



Latest updates: <https://dl.acm.org/doi/10.1145/3674805.3686684>

RESEARCH-ARTICLE

## A Comparative Study on Large Language Models for Log Parsing

**MERVE ASTEKIN**, Simula Research Laboratory, Barum, Akershus, Norway

**MAX HORT**, Simula Research Laboratory, Barum, Akershus, Norway

**LEON MOONEN**, Simula Research Laboratory, Barum, Akershus, Norway

**Open Access Support** provided by:

**Simula Research Laboratory**



PDF Download  
3674805.3686684.pdf  
20 December 2025  
Total Citations: 3  
Total Downloads: 1134

Published: 24 October 2024

Citation in BibTeX format

ESEM '24: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement  
October 24 - 25, 2024  
Barcelona, Spain

Conference Sponsors:  
SIGSOFT

# A Comparative Study on Large Language Models for Log Parsing

Merve Astekin  
Simula Research Laboratory  
Norway  
merve@simula.no

Max Hort  
Simula Research Laboratory  
Norway  
maxh@simula.no

Leon Moonen  
Simula Research Laboratory  
Norway  
BI Norwegian Business School  
Norway  
Leon.Moonen@computer.org

## Abstract

**Background:** Log messages provide valuable information about the status of software systems. This information is provided in an unstructured fashion and automated approaches are applied to extract relevant parameters. To ease this process, log parsing can be applied, which transforms log messages into structured log templates. Recent advances in language models have led to several studies that apply ChatGPT to the task of log parsing with promising results. However, the performance of other state-of-the-art large language models (LLMs) on the log parsing task remains unclear.

**Aims:** In this study, we investigate the current capability of state-of-the-art LLMs to perform log parsing.

**Method:** We select six recent LLMs, including both paid proprietary (GPT-3.5, Claude 2.1) and four free-to-use open models, and compare their performance on system logs obtained from a selection of mature open-source projects. We design two different prompting approaches and apply the LLMs on 1,354 log templates across 16 different projects. We evaluate their effectiveness, in the number of correctly identified templates, and the syntactic similarity between the generated templates and the ground truth.

**Results:** We found that free-to-use models are able to compete with paid models, with CodeLlama extracting 10% more log templates correctly than GPT-3.5. Moreover, we provide qualitative insights into how usable these six models are for log parsing.

**Conclusions:** Our results reveal that some of the smaller, free-to-use LLMs can considerably assist log parsing compared to their paid proprietary competitors, especially code-specialized models.

## CCS Concepts

- Software and its engineering → Software reliability.

## Keywords

log parsing, large language models, log analysis

## ACM Reference Format:

Merve Astekin, Max Hort, and Leon Moonen. 2024. A Comparative Study on Large Language Models for Log Parsing. In *Proceedings of the 18th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24), October 24–25, 2024, Barcelona, Spain*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3674805.3686684>



This work is licensed under a Creative Commons Attribution International 4.0 License.

ESEM '24, October 24–25, 2024, Barcelona, Spain  
© 2024 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-1047-6/24/10  
<https://doi.org/10.1145/3674805.3686684>

## 1 Introduction

Software systems produce system logs that offer valuable information about the system's status and execution. Developers and operators commonly leverage these logs to ensure the reliability of software systems, as they represent the only source of data that records software runtime information. This information serves as a foundation for fault detection, prediction, diagnosis, and recovery activities.

Log messages consist of natural text as well as dynamic variables that are determined at the time of writing the message. An example log message is shown in Figure 1. To make use of such a message, one first needs to transform the data into a structured format via the application of log parsers.

Two main approaches have been followed for parsing log data [11]: (*i*) *Syntax-based approaches* that utilize pre-designed features and rules requiring domain knowledge; (*ii*) *Semantic-based approaches* that usually employ deep neural networks to learn semantics within log messages. Most of the existing research is directed towards syntax-based log parsing [2, 4, 6, 8]. However, it is known that syntax-based log parsers struggle when faced with log data in different formats that do not comply with predefined rules [32]. As an alternative to syntax-based log parsers, semantic-based log parsing techniques have been proposed by several studies [9, 14, 18, 21]. By extracting semantic information from log messages, semantic-based log parsers can increase parsing accuracy. However, recent benchmark studies [12, 13] revealed insufficient performance on larger and more diverse log datasets.

The similarities between software system logs and natural language have directed researchers to explore language models for log analysis. As with other software engineering tasks [23], Large Language Models (LLMs) gained popularity for performing log-oriented tasks, such as log parsing [11, 15, 30], and log anomaly detection [20, 26]. Here, the majority of studies based on LLMs employ ChatGPT [11, 16, 23]. Although ChatGPT achieves promising results for log parsing, there is a lack of benchmarking studies that compare different LLMs to determine how and where they differ and find which ones are best suited for log parsing. Moreover, from a privacy and cost perspective, it is unclear whether there exist free-to-use models that perform competitively with ChatGPT for the log parsing task.

To address this gap, we analyze the performance of six recent LLMs—including both proprietary and free-to-use models—on system logs collected from a range of widely used open-source projects. As part of our analysis, we conclude several observations that help interpret the capability of cutting-edge LLMs to perform log parsing across a wide range of system logs. Moreover, these observations reveal that some of the smaller, free-to-use LLMs can considerably

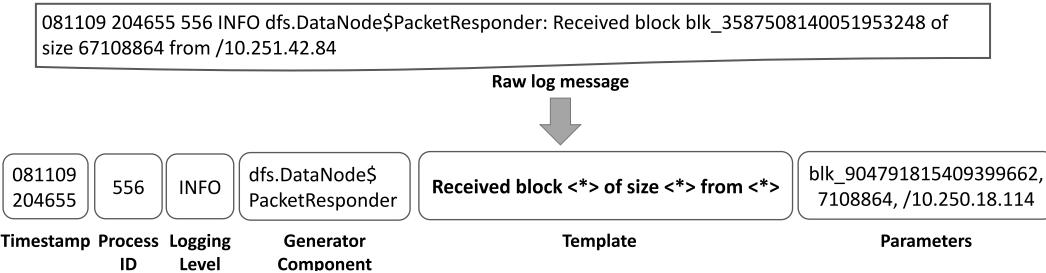


Figure 1: An example workflow of log parsing.

assist log parsing compared to their paid proprietary competitors. This is especially true for code-specialized models, such as CodeLlama [27], which is the best performing LLM on two out of three investigated metrics.

## 2 Related Work

Automated log analysis techniques mostly require structured data. Log parsing, which aims to convert unstructured log messages into structured log templates and extract the parameters, is typically the initial step toward automated log analysis. Although there have been several studies that aim at parsing system logs automatically, we focus on LLM-based approaches and evaluation metrics in this section.

### 2.1 Log Parsing with Large Language Models

Pre-trained language models, or PTMs, have been applied in several studies for different phases of log analysis, including anomaly detection [3, 7, 17] and log parsing [16, 24]. Le and Zhang [16] proposed LogPPT, for which they trained a RoBERTa model with few-shot learning. The model receives a log message as input and for each token is asked to predict whether it is a dynamic parameter or not.

