We set $a + b + 1 = w$, where w is the maximum input length of the model, to compact the training samples. Thus, each training sample has multiple new lines where labels would be assigned. Because we included unlabeled tokens before the start and after the end of labeled snippets, those tokens are always ignored. Similarly to the pretraining step, the model is trained by minimizing the cross-entropy loss using the AdamW optimizer [22].

4.3 Baseline

While, to the best of our knowledge, there are no other tools that perform segmentation on multiline logs, it is possible to use lightweight heuristics to identify segments of specific formats. Existing approaches range from collection-time segmentation, which directly collects multiline logs as a single record, to pattern-matching-based state-machines to identify segment boundaries.

As a baseline, inspired by existing approaches and adapted to the dataset at hand, we implemented a greedy iterative sliding-window-based heuristic approach. Because some types of segments we found don't contain identifiable patterns at their boundary, a sliding window approach allows us to use quantitative heuristics that don't require matching against patterns, such as character class instance counting. Beginning with a window at its maximum size, we applied heuristic rules defined for each format in a specific order. If no rule identifies a segment, we decrease the window size by one until it reaches 1; in this case, a single-line segment is reported. We then move to the next iteration by shifting the window by the length of the last found segment. Heuristic rules are ordered from the most strict to the least strict. We summarize the different types of heuristics used in Table 3.

When available, we use off-the-shelf parsers as heuristics: we attempt to parse the candidate lines with the parser; if no error is thrown, the heuristic successfully identifies the segment. Off-the-shelf parsers are used for YAML-Like and Diff Hunk formats. We do not use a JSON parser for the JSON-Like format since, in our labeling, we considered formats using delimited brackets as JSON-Like, but they rarely are valid JSON objects. We use a paired delimiter matcher instead.

For other formats without off-the-shelf parsers, we employed RegEx patterns and rules. RegExes are constructed for each format by first sampling distinct lines of that format. By looking at each example in the sample, we look line by line and remove repeated parts and distinct natural language in each line until only common words, symbols, and digits remain. And finally, each remaining distinct line is used to build a RegEx pattern. For other formats, we ensure that the rules match our definition for the respective formats and that they are able to identify segments individually.

Just Accepted

Table 4. Performance of the baseline rule-based approach (Rules) and our deep-learning based approach (DL) on the manually labeled dataset.

Method	F1	Precision	Recall
Rules	0.73	0.64	0.85
DL	**0.88**	**0.84**	**0.93**

Table 5. Performance of the baseline rule-based approach (Rules) and our deep-learning-based approach (DL) on format classification tasks on the manually labeled dataset.

	F1		Precision		Recall	
Format	Rules	DL	Rules	DL	Rules	DL
Code	0.38	**0.56**	0.24	**0.83**	**0.85**	0.42
Items	0.2	**0.87**	**1**	0.91	0.11	**0.83**
Paragraph	0.15	**0.67**	0.13	**0.85**	0.19	**0.55**
JSON-Like	0.14	**0.86**	0.1	**1.00**	0.2	**0.75**
Progress Bar	0.91	**0.96**	**0.93**	0.92	0.9	**1.00**
Stack Trace	0.54	**0.64**	0.94	**1.00**	0.38	**0.48**
YAML-Like	0.26	**0.93**	0.54	**0.95**	0.17	**0.90**

Just Accepted

4.4 Evaluation

When training the model, we split our manually labeled dataset into a training set and an evaluation set. We train the model for 100 epochs, and we select the checkpoint that achieves the smallest cross-entropy loss on the evaluation set, which was at epoch 16. In this way, we do not select an overfit checkpoint. For the segmentation task, we measure the accuracy of the model at identifying boundaries of segments. For the classification task, we measure the binary classification accuracy. We use the one-versus-all F1-score, precision, recall, and ROC AUC (Receiver Operating Characteristic Area Under the Curve) on the evaluation set.

4.5 Results

RQ. 4: How accurately can we automatically identify multiline logs and their structures? Table 4 shows the performance of our baseline and deep-learning approach at the segmentation task. **The model achieves an F1-score of 0.88, outperforming the baseline rule-based approach.** It scored a higher recall (0.93) than precision (0.84), indicating that the model performs well in identifying the boundaries of most logarithmic segments, but its predictions can have false positives.