Several recent and concurrent studies adopt LLMs, such as ChatGPT, for various software engineering tasks [23], including log parsing [11, 15, 30], and log anomaly detection [20, 26]. Jiang et al. [11] presented the LILAC framework, which queries ChatGPT (gpt-3.5-turbo-0613), Davinci, and Curie for log parsing. LILAC applies a selection procedure to find relevant template extraction examples to add to the prompts. Moreover, it stores the generated templates in a cache. For every new log message, the cache is queried to determine whether a corresponding log template has already been generated. This reduces the number of times LLMs have to be queried. Xu et al. [30] proposed the DivLog framework, which applies few-shot prompting with an optimized procedure for selecting examples for the prompt.

Ma et al. [22] proposed an LLM-based log parser, LLMParse, which employs four different LLMs: Flan-T5-small, Flan-T5-base, LLaMA-7B, and ChatGLM-6B. They investigated the effect of training size, model size, and pre-training on log parsing accuracy. They experimented only with models pre-trained on natural language tasks.

Le and Zhang [15] studied the ability of ChatGPT for log parsing under consideration of different prompting strategies. Instead of investigating different prompting strategies for ChatGPT, Mugdal

and Wouhaybi [23] considered seven prompts for different log-related tasks. Among others, they considered log parsing, anomaly detection, and log prediction tasks.

From the existing studies, we can observe that the number of different LLMs investigated is low, and there is no comparison between the state-of-the-art LLMs including the free-to-use and paid proprietary models. Moreover, there is no study exploring the analysis performance of the models pre-trained on both natural language and programming (code) language. We aim to fill this gap, by comparing six LLMs for the task of log parsing (see Section 4.3) including paid proprietary, free-to-use open models, and also code-specialized models.

### 2.2 Evaluating Log Parsing

To evaluate the performance of log parsers, several metrics have been proposed in the past. The two most frequently applied metrics are Parsing Accuracy (PA) [11, 15, 16, 30] and Grouping Accuracy (GA) [11, 15, 16, 19]. PA, sometimes called message level accuracy, determines whether log messages have been parsed correctly [5]. This includes the correct replacement of variables with placeholders. In particular, the extracted template needs to agree with the ground truth. GA, sometimes called RandIndex, determines whether log messages that correspond to the same template are parsed identically [6, 32]. Therefore, GA is not concerned with the correct parsing of templates, but rather with a consistent parsing of log messages.

Among the two metrics, GA has received criticism [13, 21, 24], leading to the proposal of other metrics for evaluating log parsers. Nedelkoski et al. [24] criticized grouping accuracy given that the extracted template could be completely different from the ground truth, as long as it is consistent for each log message belonging to the group. To combat this, they proposed the use of Levenshtein Edit Distance (ED), as a means of quantifying the similarity of extracted templates and ground truth. Edit distance has been adapted by LLM approaches [15, 16], both of which report the median edit distance. Petrescu et al. [25] reported the mean edit distance.

Given the relevance of string similarity metrics, such as ED, we set out to investigate the applicability of another similarity metric (i.e., longest common subsequence), which to our knowledge has not yet been investigated for log parsing. Moreover, we provide insight into the difference between reporting mean or median values for similarity metrics, as well as on normalizing their magnitude. Such choices can be of particular interest for the use of LLMs, which can generate diverse outputs (e.g., empty or overly long strings).

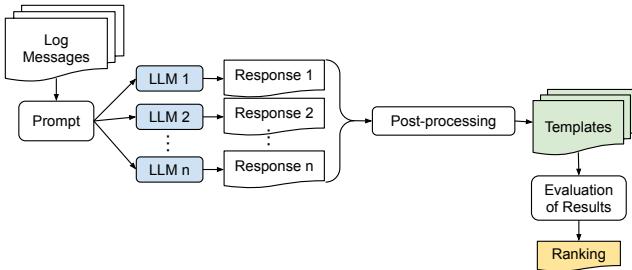


Figure 2: Overview of the experimental method.

### 3 Methodology

Figure 2 gives an overview of the methodology applied in this study. First, we design a prompt prefix that contains a general context about log parsing and instructions to perform log parsing, in zero-shot and few-shot settings (Figure 3). We then insert a log message, for which we want to extract a template for, into the designed prompt prefix and send this prompt to a selected LLM. After receiving the response from the LLM, we extract the template part from the entire response. Subsequently, we perform post-processing on the extracted templates to normalize them with standard placeholders. Afterward, we evaluate the results by comparing the generated templates with the ground truth in terms of their accuracy and syntactic similarity and rank the LLMs based on evaluation metrics.

**Prompt:** As a first step in our study, we design prompts following approaches that have been applied in related works: zero-shot [15, 23] and few-shot [11, 15, 16, 30]. We adopt the prompt introduced by Astekin et al. [1]. The prompts begins with a prefix, which consists of a general context about log parsing, instructions to perform the log parsing task, and an output indicator to specify the format of the extracted template. In zero-shot prompting, we use the prompt prefix as is. To use few-shot prompting, we provide four examples in the prompt prefix. The provided examples are constructed as Q&A pairs where each has two parts: the log message in the question and the corresponding template as an answer.

Figure 3 shows both prefixes, where the area shaded in green is only used in the few-shot prompts, and the remaining area is used for both zero- and few-shot prompting. Two of the examples are selected from the LogHub dataset [31] (Section 4.2). However, they are different from the actual log messages used as inputs in the experiment. The remaining two of these four examples are taken from unrelated datasets. These four examples remain the same for each log message.

To identify the extracted template in the response generated by the LLM, we add instructions to the prompt to generate customized responses with specific tags. We specify the format of the log templates to be extracted as surrounded by a <TPL> and </TPL> pair for each corresponding input log message. In a similar manner, we indicate the input log message to be parsed in the form of a <MSG> and </MSG> pair. We attach the log message to be parsed into the designed prompt prefix to construct the final form of the prompt. Each prompt has a single log message to be parsed as an input.

**Ask LLM:** We ask LLMs to extract log templates for each of the given log messages with both zero-shot and few-shot-prompts. To do this, we send requests to the selected set of LLMs for each

You will be provided with a log message delimited by <MSG> and </MSG>.

The log texts describe various system events in a software system. A log message usually contains a header that is automatically produced by the logging framework, including information such as timestamp, class, and logging level (INFO, DEBUG, WARN etc.).

The log message typically consists of two parts:

1. **Template** - message body, that contains constant strings (or keywords) describing the system events;
2. **Parameters/Variables** - dynamic variables, which reflect specific runtime status.

You must identify and abstract all the dynamic variables in the log message with suitable placeholders inside angle brackets to extract the corresponding template.

You must output the template corresponding to the log message. Print only the input log's template surrounded by <TPL> and </TPL>. Never print an explanation of how the template is constructed.