For log format classification, the model achieves a weighted average of F1-scores of 0.84 across all formats. Table 5 shows the format classification performance. However, the evaluation results for the Diff Hunk, Tree-like, and Variable Assignment labels were not stable due to their total training samples not exceeding five samples, and were therefore omitted in Table 5. The model performed the worst at predicting the Code label, with an F1-score of only 0.56, as this format contained the most complex structures, semantics, and noise. Similarly, the model does not perform well on the Paragraph label with a low recall due to the similarity with single-line logs. The Stacktrace format has high variability since different programs output different-looking stacktraces, resulting in a low recall.

The deep-learning approach outperforms the baseline approach on all formats (F1). One cause for the bad performance of the baseline approach is that it is constrained by the sliding window size: multiline structures can be broken. The deep-learning approach solves this by learning finer-grained structures and semantics that are

Listing 3 Example of a segment where heuristics are hard to define due to natural language and distinct patterns.

```
# a sequence of versions; may contain sematic versioning numbers, names and comments
    system
    5.6
    5.6.40
    7.1
    7.1.27
    7.2
* 7.2.15 (set by /home/travis/.phpenv/version)
    hhvm
    hhvm-stable
```

Listing 4 Example of a segment where it is hard for a heuristic method to determine whether this should be classified as a Code format or Paragraph format.

```
Overfull \hbox (3.10783pt too wide) in paragraph at lines 12326--1232
[]\T1/ptm/m/n/10 A data frame. Used to re-trieve the col-umn names as choices f
or a [][]\T1/zi4/m/n/10 selectInput()[][][]
```

hard to capture with heuristic rules. For example, on the Items format, the DL method achieves a F1-score of 0.87, while the heuristic method only achieves 0.2; this is explained by the low recall (0.11) caused by the difficulty in defining heuristics that can capture sequences of items containing natural language or distinct structure. Listing 3 illustrate this problem. Another cause is that heuristic rules can overlap, as can be seen by the larger differences between precision and recall. For example, code contain a wide range of patterns, e.g. a python dictionary literal is really similar to JSON-like formats. Consider also the segment in Listing 4, where LaTeXcommands are mixed in with natural language and code-like paired delimiters: heuristics relying on patterns found in code would fail in such cases. For heuristic rules, it is difficult to determine if such a segment is part of a larger Code segment, or a JSON-like formatted segment, or even some Paragraph containing code elements, explaining the low precision (0.24). The DL method, instead of relying on heuristic patterns, can take advantage of the surrounding semantic context and learned structures to determine the boundaries, making it more generalizable and robust compared to the heuristic method. This is reflected by the lower gap between precision and recall of the DL method. For example, in Listing 4, it learns the overall structure of the "hbox overfull" event, and in Listing 3, it learns the semantic meaning of a version specifier.

It took 5.3 seconds per file on average to segment every file in the LogChunks dataset, or 1.4 milliseconds per line of log, using the inference algorithm, as will be detailed in Section 5, on a single workstation with an RTX 5090 GPU and 32GB of RAM. For each of the 797 log files, an individual process is spawned, and thus the overhead associated with the loading of weights is also included.

> **Finding 4.** Our deep-learning-based approach can identify the boundaries of log segments with an F1-score of 0.88. It can predict the format of multiline log segments with a weighted average of F1-score of 0.84 across all formats. It outperforms the baseline heuristic rule-based approach on both tasks.

5 Impact of Multiline Log Segments on Log Parsing

In this section, we explore the impact of multiline segments in console logs on log parsing (RQ. 5) by segmenting logs using the method presented in Section 4.

Just Accepted

Algorithm 1 The algorithm to segment lines of log messages into segments representing discrete events.

Require: classifier M **and** sequences of logs $s_{1..n}$ **and** window size w **and** lookahead size a
Ensure: E contains all log segments

```
1: T ← stream of tokens of s_{1..n}
2: B ← empty sliding window of size w
3: S ← empty segment
4: E ← empty list of segments
5: shift a tokens from T into B
6: for i ← 1..n do
7:     shift l tokens from T into B
8:     y ← M(B)
9:     append (s_i, y_{w−a}) to S
10:    if y_{w−a} = EOL then
11:        append S to E
12:        S ← new empty list
13:    end if
14: end for
```

5.1 Motivation and Approach

As shown in Section 3, multiline log segments contain sequential and recursive structures that span multiple lines, and they are relatively noisy in the context of log parsing compared to system event logs. It remains unknown whether the presence of these multiline segments would impact log parsing and, in turn, other downstream tasks that rely on log parsing.