Here are a few examples of log messages (labeled with Q:) and corresponding templates (labeled with A:)

**Q:** <MSG>[081109 204453 34 INFO dfs.FSNamesystem: BLOCK\* NameSystem.addStoredBlock: blockMap updated: 10.250.11.85:50010 is added to blk\_2377150260128098806 size 67108864]</MSG>

**A:** <TPL>[BLOCK\* NameSystem.addStoredBlock: blockMap updated: <\*>:<\*> is added to <\*> size <\*>]</TPL>

**Q:** <MSG>- 1129734520 2005.10.19 R17-M0-N0-I:J18-U01 2005-10-19-08.08.40.058960 R17-M0-N0-I:J18-U01 RAS KERNEL INFO shutdown complete</MSG>

**A:** <TPL>shutdown complete</TPL>

**Q:** <MSG>20231114T101914E ERROR 14 while processing line 123: cannot find input '42'</MSG>

**A:** <TPL>ERROR <\*> while processing line <\*>: cannot find input <\*></TPL>

**Q:** <MSG>2023-01-14 23:05:14 INFO: Reading data from /user/input/file.txt</MSG>

**A:** <TPL>Reading data from <\*></TPL>

Here is the input log message:<MSG>...</MSG>  
Please print the corresponding template.

Figure 3: The prompt prefix. Few-shot examples are highlighted in green. The log messages are inserted in <MSG>...</MSG> at the bottom of the prompt.

prompt. We use the same prompt for all the LLMs and collect the generated responses.

**Post-processing:** After getting responses from the LLMs, we perform post-processing on the generated outputs to obtain the template from the responses. We extract the text enclosed in pairs of <TPL> and </TPL> as the raw template indicates. In some rare cases, the responses generated by the LLMs do not include the pairs of <TPL> and </TPL>, use either <TPL> or </TPL>, or contain multiple <TPL> and </TPL> pairs. Sometimes, the LLMs return

plain text without tags. In those cases, we manually search for the raw templates in the responses and extract them if present.

We perform normalization on the raw templates before evaluating them further. One challenge is that LLMs at times do not follow the instructions of the defined output format, and use different placeholders and various formatting for the dynamic variables in the log template. The placeholders for the dynamic variables in the log templates often have different names and formats. Even if we define the format to show the dynamic variables inside angle brackets, we encounter various formats (curly brackets, square brackets, dollar signs, etc.). To address this problem, we investigate all the raw templates to detect formatting issues and construct regular expressions to automatically convert them into the expected format.

**Evaluation of Results:** After obtaining normalized templates, we measure the quality of the extracted templates in two main aspects: *accuracy* and *syntactic similarity*. The *accuracy* is evaluated with the number of correctly parsed log messages, where a log message is considered correctly parsed only when each token in the log message is correctly recognized as a template or variable based on the ground truth. The *syntactic similarity* is measured by comparing the text similarity between the generated templates and ground truth. It is determined based on Edit Distance, Longest Common Subsequence, and their variants. More details about the metrics are provided in Section 4.4. We provide the ranking of LLMs in terms of each evaluation metric.

## 4 Experimental Design

### 4.1 Research Questions

We address the following research questions in our empirical study:

#### RQ1. How do the different LLMs perform for log parsing?

In the first research question, we set out to assess the performance of six LLMs (Section 4.3) based on three metrics (Section 4.4) on a real-world dataset (Section 4.2). We provide details on the performance of LLMs for each of the three metrics and determine which models perform best. Moreover, we compare the performance of paid and free-to-use models.

#### RQ2: What is the impact of the studied metrics on the ranking of LLMs?

After determining the best-performing LLMs according to three metrics in RQ1, we investigate how influenceable the ranking of LLMs by the investigated metrics is. For instance, the reported similarity metrics in RQ1 are reported based on median values, and RQ2 checks whether the ranking of LLMs changes when reporting mean values (see Section 4.4 for more details).

#### RQ3: How easy to use are the generated responses by the LLMs?

While RQ1 and RQ2 are concerned with the performance of LLMs, RQ3 is concerned with the usability. In particular, we are interested in how easy it is to use the generated outputs of LLMs to extract the intended part from the entire output. Therefore, we provided qualitative insights into the manual work involved in evaluating the outputs generated by each of the six LLMs.

**Table 1: Description of the LogHub dataset. Number of unique templates for the original and corrected datasets (Khan et al. [13]) are shown.**

Type	Project	# Original [31]	# Khan et al. [13]
Distributed Systems	HDFS	14	14
	Hadoop	114	114
	Spark	36	36
	Zookeeper	50	50
Super computers	OpenStack	43	43
	BGL	120	120
	HPC	46	46
Operating systems	Thunderbird	149	149
	Windows	50	50
	Linux	118	118
Mobile systems	Mac	341	341
	Android	166	158
Server applications	HealthApp	75	75
	Apache	6	6
Standalone software	OpenSSH	27	26
	Proxifier	8	8
Total		1363	1354

## 4.2 Dataset

To evaluate the log parsing performance of LLMs, we apply them to LogHub [31], a dataset of real-world log messages. These messages originate from different systems (i.e., distributed systems, supercomputers, operating systems, mobile systems, server applications, standalone software) distributed over 16 projects. For each of the 16 projects, LogHub provides 2,000 log messages each linked to a corresponding log template that should be extracted. In this context, multiple log messages of a project can have the same template.

An overview of the 16 projects and the unique number of log templates is shown in Table 1. In addition to listing the original number of unique log templates, we provide the number of templates from a modified LogHub version by Khan et al. [13]. Like related works [15, 16, 30], we consider the modified templates by Khan et al. [13]. Among others, they replaced hard-coded boolean values (*true*, *false*) with placeholders. This leads to a reduction of the total number of templates in the dataset from 1,363 to 1,354. Therefore, we carry out experiments with 1,354 log messages, one message for each of the templates. This allows us to investigate LLMs on a large variety of log templates while reducing the total number of log messages evaluated and keeping manual effort for the post-processing within reason.

## 4.3 Large Language Models

Table 2 presents an overview of the LLMs used in this study. The columns indicate the version, availability, and base model, in case it is built by fine-tuning another LLM. Since our goal is to explore the capability of state-of-the-art LLMs in the log parsing task, we select LLMs with different availability, parameter size, and fine-tuning options. The selected LLMs contain both closed-source and publicly accessible models, including base and fine-tuned models with different parameter sizes.

**Table 2: Overview of large language models.**

Model	Version/ Release time	Base model	Availability
GPT-3.5	gpt-3.5-turbo-0613/ June, 2023	-	Closed-source
Claude 2.1	November, 2023	-	Closed-source
Llama 2	llama2:7b/ July, 2023	-	Public
CodeLlama	codellama:7b-instruct/ August, 2023	Llama 2	Public
Zephyr Beta	zephyr:7b/ October, 2023	Mistral-7B	Public
CodeUp	codeup:13b/ July, 2023	Llama 2	Public

Of the six models, two (GPT-3.5<sup>1</sup> and Claude 2.1<sup>2</sup>) are proprietary, paid-for models that can be accessed via API calls. The other four LLMs (Llama2 [28], CodeLlama [27], Zephyr [29], CodeUp [10]) are publicly accessible and available for download from Hugging Face.<sup>3</sup> In our implementation, we utilized Ollama<sup>4</sup> to access the free-to-use models and used the APIs provided to access the paid proprietary models.<sup>5</sup>

Among the six models, GPT-3.5, Claude 2.1, and Llama2 are base models, while Zephyr, CodeUp, and CodeLlama are fine-tuned models. Zephyr is fine-tuned for improved chat and is stated as the best open-access RLHF-based model [29]. CodeUp and CodeLlama are fine-tuned for code generation tasks. Paid proprietary models contain a large number of parameters, where Claude 2.1 contains approximately 200 billion and GPT-3.5 has 175 billion parameters. The variants of Llama2, CodeLlama, and Zephyr are used in sizes of 7 billion parameters, and CodeUp with 13 billion parameters.