To quantify the impact of multiline logs, we evaluated the performance of existing log parsers on multiline logs using both our manually labeled dataset and all logs from LogChunks before and after segmentation. We started by segmenting the logs in the dataset. Then, since the log parsers do not support multiline logs out of the box, we applied three different adaptation schemes: 'singleline', 'concatenate', and 'firstline'. We also constructed a baseline dataset from the single-line-only logs. In our experiment, we used Drain [8], a commonly used open-source log parser, and LogBatcher [26], a performant LLM-based log parser. Fig. 3c summarizes our approach.

Step 1: Segmenting the Logs. To segment the multiline logs using the model presented in Section 4, we inferred each line's format label using an efficient windowing algorithm. It iterates over the log files line by line, performs classification on each line, and accumulates them into lists that represent individual segments. Because the lengths of the lines are not known ahead of time and may exceed the maximum context length for which the model is trained, a sliding window centered on the new line token is used. This window is defined by a window size w and a look-ahead size a. To position this sliding window at each iteration, tokens from the stream are shifted into a buffer list with a maximum size w; as tokens are added to the end of the list, tokens are removed from the start in a first-in-first-out manner to ensure that its maximum size is not exceeded.

The pseudo-code of the inference algorithm is shown in Algorithm 1. The algorithm starts by shifting a tokens (line 5) into the buffer B that acts as the sliding window. Then, for each line, it shifts l tokens where l is the number of tokens in the line (line 7). At each iteration, the new line character of the line is guaranteed to be at the position $w - a$ in the buffer. Next, B is used as input to the segmentation model (line 8). The labeled line is then added to the list of lines S (line 9) representing a partially constructed segment. If the model assigns an EOL

Just Accepted

Listing 5 Examples of Overgeneric and Overspecific templates.

```
// overgeneric, since it can match different events that should not have the same template
<*> <*> && \

// overspecific, since the variable will always be 100 (progress bar is full)
######################################################################## <*>%
```

label, the segment is completed, and S is appended to the output list E. Then, S is reinitialized to a new list (lines 10 to 12). We continue the iterations and repeat the accumulation step for each line of the log file.

Since each token is evaluated up to a constant number of times, depending on the window size, and since the window size is constant, Algorithm 1 runs in linear time with respect to the number of tokens.

We applied this method to segment logs from the manually labeled dataset and from the entire LogChunks dataset.

Step 2: Adapt multiline segments to log parsers. To work around the limitation of current log parsers that only take single-line logs as input, we need to adapt multiline segments into single lines. We used three different adaptation schemes:

(1) In the **Singleline** scheme, multiline segments are split into individual single-line logs and given to log parsers as-is. This corresponds to not applying any aggregation to the multiline segments.
(2) In the **Concatenate** scheme, lines of each multiline segment are joined into a single line using a whitespace character. This scheme preserves all information, including noisy structures. Due to the length of the multiline segments, the resulting line length may increase the computational cost of log parsing.
(3) In the **Firstline** scheme, only the first line in each segment is used and given to log parsers. Compared to other schemes, this incurs the lowest computational cost, at the expense of preserving information. Nonetheless, as we've found in Section 3, multiline segments may contain repetitive structures where the first line contains enough of the structure for log parsers to find.

Step 3: Finding the Log Templates. Using each of the log parsers and each of the segment adaptation schemes, we computed a set of templates of the logs in our manually labeled datasets and of the logs from each of the 80 projects. We also computed sets of templates, used as a comparison baseline, from the same logs, but with multiline segments removed, such that only single-line logs are used to generate the templates. However, since our segmentation method will inevitably yield false negatives, we filter out templates that potentially correspond to misclassified multiline logs.

When using the Singleline scheme, some templates will be obtained from individual logs that are in multiline segments, which we refer to as multiline-only templates. These templates may not be structurally or semantically complete. To find these templates, we take the difference between the set of templates obtained with the Singleline scheme and the baseline set of templates.

Step 4: Measuring the Impact. Because we do not have manually verified labels of the constructed log templates, we can not measure the performance of log parsing using traditional metrics such as parsing accuracy, grouping accuracy [13], or metrics that require ground truth. Instead, we use the count of templates that are overspecific and overgeneralized to quantify the negative impact on log parsing.

We define **overspecific templates** as **templates that can only match logs that share the exact same message**. Similarly, we also define **overgeneralized templates** as **templates that match a superset of logs that another template that is not overspecific**. Listing 5 shows an example of each. We find overgeneralized templates by directly matching templates against other templates as if they were logs. The presence of overspecific and overgeneralized templates is a result of the negative impact. We interpret the higher count of these templates as a worse log parsing performance.