Lower temperatures make the LLMs more targeted and deterministic. Considering the trade-off between coherence and creativity for the log parsing task, we set their temperatures to 0.2 so that LLMs would generate more deterministic outputs for the same type of prompts.

#### 4.4 Evaluation Metrics

In accordance with related work, we use PA to evaluate the accuracy of LLMs for log parsing [11, 15, 16, 30]. Given the circumstance that we consider a single log message per template, we are not able to measure grouping accuracy. To measure the textual similarity of extracted templates  $T$  and the ground truth  $GT$ , we use ED and LCS.

In addition to reporting ED and LCS, we also consider their normalized variants. For this purpose, we compute Edit Similarity (ES) instead of Edit Distance, such that similarity is maximized at 1 and minimized at 0, in accordance with normalized LCS and PA:

$$ES_{norm} = 1 - (ED(T, GT) / \max(\text{len}(T), \text{len}(GT))) \quad (1)$$

$$LCS_{norm} = LCS(T, GT) / \text{len}(GT) \quad (2)$$

<sup>1</sup><https://platform.openai.com/docs/models/gpt-3-5>

<sup>2</sup><https://www.anthropic.com/news/clause-2-1>

<sup>3</sup><https://huggingface.co/>

<sup>4</sup><https://ollama.ai/>

<sup>5</sup>The costs for running these experiments were low: the GPT-3.5 experiments cost approx. USD 3, and the costs for Claude 2.1 were USD 1.19.

For RQ1, we report the median values for ED and LCS, in accordance with related works [15, 16]. For RQ2, we consider normalized and standard ED and LCS, as well as the median and mean values. To better compare normalized ED and LCS with PA, we report their sum rather than mean. In total, we consider four variants for each similarity metric: standard-median, standard-mean, normalized-median, normalized-sum.

#### 4.5 Threats to Validity

There are three factors impacting the threats to validity of our study: LLMs, dataset, and metrics.

We considered a total of six LLMs for our study, which extends the number of LLMs investigated in previous works. Four out of six LLMs are publicly accessible. We report the version we used to allow for reproducible results. Closed source models may not be as reproducible given their nature (i.e., GPT-3.5 and Claude 2). For generating outputs, we employed a temperature of 0.2. We note that this value is chosen arbitrarily, as we observed varying temperatures applied in research articles and Hugging Face implementations. To limit manual work, we focused on a single temperature but note that a deeper investigation of other values could be beneficial.

We measured a total of three metrics, two of which are applied in related works (ED, PA) and have shown useful in assessing log parsers. For LCS, which to the best of our knowledge is used to evaluate log parsing for the first time, we compare it to ED and provide insight on when the use of LCS is preferable (RQ2). This supports the applicability of LCS to log parsing.

The dataset we used is a subset of the popular LogHub dataset. Given the manual post-processing required when using LLMs (RQ3), we chose a single log message per available template to test LLMs on the largest variety of log templates, while keeping manual post-processing within reason.

### 5 Results and Discussion

#### 5.1 RQ1: Performance Comparison

**Parsing accuracy:** Table 3 compares the PA of the six LLMs. For this purpose, we sum up the number of correctly parsed templates for each of the 16 projects. We can observe that CodeLlama is the best performing LLM for log parsing based on PA, with 586 out of 1,354 log messages parsed correctly when using the few-shot prompt. The best performing LLM is CodeLlama for 9 out of 16 projects. The next best-performing LLMs are GPT3.5 and Claude 2, the two paid models, both under consideration of few-shot prompts.

Generally speaking, the results indicate that few-shot prompts allow for a better performance of LLMs in 5 out of 6 cases. Only for Zephyr, the zero-shot prompt is able to achieve a better performance than the use of few-shot prompts. One reason might be that Zephyr is not able to use the additional information but rather gets confused by the long text. Supporting this assumption, there are 56 cases where Zephyr-few does not return an output at all, and 364 cases where it returns the prompt template.

Moreover, none of the LLMs are able to correctly parse a log message for the Proxifier project. This could be explained by the low number of messages, given the 8 distinct log templates. However, this shows that PA does not always allow for a comparison between methods and highlights the importance of other metrics, such as

**Table 3: Parsing accuracy comparison.** The best values are highlighted in bold. For each model, we provide results for zero-shot (Zero) and few-shot (Few) prompts.

	GPT-3.5		Claude 2.1		Llama 2		CodeLlama		CodeUp		Zephyr	
	Zero	Few	Zero	Few	Zero	Few	Zero	Few	Zero	Few	Zero	Few
HDFS	0	2	3	<b>5</b>	2	3	2	2	2	3	1	1
Hadoop	16	39	30	<b>50</b>	15	10	<b>50</b>	43	15	22	8	6
Spark	10	<b>21</b>	13	20	11	11	18	17	9	20	7	2
Zookeeper	8	27	19	25	9	11	<b>31</b>	29	8	22	3	4
OpenStack	8	13	6	25	6	9	<b>28</b>	25	5	3	2	2
BGL	11	41	14	<b>66</b>	1	4	8	32	0	8	0	6
HPC	0	3	0	1	0	7	7	<b>10</b>	1	4	2	1
Thunderbird	18	45	13	21	0	30	19	<b>50</b>	19	37	0	0
Windows	17	<b>21</b>	15	15	1	13	13	19	0	3	12	0
Linux	41	64	29	48	26	41	60	<b>76</b>	41	66	20	12
Mac	96	125	105	55	29	53	106	<b>130</b>	62	105	64	25
Android	47	77	72	81	6	56	73	<b>91</b>	22	58	5	3
HealthApp	28	33	21	19	0	25	39	<b>47</b>	6	34	2	1
Apache	1	<b>5</b>	0	0	0	0	4	3	1	0	0	0
OpenSSH	8	<b>15</b>	6	8	1	8	12	12	2	8	0	0
Proxifier	0	0	0	0	0	0	0	0	0	0	0	0
Total	309	531	346	439	107	281	470	<b>586</b>	193	393	126	63

**Table 4: Edit distance and longest common subsequence comparison.** This table shows median values, with the best scores highlighted in bold. Edit distance is to be minimized; longest common subsequence is to be maximized. For each model, we provide results for zero-shot (Zero) and few-shot (Few) prompts.