Just Accepted

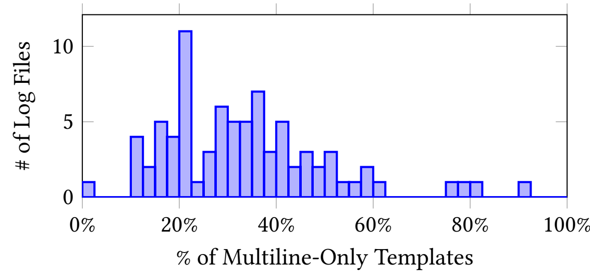


Fig. 8. Distribution of proportion of multiline-only templates of log files.

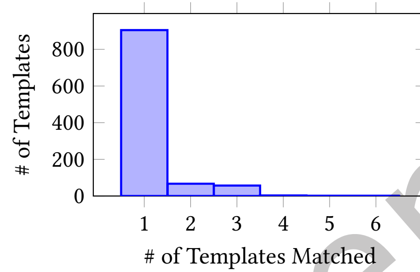


Fig. 9. Distribution of the number of templates matched by overgeneralized multiline-only templates.

Just Accepted

We counted the number of templates, overgeneric templates, and overspecific templates for each scheme and the baseline. We present our findings as relative changes of the counts against the baseline using Eq. (6).

$$\%\Delta x = \frac{x - x_{baseline}}{x_{\text{baseline}}} \tag{6}$$

5.2 Results

RQ. 5: What is the impact of multiline log segments on log parsing? **A significant proportion of templates found were multiline-only templates.** We found a total of 72,464 templates, 54,903 single-line-only templates, and 19,265 multiline-only templates. Additionally, when parsing single-line-only logs, we found 1,704 templates that are not in $\mathcal{T}$. That is, 24% of all templates are multiline-only. Fig. 8 shows the distribution of this proportion for individual log files in the dataset. The average proportion of multiline-only templates per log file is 34%. For some files, this proportion is much greater, up to 91%. We note that these templates are likely inaccurate because they span multiple lines, whereas the log parsers were designed only for single-line logs.

A significant proportion of multiline-only templates are overspecific or overgeneralized. Out of the multiline-only templates, 5,750 (30%) are overspecific. That is, they could only parse logs containing the same message. We found 1,034 multiline-only overgeneralized templates, that is, 5% of all multiline-only templates. Fig. 9 shows the distribution of the number of templates matched by the overgeneralized templates. Nine hundred five overgeneralized templates matched one template, while only 129 matched two or more. In total, 6,784 multiline-only templates were either overspecific or overgeneralized.

On our manually labeled dataset, **using the Singleline scheme, template quality is reduced while applying the Concatenate and Firstline increases the quality of the templates produced by both Drain**

Table 6. Relative increase in average count of overgeneric and overspecific templates produced by Drain and LogBatcher on log segments from our manually labeled dataset. %Δ is the relative change compared to the baseline. A larger %Δ indicates worse template quality.

Log Parser	Parsing Mode	%Δ Templates	%Δ Overgeneric	%Δ Overspecific
drain3	singleline	0.535	0.619	0.562
drain3	concatenate	0.144	**0**	0.168
drain3	firstline	0.066	**0**	**0.073**
logbatcher	singleline	0.517	0.3	0.535
logbatcher	concatenate	0.161	0.067	0.187
logbatcher	firstline	0.115	**−0.067**	**0.135**

Table 7. Relative increase in number of overgeneric and overspecific templates produced by Drain and LogBatcher on all logs from LogChunk using segments found with our automatic approach. %Δ is the relative change and the p-value is from Welch's t-test. A larger %Δ indicates worse template quality.

Log Parser	Scheme	#Templates		#Overgeneric		#Overspecific	
		%Δ	p-value	%Δ	p-value	%Δ	p-value
drain3	singleline	**0.508**	**0.001**	0.421	0.168	**0.425**	**0.004**
drain3	concatenate	0.148	0.254	0.093	0.757	0.141	0.297
drain3	firstline	0.058	0.646	0.149	0.612	0.052	0.693
logbatcher	singleline	**0.604**	**0.002**	0.43	0.074	**0.623**	**0.001**
logbatcher	concatenate	0.235	0.213	0.139	0.551	0.216	0.222
logbatcher	firstline	0.191	0.306	0.029	0.898	0.179	0.307

and LogBatcher. Table 6 shows the relative change of the count of overgeneric and overspecific templates compared to the baseline (logs with multiline segments removed). When applying the Singleline scheme (no aggregation), the number of overgeneric and overspecific templates produced by both Drain and LogBatcher increases, implying a negative impact of multiline segments on log parsing if no aggregation is performed. When the Concatenate and Firstline scheme is used, Drain did not produce any additional overgeneric templates, and LogBatcher produced some additional overgeneric templates. Interestingly, LogBatcher produced fewer additional overgeneric templates than Drain when using the Singleline scheme and fewer overgeneric templates than the baseline. We observe that multiline segments cause a greater increase in overspecific templates than templates in general.