	Edit Distance - Median (small is better)						Longest Common Subsequence - Median (large is better)					
	GPT-3.5 Zero	GPT-3.5 Few	Claude 2.1 Zero	Claude 2.1 Few	Llama 2 Zero	Llama 2 Few	CodeLlama Zero	CodeLlama Few	CodeUp Zero	CodeUp Few	Zephyr Zero	Zephyr Few
HDFS	13.0	11.0	8.0	<b>3.0</b>	23.0	8.5	4.0	8.0	18.0	6.0	29.0	15.5
Hadoop	22.0	8.0	8.0	4.0	26.5	13.0	<b>3.5</b>	<b>3.5</b>	70.5	15.0	55.0	59.5
Spark	15.5	<b>0.0</b>	6.5	<b>0.0</b>	20.0	5.5	1.5	2.5	44.5	26.5	28.0	
Zookeeper	3.5	<b>0.0</b>	3.5	<b>0.0</b>	29.5	9.0	<b>0.0</b>	<b>0.0</b>	89.0	3.0	51.5	38.5
OpenStack	16.0	16.0	85.0	<b>0.0</b>	36.0	18.0	<b>0.0</b>	<b>0.0</b>	34.0	18.0	71.0	43.0
BGL	20.0	3.0	63.0	<b>0.0</b>	44.0	13.0	20.0	3.0	51.5	12.0	49.5	44.5
HPC	33.5	24.5	38.0	29.0	31.0	<b>18.0</b>	22.0	22.5	35.0	25.0	37.5	34.0
Thunderbird	10.0	6.0	13.0	12.0	26.0	8.0	9.0	<b>5.0</b>	19.0	10.0	25.0	33.0
Windows	10.0	<b>4.5</b>	6.5	7.5	37.5	10.0	7.0	5.0	50.0	13.5	28.0	36.5
Linux	3.5	<b>0.0</b>	9.0	4.5	17.0	5.0	<b>0.0</b>	<b>0.0</b>	5.5	<b>0.0</b>	30.0	30.0
Mac	10.0	<b>6.0</b>	14.0	22.0	38.0	24.0	14.0	7.0	32.0	10.0	40.0	43.0
Android	6.0	<b>0.0</b>	3.0	<b>0.0</b>	25.0	4.0	2.0	<b>0.0</b>	17.0	4.0	25.0	34.0
HealthApp	3.0	2.0	20.0	18.0	35.0	5.0	<b>0.0</b>	<b>0.0</b>	49.0	1.0	49.0	42.0
Apache	8.5	<b>0.0</b>	14.5	12.0	33.0	28.0	<b>0.0</b>	4.0	31.0	8.5	17.5	35.0
OpenSSH	7.0	<b>0.0</b>	14.0	11.0	41.0	15.5	6.0	3.0	20.5	7.0	42.0	45.0
Proxifier	48.5	23.0	17.0	25.5	23.5	20.5	18.0	<b>16.0</b>	52.0	54.0	45.5	69.5
Total	11.0	4.0	14.0	11.0	31.0	11.0	7.0	<b>3.0</b>	36.0	9.0	35.0	37.0
									38.0	<b>39.0</b>	<b>39.0</b>	38.0
									12.0	35.0	37.0	38.0
										35.0	38.0	33.0
										33.0	28.0	

ED and LCS. With these two metrics, one can assess how good the extracted templates are, even when none of them is 100% correct.

CodeLlama is the best model based on Parsing Accuracy.

**Edit distance and Longest common subsequence:** Table 4 presents the ED and LCS results for the six LLMs. We report median values for each of the 16 projects, as well as the median overall 1,354 in the “Total” column. Here, ED should be minimized, whereas LCS should be maximized to indicate good performance.

The left half of Table 4 shows results for ED. Similar to PA, the best performing LLM is CodeLlama, with a median ED of 3 and the best results on 9 out of 16 projects. CodeLlama is followed by GPT3.5-few, which has a median ED of 4. Unlike the results for

PA, the third ranking LLM is CodeUp-few, performing better than Claude 2.

We can also observe that the use of median ED values leads to ties among the LLMs, in particular when an LLM is able to extract more than 50% of the templates correctly, leading to an ED of 0.

CodeLlama is the best performing model based on Edit Distance.

The right half of Table 4 presents the median LCS for the six LLMs over the 16 projects. We observe a tie for the best performing model, with GPT3.5-few and Claude 2-zero exhibiting a median LCS of 39. These LLMs are followed by CodeLlama-few, CodeUp-few, GPT3.5-zero and Claude 2-few, all of which share an LCS of 38. The lowest LCS is achieved by Llama 2 and Zephyr.

Similar to the reported median values for ED, we observe ties among the LLMs for LCS as well. In this case, ties happen for the best performing models, which makes it harder to distinguish LLMs based on their performance.

Paid models (GPT-3.5, Claude 2.1) are the best performing models based on Longest Common Subsequence.

**Paid vs. free-to-use LLMs:** CodeLlama, a free model, is the best performing on metrics employed by existing work: PA, ED. Paid models, in particular GPT-3.5, are only able to outperform free-to-use models when measuring LCS, to our knowledge first used for log parsing in this publication. CodeLlama shares the second rank for LCS and would therefore be a feasible option as well. In particular, free-to-use LLMs trained on code have achieved good results (CodeUp and CodeLlama), outperforming Llama 2 and Zephyr.

This shows that one does not need to spend money on paid LLMs to research log parsing, as free models, such as CodeLlama can even outperform them for two out of three metrics in our experiments. The only downside of using free models is that one needs to be able to load them (e.g., CodeLlama has 7 billion parameters) and have a modern processing unit, such as a GPU, to obtain results quicker. If hardware is a limiting factor, GPT3.5 and Claude 2.1 are promising alternatives.

Free models, such as CodeLlama, are a high-performing alternative to paid APIs for log parsing.

## 5.2 RQ2: Impact of Metrics

**Ranking Stability:** First, we investigate the ranking of LLMs according to their performance over different metrics and whether their rank changes. In total, we report nine metrics: PA, four variants of ED, and four variants of LCS. The four variants of ED and LCS are median, mean, normalized median, and normalized mean (Section 4.4). This allows us to investigate ranking differences between metrics (PA, ED, LCS) and differences based on reporting choices (mean, median, normalizing), all of which can impact which LLM is perceived as best performing.

The ranking of each of the 6 LLMs, given two prompting types (i.e., zero- and few-shot), for each of the nine metrics is shown in Figure 4. Here, rank 1 signifies the best model, while rank 12 is the worst-performing one. We observe that the results for best and worst performing LLMs are consistent over similarity metrics. The best performing LLM for all four ED variants is CodeLlama-few. GPT3.5 is the highest ranking model for LCS; in two out of four cases there is a tie for first place with Claude 2.1. The worst model for each metric is either Zephyr or Llama 2.

The ranking over different metrics can be stable with some models deviating by 2-3 ranks (i.e., Zephyr-few, CodeUp-few, CodeUp-zero, CodeLlama-few) over all 9 computed metrics. However, we can also notice high deviations, in particular ones caused by computing mean values. For example, GPT3.5-few, which ranks among the 1st-4th best models for 8 out of 9 metrics, is ranked 11th when computing mean edit distances. GPT3.5-zero which was 3rd ranked

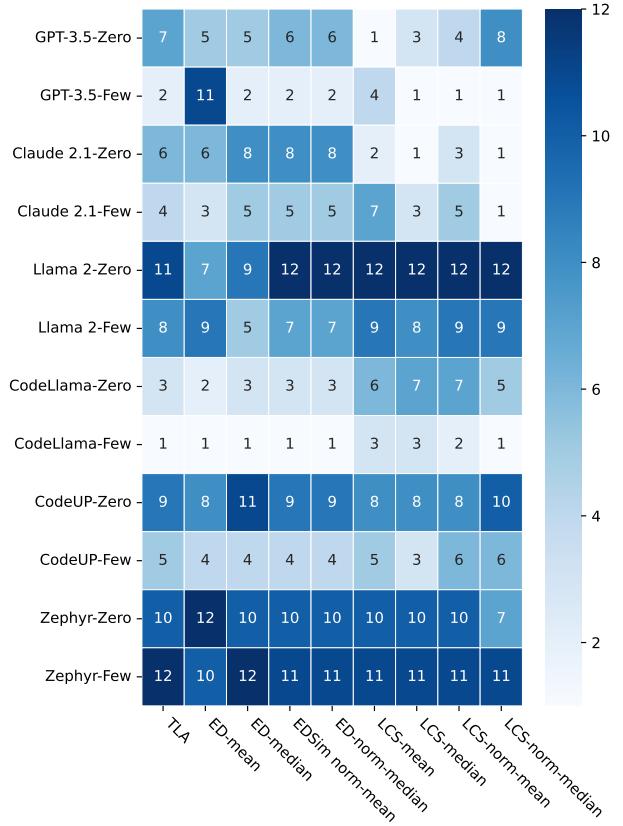


Figure 4: Ranking based on metrics.

at best for 8 metrics, is the best on LCS-mean. Such deviations when computing means can be caused by outliers. For GPT3.5-few, this happens on the Hadoop project, where the following template is extracted:

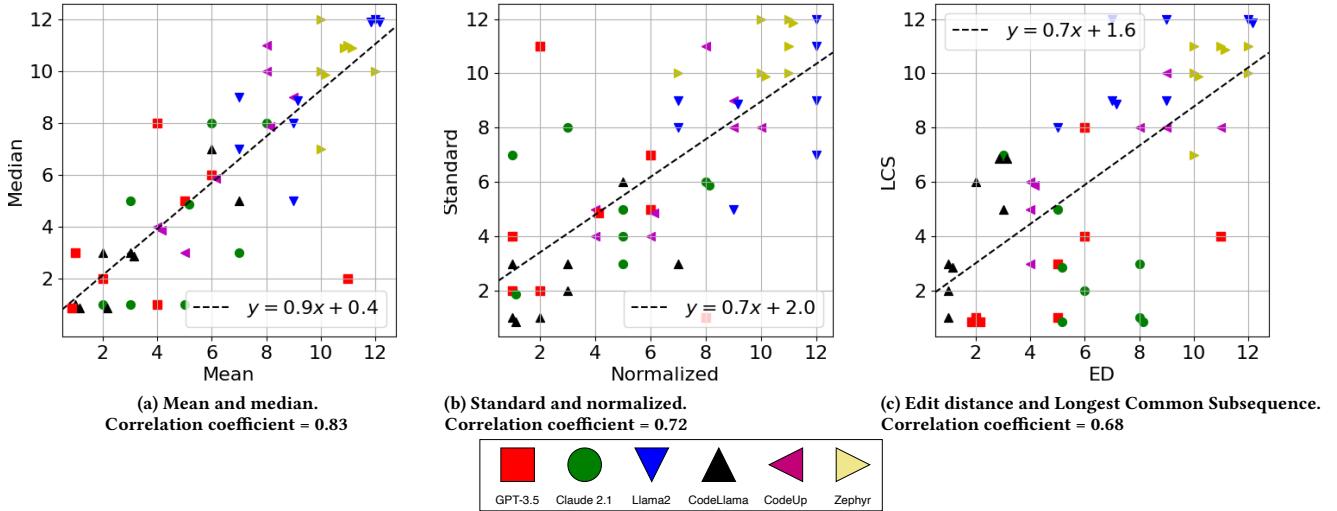
```
attempt_<*><*><*><*><*><*><*><*> ... <*><*>
```

<\*> is repeated multiple times such that the edit distance for a single template went up to 5,272. Reporting normalized mean values for ED and LCS does not suffer from such outliers, as values are in a range of [0,1].

Do not report mean values for LLMs without normalizing.

**Metric comparison:** Next, we take a closer look at what choices can impact ranking deviations. In particular, we consider three comparisons for similarity metrics: ED and LCS, mean and median, standard and normalized. The purpose of these comparisons is to determine whether and how much of a difference the corresponding choices have on the results (i.e., overall rank) of the LLMs.

Figure 5 visualizes the three comparisons. For each comparison, we compare the ranks of LLMs based on the two options, each determined by one of the axes in the plots. For each model, we visualize 8 markers ( $2^4$ ), for the two prompt types and four metric



**Figure 5: Comparison of different aspects based on ranking with Pearson correlation coefficient. Colour and shape show the different LLMs.**

variants of ED and LCS. Moreover, we report the Pearson correlation coefficient between ranks over all LLMs. If we find a strongly correlated pair, the choice between both alternatives (e.g., mean or median) would not be critical for the results. If however there is no correlation between the two options, then one needs to be careful when picking metrics for evaluation.

Median and mean have the highest correlation even though the mean is impacted by outliers, such as the ones observed for GPT3.5 (i.e., a single log message with ED of 5,272). For traditional approaches, e.g. regex over log messages, it might not make a difference which to use, but due to the stochastic nature of LLMs it is preferred to use median over mean to avoid outliers (or normalized mean for this purpose). One disadvantage that can be avoided by using mean values is the lower chance of ties when ranking LLMs, in particular when more than 50% of the templates are extracted correctly. For traditional approaches, Petrescu et al. [25] reported mean ED values.

Median loses meaning when more than 50% of the log messages are solved correctly. Mean values take every log message into account.

The ranks of standard and normalized values have a Pearson correlation coefficient of 0.72. Similar to the comparison of mean and median, this can be impacted by outliers, with normalized values being bound within a range of [0,1]. By normalizing LCS and ED, each log message has the same level of impact, independent of length. This is similar to PA, where a correctly extracted template counts as 1 while an incorrect template as 0. Using standard ED measures, an empty message corresponding to a long ground truth template would receive a worse score than a short one, even though they are incorrect to the same degree.

```
example1 = ""
template1 = queue: default
ED = 14
EDnorm = 1

example2 = ""
template2 = ciod: Error creating node map
           from file <*>: No child processes
ED = 63
EDnorm = 1
```

Normalizing avoids outliers and gives every log message the same importance.

ED and LCS are the least correlated pair in our comparison, which can be explained by differences in dealing with timestamp/message scope as well as placeholders:

```
example1 = CBS SQM: Cleaning up report files
           older than <*> days.
template1 = SQM: Cleaning up report files
           older than <*> days.
ESnorm = 0.88
LCSnorm = 1

example2= setDataSource(<*>, <*>, <*>)
template2= setDataSource(166, 0, 576460752303423487)
ESnorm = 0.41
LCSnorm = 0.68
```

From this example, we can see that LCS ignores timestamps (excess tokens as long as the ground truth is part of the message) while ED would remove these. Note that we report ES to better

compare with LCS, as both are maximized at 1. When an LLM fails to replace a variable by the placeholder <\*>, LCS is not impacted by the length of the variable, ED on the other hand is. The use of LCS and ED should therefore be made based on the application scenario, or both can be considered.