Table 7 shows the result of the same experiment but with the whole LogChunks dataset segmented using our method described in Section 4. We observe that Drain produces 51% more templates and 43% more overspecific templates, and LogBatcher produces 60% more templates and 63% more overspecific templates. This difference is statistically significant, with a p-value from Welch's t-test, in both cases, of less than 0.01. In all other combinations of parser, scheme, and metric, while we do observe an increase in the number of templates, the differences are, however, not statistically significant. While this could indicate that aggregating multiline segments can help with improving templates produced by existing parsers, we do note that it could also indicate instability in the performance of our segmentation approach. Furthermore, the increase in overspecific templates is less than the

Just Accepted

increase in templates in general, suggesting that the impact of the multiline nature of console logs is less than that of the complex structures it has. While we show in Section 4.5 that our method can accurately identify segments, errors quickly negate the usefulness of aggregating multiline segments for log parsing.

> **Finding 5.** We find that the presence of multiline logs can negatively impact log parsing performance by causing log parsers to produce more overgeneric and overspecific templates when using the singleline scheme. When using a scheme that aggregates multiline segments, the impact is reduced. Using an accurate segmentation method combined with aggregating multiline segments is critical in mitigating this impact.

Just Accepted

6 Discussion and Implications

In this section, we discuss our findings from Sections 3 to 5 and present their implications.

The problem with log templates. In our qualitative analysis of console logs in Section 3, we find complex structures (Finding 1) and varying formats in which the information is presented (Finding 2). Most often, these structures cannot be recognized by the limited grammar of log templates. Take sequential structures as an example. Many existing log parsers produce log templates [8, 12, 27, 30], which cannot effectively capture sequences of dynamic values with varying lengths. In log templates, each wildcard can match only a single dynamic value. Different sequence lengths would require different log templates to accommodate each possible number of dynamic values, resulting in an exploding log template count. Listing 1 is an example of such logs. This problem extends to tabular and recursive logs.

Prevalence of multiline logs. In our analysis, logs that span multiple lines represent 66% of all lines, and each segment tends to be long (Finding 3). In fact, we find that the median length of multiline segments is 57 lines; some of the logs have lengths as high as 31,288 lines. If console logs were to be analyzed on a line-by-line basis or without parsing them into discrete events, the lengthiness of multiline segments could introduce imbalance and bias towards the information contained in multiline segments. Properly identifying lines that are part of the same segment can thus allow downstream tasks to filter out or weigh down these segments and reduce the bias towards these lengthy segments.

Impact of multiline log segments. We find that, without a specific parsing technique tailored to multiline logs and by using an existing log parser that assumes that logs are single line, the quality of the templates found is degraded (Finding 5). In fact, on average, 34% of the templates found in log files, using Drain, correspond to lines that are part of multiline log segments. In these logs, a significant proportion are overspecific and overgeneralized (30% and 5%, respectively). On the one hand, overspecific templates increase the noise of the parsed log sequences. The same or similar log event can have different event IDs for having different dynamic values. On the other hand, overgeneralized templates hide information by grouping different log events under a single event ID. Overall, their presence indicates degraded log parsing performance. Preprocessing these logs by simply removing multiline log segments before log parsing can have a significant positive impact on downstream tasks by taking out overspecific and overgeneralized templates. Future work can investigate specialized log parsers for specific formats, such as the one produced by our segmentation approach (Section 4).

7 Threats to Validity

Internal Validity. In our qualitative analysis of the console logs in Section 3, we label a sample of logs with a manual coding approach. The process of assigning labels and identifying the boundaries of log segments is inherently subjective to each annotator, depending on both experience and preferences. To mitigate, we used statistical tools such as Cohen's and Fleiss's Kappa score to quantify the agreement between the annotators.

The training data set used for fine-tuning contains varying numbers of samples for different formats of multiline logs. This data imbalance may limit the ability of the deep-learning model to learn representative features for each format. To address this issue, we introduce a pretraining step to allow the model to learn not only labelled data, but also the general structures of console logs through masked language modeling.

In the impact analysis of multiline logs on log parsing in Section 5, errors from the automatic segmentation step are propagated to log parsing. To isolate our results from this error, we filter out templates that potentially correspond to misclassified multiline logs and we repeat our experiment using manually labelled segments from Section 3.

External Validity. Our study revolves around a single dataset, LogChunks, which consists of logs from the Travis CI automated build system. These logs may not represent the diversity of console logs encountered in other contexts, such as industrial production environments, which may depend on console applications. As a result, the structures and formats we identify in Section 3 and the evaluation of our approach in Section 4 may not generalize to other console logs. However, the logs in LogChunks is taken from a wide variety of open-source projects using different build steps and tools. Furthermore, our automatic console splitting approach does not make assumptions on the format of the log files, and thus it can easily be adapted to new datasets by continuing the pretraining and fine-tuning.

Construct Validity. In our qualitative study, we define multiline log segments as a sequence of lines with consistent formatting and semantics pertaining to a single event in the system. However, the concept of an "event" in log analysis can be interpreted in various ways depending on the context and may not be reflected by well-defined syntactic or structural features. Furthermore, some of the format labels that we use overlap in some cases, as we find in Section 4.5, which introduces ambiguity in both manual annotation and model prediction. To mitigate these issues, through discussion sessions, we formulate a single definition of what a multiline log segment is.

8 Related Work

To the best of our knowledge, our study is the first work that proposes a preprocessing step to split multiline logs into distinct segments before log parsing. However, there are related studies in text segmentation and multiline log parsing. We briefly present them in this section.

Text Segmentation. Splitting text into segments is not a unique problem in console log parsing. In natural language processing (NLP), splitting text into sentences, words, and tokens is a common data preprocessing step. WtP [24] is a self-supervised approach to identifying sentences in multiple languages, including those where punctuation is not always used to delimit sentences. They fine-tune a pretrained language model to predict the probability of observing a new line `\n` after each character. Using a threshold, they mark the end sentences on characters with a high probability of preceding a new line. SaT [7] improves the efficiency and robustness of the previous approach by using subword tokenization and augmenting training data to include corruption comparable to those seen in real-world text from various sources. While text segmentation is closely related to multiline log segmentation, there is a crucial difference: new lines can appear in the middle of a multiline log and do not indicate the end of a segment.

Multiline Logs. Most log parsing approaches are based on the assumption that logs are delimited. However, as discussed in Section 2, this is not the case for all logs. Few approaches consider multiline logs. Yu et al. [29] propose LPME, a log parsing framework that produces events that span multiple lines. Although this approach can parse multiline logs, it assumes that each line can be parsed using an existing log parser. However, this assumption does not hold, as we show in Section 2. Xu et al. [27] introduce the idea of **hybrid logs**, that is, logs where single line event logs are mixed with other types of multiline logs. In their study, they consider three types: event logs, table logs, and text logs. They propose Hue, a log parser capable of parsing these types of log. Unlike

Just Accepted

LPME, Hue does not rely on parsing individual lines before extracting log templates. However, it assumes that the lines in the same log are similar or have a consistent indentation. In Section 3, we show that logs can contain more types of structure in even more inconsistent formats.

Log Parsing. Many other automated approaches [13, 18] have been proposed to abstract logs into digestible representations. However, most make restrictive assumptions that are not compatible with multiline console logs. For example, fixed-depth prefix tree-based approaches [8, 30], have seen wide usage. Other approaches leverage semantic information [11, 19, 21] to identify dynamic variables while providing additional semantic information to downstream tasks. Recently, many LLM-based approaches have also been proposed [12, 16, 23, 31] leveraging various few-shot prompting techniques to generate templates on-the-fly.

9 Conclusion

In this paper, we present a qualitative analysis of multiline segments in console logs. Our study reveals the structural and formatting diversity of these segments and their disproportionate presence in console logs. Based on our findings, we propose an automated log splitting approach using a deep learning model pretrained on console log data and fine-tuned on manually labelled examples. Our model effectively identifies segment boundaries and predicts formats, achieving relatively good performance across most formats. Using our approach, we find that multiline logs had tangible impact on log parsing. Our results also demonstrate the importance, feasibility, and value of automated segmentation of multiline logs as a preprocessing step for log analysis. Future work can explore broader log sources, refine structure type and format taxonomies, and integrate format-aware parsing for downstream tasks.