Edit distance is preferred when timestamps should be removed (scope). Longest common subsequence is preferred when the detection of placeholders is of higher importance.

### 5.3 RQ3: Usability of Responses

We investigate how usable the LLM responses are when extracting log templates. The usability of a response is a qualitative indication of the clarity to distinguish the log template in the correct format from the generated response. Even though we explicitly describe the expected response format in the prompt, there are still many responses that do not meet the specifications (e.g., responses with <TPL> and </TPL>).

It is a challenging task to find the placeholders in the generated log templates since LLMs produce them in different formats and names than defined in the prompt. In the zero-shot prompt, we do not specify the placeholder for the dynamic variables, we only instruct to print the dynamic variables with suitable placeholders inside angle brackets. In this way, we aim to observe their preference for finding suitable placeholder names. However, we observe that none of the LLMs are able to follow the formatting of angle brackets <> consistently in zero-shot prompting. GPT-3.5 zero-shot experiment produces the best results, following the formatting of angle brackets <> with only 7 exceptions of log messages. Other models use a variety of brackets such as (double) curly brackets, (double) square brackets, tags enclosed in left and right angle brackets, @ sign, dollar sign, etc. More details on the usability of responses with zero-shot prompting are provided in Section 5.3.1.

In the few-shot experiments, we give four examples of the expected output (<\*>) while we still instruct LLMs to print the variables inside angle brackets. With these examples, GPT-3.5, Claude 2.1, CodeLlama, and CodeUp are able to format variables using <\*> with only a few exceptions. More details about the few-shot experiments of the LLMs are provided in Section 5.3.2.

We present our observations from both zero-shot and few-shot experiments in terms of two aspects: LLMs' placeholder selection and variable formatting.

#### 5.3.1 Observations from zero-shot experiments.

**Placeholder selection:** GPT-3.5 and Claude 2.1 predominantly employ generic placeholder names like VAR, variable1, and variable2. Frequently, they opt for the repetition of preceding identifier names as placeholders, as exemplified by constructs such as lock = <lock> and flag = <flag>. Besides that, they are able to generate appropriate placeholder names such as pid, block\_id, ip, and port. In the majority of responses, CodeLlama and CodeUp employ suitable placeholder names such as class\_name, method\_name, hostname, instance\_id, and version. However, sporadically, they resort to utilizing generic placeholders and irrelevant designations such as TPL, and MSG. Notably, CodeLlama generated a generic template (i.e. \$DATE-\$TIME---\$EVENT) in one instance, and in another, it

omitted the template in the response while listing dynamic variables. Llama 2 primarily parses log messages in accordance with a generic log message format (e.g. <timestamp> <class> <process\_id> <message\_body>), and occasionally, it produces generic names for variables such as value, variable, and var. Moreover, it occasionally generates irrelevant responses in HTML format. Zephyr frequently opts for generic variable names such as dynamic\_variable\_1, VAR, TIMESTAMP, although it rarely assigns more appropriate names like NODE\_ID, APP\_ID. Additionally, it generates timestamp formats in the YYYY-MM-dd HH:mm:ss structure uses [0-9] for integers, and employs placeholders such as XX, YY, ###, \_\_\_, \*. For 179 log messages, it responded with the same prompt provided as an input, and it did not send any responses for 9 log messages.

**Variable formatting:** Table 5 presents different variable formats generated in the responses by the LLMs in the zero-shot and few-shot experiments.

GPT-3.5 mostly employs angle brackets to surround variables in accordance with the instructions provided in the prompt. However, it occasionally diverges from this convention by utilizing alternative formats such as {}, <TPL>, or employing opening and closing tags. Similarly, CodeUp primarily adopts angle brackets for variable formatting but may follow different formatting in some cases. Claude 2.1 demonstrates a broader spectrum of bracketing styles and formatting techniques. While it commonly surrounds dynamic variables with angle brackets, it also incorporates other listed formats. Notably, there are instances where variables are omitted entirely without replacement with a placeholder. CodeLlama mainly uses curly brackets to surround variables, whereas a minority adheres to the convention of angle brackets. Rarely, it employs additional formatting styles outlined in Table 5 to surround the placeholders.

Llama 2 exhibits a wide range of bracketing styles and formatting variations for placeholders. Notably, within a single log message, it may employ different formats for variables. Zephyr's behavior regarding variable formatting is characterized by instability and inconsistency, as Zephyr uses various formats for dynamic variables. In some cases, the variable is either omitted entirely from the message or replaced with placeholders, such as \_\_\_. In rare cases, it capitalizes the name of the variable without formatting any further. Furthermore, Zephyr occasionally diverges from a consistent formatting approach within a single log message, resulting in a mixture of various formats across different variables of the same log message. These non-deterministic tendencies present considerable challenges in discerning between templates and variables within responses, thus requiring strenuous effort to extract templates.

#### 5.3.2 Observations from few-shot experiments.

**Placeholder selection:** In the majority of responses, GPT-3.5, Claude 2.1, CodeLlama, and CodeUp generate \* for the dynamic variables as shown in the examples of the few-shot prompt. However, in rare cases, they produce variable names instead of using the \* sign. CodeUp exhibits a distinct behavior, where it removes only the header part (timestamp, etc.) from log messages and presents them as templates, particularly observed in the BLG and Mac projects.

The Llama 2 responses with few-shot prompting are easier to use than their zero-shot counterparts. It mostly uses a \* sign for variables, and variable names in some cases. However, it also produced

**Table 5: Variable formats followed by LLMs.**

Model	Zero-shot	Few-shot
GPT-3.5	<>, {}, X, <TPL>, opening-closing tag pairs	< * >, <>
Claude 2.1	<>, {}, \${}, @ @, \$ \$, {}, [], opening-closing tag pairs	< * >, *
Llama 2	\${}, <>, \${}, \$ \$, [], []+, \${}, (), %, ?, ** **, #, no closing brackets, opening-closing tag pairs	{}, *, <>, < * >, < *
CodeLlama	{}, <>, \${}, \$, %, {}, [], opening-closing tag pairs	< * >, <>, *, < ** >
Zephyr Beta	<>, {}, {}, [], [] %, #, \$, \$ \$, X _____, opening-closing tag pairs	{}, ' ', [], {{}}, X *, < ** >, < * >, < *
CodeUp	<>, {}, {}, %, \${}, [], <>>, <> %, [ ], #, & no closing brackets, opening-closing tag pairs	< * >, *, < *, ...

a few plain messages without any templates and some templates with repetitive placeholders (i.e. < \* >< \* >< \* > .... < \* >).

Zephyr returned the input prompt as a response for 364 log messages, while it yielded empty responses for 56 log messages. Occasionally, it generates generic templates, exemplified by structures such as <\*CLASS\*> <\*LOGGER\*> <\*MESSAGE\*> <\*VARIABLE\*>. Notably, responses generated for log messages originating from the BGL project were occasionally intertwined with HDFS log templates, a few of which were originally shared as examples in the prompt. Throughout the remainder of its responses, it mostly uses \*, and sometimes variable names or alternative designations like XXX or YYYY-MM-DD.