Acknowledgement

We gratefully acknowledge the financial support of: (1) JSPS for the KAKENHI grants (JP24K02921, JP25K03100, JP25K22845); (2) Japan Science and Technology Agency (JST) as part of Adopting Sustainable Partnerships for Innovative Research Ecosystem (ASPIRE), Grant Number JPMJAP2415, and (3) the Inamori Research Institute for Science for supporting Yasutaka Kamei via the InaRIS Fellowship.

Just Accepted

References

[1] Carolin Brandt, Annibale Panichella, and Moritz Beller. 2020. LogChunks: A data set for build log analysis. doi:10.5281/ZENODO.3632351

[2] Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R Lyu. 2021. Experience report: Deep learning-based system log analysis for anomaly detection. **arXiv [cs.SE]** (July 2021). arXiv:2107.05908 [cs.SE] http://arxiv.org/abs/2107.05908

[3] Moataz Chouchen, Ali Ouni, Raula Gaikovina Kula, Dong Wang, Patanamon Thongtanunam, Mohamed Wiem Mkaouer, and Kenichi Matsumoto. 2021. Anti-patterns in modern code review: Symptoms and prevalence. In **2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)**. IEEE, 531–535. doi:10.1109/saner50967.2021.00060

[4] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. **Educational and psychological measurement** 20, 1 (April 1960), 37–46. doi:10.1177/001316446002000104

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. **arXiv [cs.CL]** (Feb. 2020). arXiv:2002.08155 [cs.CL] http://arxiv.org/abs/2002.08155

[6] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. **Psychological Bulletin** 76, 5 (Nov. 1971), 378–382. doi:10.1037/h0031619

[7] Markus Frohmann, Igor Sterner, Ivan Vulić, Benjamin Minixhofer, and Markus Schedl. 2024. Segment any text: A universal approach for robust, efficient and adaptable sentence segmentation. **arXiv [cs.CL]** (June 2024). arXiv:2406.16678 [cs.CL] http://arxiv.org/abs/2406.16678

[8] 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. doi:10.1109/icws.2017.13

[9] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2022. A survey on automated log analysis for reliability engineering. **ACM computing surveys** 54, 6 (July 2022), 1–37. doi:10.1145/3460345

[10] Shaohan Huang, Yi Liu, Carol Fung, He Wang, Hailong Yang, and Zhongzhi Luan. 2023. Improving log-based anomaly detection by pre-training hierarchical transformers. **IEEE Transactions on Computers. Institute of Electrical and Electronics Engineers** 72, 9 (Sept. 2023), 2656–2667. doi:10.1109/tc.2023.3257518

[11] Yintong Huo, Yuxin Su, Cheryl Lee, and Michael R Lyu. 2021. SemParser: A Semantic Parser for Log Analysis. **arXiv [cs.SE]** (Dec. 2021). arXiv:2112.12636 [cs.SE] http://arxiv.org/abs/2112.12636

[12] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2024. LILAC: Log parsIng using LLMs with adaptive parsing cache. **Proceedings of the ACM on Software Engineering** 1, FSE (July 2024), 137–160. doi:10.1145/3643733

[13] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Jieming Zhu, and Michael R Lyu. 2024. A large-scale evaluation for log parsing techniques: How far are we?. In **Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis**, Vol. 32. ACM, New York, NY, USA, 223–234. doi:10.1145/3650212.3652123

[14] Lukasz Korzeniowski and Krzysztof Goczyla. 2022. Landscape of automated log analysis: A systematic literature review and mapping study. **IEEE access: practical innovations, open solutions** 10 (2022), 21892–21913. doi:10.1109/access.2022.3152549

[15] J R Landis and G G Koch. 1977. The measurement of observer agreement for categorical data. **Biometrics** 33, 1 (March 1977), 159–174. doi:10.2307/2529310

[16] Van-Hoang Le and Hongyu Zhang. 2023. Log parsing with prompt-based few-shot learning. **arXiv [cs.SE]** (Feb. 2023). arXiv:2302.07435 [cs.SE] http://arxiv.org/abs/2302.07435

[17] Van-Hoang Le and Hongyu Zhang. 2024. PreLog: A pre-trained model for log analytics. **Proceedings of the ACM on Management of Data** 2, 3 (May 2024), 1–28. doi:10.1145/3654966

[18] Zhijing Li, Qiuai Fu, Zhijun Huang, Jianbo Yu, Yiqian Li, Yuanhao Lai, and Yuchi Ma. 2024. Revisiting log parsing: The present, the future, and the uncertainties. **IEEE transactions on reliability** 73, 3 (Sept. 2024), 1459–1472. doi:10.1109/tr.2023.3340020

[19] Zhenhao Li, Chuan Luo, Tse-Hsun Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. 2023. Did we miss something important? Studying and exploring variable-aware log abstraction. **arXiv [cs.SE]** (April 2023). arXiv:2304.11391 [cs.SE] http://arxiv.org/abs/2304.11391

[20] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in IBM BlueGene/L event logs. In **Seventh IEEE International Conference on Data Mining (ICDM 2007)**. IEEE, 583–588. doi:10.1109/icdm.2007.46

[21] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2022. UniParser: A unified log parser for heterogeneous log data. **arXiv [cs.SE]** (Feb. 2022). arXiv:2202.06569 [cs.SE] doi:10.1145/3485447.3511993

[22] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. **arXiv [cs.LG]** (Nov. 2017). arXiv:1711.05101 [cs.LG] http://arxiv.org/abs/1711.05101

[23] 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. **arXiv [cs.SE]** (April 2024). arXiv:2404.18001 [cs.SE] doi:10.1145/3597503.3639150

[24] Benjamin Minixhofer, Jonas Pfeiffer, and Ivan Vulić. 2023. Where's the Point? Self-supervised multilingual punctuation-agnostic sentence segmentation. In **Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)**. Association for Computational Linguistics, Stroudsburg, PA, USA, 7215–7235. doi:10.18653/v1/2023.acl-long.398

[25] Feifan Wu, Zhengxiong Luo, Yanyang Zhao, Qingpeng Du, Junze Yu, Ruikang Peng, Heyuan Shi, and Yu Jiang. 2024. Logos: Log guided fuzzing for protocol implementations. In **Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis**, Vol. 1. ACM, New York, NY, USA, 1720–1732. doi:10.1145/3650212.3680394

[26] Yi Xiao, Van-Hoang Le, and Hongyu Zhang. 2024. Stronger, cheaper and demonstration-free log parsing with LLMs. **arXiv [cs.SE]** (June 2024). arXiv:2406.06156 [cs.SE] http://arxiv.org/abs/2406.06156

[27] 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**. ACM, New York, NY, USA. doi:10.1145/3611643.3616260

[28] Kundi Yao, Guilherme B de Pádua, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2020. Log4Perf: suggesting and updating logging locations for web-based systems' performance monitoring. **Empirical Software Engineer** 25, 1 (Jan. 2020), 488–531. doi:10.1007/s10664-019-09748-z

[29] Mingguang Yu and Xia Zhang. 2023. An efficient way to parse logs automatically for multiline events. **Computer systems science and engineering** 46, 3 (2023), 2975–2994. doi:10.32604/csse.2023.037505

[30] Siyu Yu, Pinjia He, Ningjiang Chen, and Yifan Wu. 2023. Brain: Log parsing with bidirectional parallel tree. **IEEE transactions on services computing** 16, 5 (Sept. 2023), 3224–3237. doi:10.1109/tsc.2023.3270566

[31] Aoxiao Zhong, Dengyao Mo, Guiyang Liu, Jinbu Liu, Qingda Lu, Qi Zhou, Jiesheng Wu, Quanzheng Li, and Qingsong Wen. 2024. LogParser-LLM: Advancing efficient log parsing with Large Language Models. **arXiv [cs.SE]** (Aug. 2024). arXiv:2408.13727 [cs.SE] doi:10.1145/3637528.3671810

Just Accepted

[32] Xin Zhou, Donggyun Han, and David Lo. 2021. Assessing Generalizability of CodeBERT. In **2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. IEEE, 425–436. doi:10.1109/icsme52107.2021.00044

[33] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R Lyu. 2020. Loghub: A large collection of system log datasets for AI-driven log analytics. **arXiv [cs.SE]** (Aug. 2020). arXiv:2008.06448 [cs.SE] http://arxiv.org/abs/2008.06448

[34] Yichen Zhu, Weibin Meng, Ying Liu, Shenglin Zhang, Tao Han, Shimin Tao, and Dan Pei. 2021. UniLog: Deploy One Model and Specialize it for all log analysis tasks. **arXiv [cs.NI]** (Dec. 2021). arXiv:2112.03159 [cs.NI] doi:10.48550/arXiv.2112.03159

Just Accepted