**Variable formatting:** GPT-3.5, Claude 2.1, CodeLlama, and CodeUp mostly generate angle brackets to surround the dynamic variables as shown in the examples of the few-shot prompt. In some rare cases, they do not use brackets and just show \* or <>. In certain results from CodeUp, closing brackets are omitted for the variables (i.e. < \*). While Llama and Zephyr primarily employ angle brackets for variable notation, they also incorporate different formats, as outlined in Table 5.

For an automated log parsing procedure, it is crucial that the template and variables can be easily parsed. Therefore, we recommend the use of CodeLlama and CodeUp as free-to-use models when enough processing capabilities are available. Otherwise, GPT3.5 creates results that are easy to parse and cheaper than Claude 2.1.

Llama 2 and Zephyr require a considerable amount of manual post-processing effort and are therefore not recommended. CodeLlama and CodeUp are recommended as free-to-use models, while GPT-3.5 is a cheaper alternative among the paid models.

## 6 Concluding Remarks

In this paper, we carried out an empirical study to compare the performance of six LLMs (2 paid, 4 free-to-use) for log parsing. We evaluated their performance on three metrics and 1,354 log messages, each with a different underlying template. Our findings show that among the six LLMs, CodeLlama and GPT3.5 perform the best. CodeLlama is the best performing model for parsing accuracy

and edit distance, two metrics applied in existing work. When considering the longest common subsequence as a novel metric to determine the quality of the extracted templates based on string similarity, GPT-3.5 performs best. While both ED and LCS are valid metrics that can be chosen based on the preferences/requirements for a given log parsing task (e.g., how important is the removal of timestamps), we cannot recommend one over the other. However, when it comes to the calculation of ED and LCS over a collection of log messages, we believe that the sum over normalized similarities is preferred, as this takes every message into account (unlike mean values) and outliers are scaled within the bounds of [0,1].

These results support future work in performing log parsing research without incurring costs (provided that hardware is available). Practitioners can have high performance and increased privacy for log parsing by using CodeLlama. Moreover, we highlight the importance of selecting the correct metrics for evaluating log parsing approaches, according to one's needs.

Another field of research that opens up is the combination of paid and free models. It can be interesting to determine the cases where free-to-use models are “good enough” to extract the template from a given log message and only use paid models for a fraction of log messages that were not parsed correctly. Lastly, while we investigated two prompt types, further comparisons with different prompts and few-shot examples could be conducted to improve the performance of LLMs.

## Data Availability

To support open science and allow for replication and verification of our work, all artifacts are made available through Zenodo at the following URL: <https://doi.org/10.5281/zenodo.13625383>.

## Acknowledgments

This work is supported by the Research Council of Norway through the cureIT (IKTPLUSS #300461) and the secureIT projects (IKT-PLUSS #288787), and by the European Union through the Horizon Europe Marie Skłodowska-Curie Actions (#101151798). The empirical evaluation was performed on the Experimental Infrastructure for Exploration of Exascale Computing (eX3), financially supported by the Research Council of Norway under contract #270053.

## References

- [1] Merve Astekin, Max Hort, and Leon Moonen. 2024. An Exploratory Study on How Non-Determinism in Large Language Models Affects Log Parsing. In *Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering*. 13–18.
- [2] Merve Astekin, Harun Zengin, and Hasan Sözer. 2018. DILAF: A framework for distributed analysis of large-scale system logs for anomaly detection. *Software: Practice and Experience* 49 (2018), 153 – 170. <https://api.semanticscholar.org/CorpusID:59222641>
- [3] Song Chen and Hai Liao. 2022. BERT-Log: Anomaly Detection for System Logs Based on Pre-trained Language Model. *Applied Artificial Intelligence* 36, 1 (Dec. 2022), 2145642. <https://doi.org/10.1080/08839514.2022.2145642>
- [4] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2022. Logram: Efficient Log Parsing Using Nn-Gram Dictionaries. *IEEE Transactions on Software Engineering* 48, 3 (March 2022), 879–892. <https://doi.org/10.1109/TSE.2020.3007554>
- [5] Hetong Dai, Heng Li, Weiyi Shang, Tse-Hsun Chen, and Che-Shao Chen. 2020. Lograma: Efficient Log Parsing Using n-Gram Dictionaries. arXiv:2001.03038 [cs]
- [6] Min Du and Feifei Li. 2016. Spell: Streaming Parsing of System Event Logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. 859–864. <https://doi.org/10.1109/ICDM.2016.0103>
- [7] Haixuan Guo, Shuhuan Yuan, and Xintao Wu. 2021. LogBERT: Log Anomaly Detection via BERT. In *2021 International Joint Conference on Neural Networks (IJCNN)*. 1–8. <https://doi.org/10.1109/IJCNN52387.2021.9534113>
- [8] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *International Conference on Web Services (ICWS)*. IEEE, 33–40. <https://doi.org/10.1109/icws.2017.13>
- [9] Yintong Huo, Yuxin Su, Cheryl Lee, and Michael R. Lyu. 2023. SemParser: A Semantic Parser for Log Analysis. arXiv:2112.12636 [cs]
- [10] Juyoung Jiang and Sungjun Kim. 2023. CodeUp: A Multilingual Code Generation Llama2 Model with Parameter-Efficient Instruction-Tuning.
- [11] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhenn Gu, and Michael R. Lyu. 2024. LILAC: Log Parsing Using LLMs with Adaptive Parsing Cache. arXiv:2310.01796 [cs]
- [12] Zhihan Jiang, Jinyang Liu, Junjie Huang, Yichen Li, Yintong Huo, Jiazhenn Gu, Zhuangbin Chen, Jieming Zhu, and Michael R. Lyu. 2024. A Large-Scale Evaluation for Log Parsing Techniques: How Far Are We? <https://doi.org/10.48550/arXiv.2308.10828>
- [13] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel Briand. 2022. Guidelines for Assessing the Accuracy of Log Message Template Identification Techniques. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 1095–1106. <https://doi.org/10.1145/3510003.3510101>
- [14] Van-Hoang Le and Hongyu Zhang. 2022. Log-Based Anomaly Detection with Deep Learning: How Far Are We?. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1356–1367. <https://doi.org/10.1145/3510003.3510155>
- [15] Van-Hoang Le and Hongyu Zhang. 2023. Log Parsing: How Far Can ChatGPT Go?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Luxembourg, Luxembourg, 1699–1704. <https://doi.org/10.1109/ase56229.2023.00206>
- [16] Van-Hoang Le and Hongyu Zhang. 2023. Log Parsing with Prompt-Based Few-Shot Learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 2438–2449. <https://doi.org/10.1109/icse48619.2023.00204>
- [17] Yukyung Lee, Jina Kim, and Pilсung Kang. 2023. LAnoBERT: System Log Anomaly Detection Based on BERT Masked Language Model. *Applied Soft Computing* 146 (Oct. 2023), 110689. <https://doi.org/10.1016/j.asoc.2023.110689>
- [18] Zhenhao Li, Chuan Luo, Tse-Hsun (Peter) Chen, Weiyi Shang, Shilin He, Qingwei Lin, and Dongmei Zhang. 2023. Did We Miss Something Important? Studying and Exploring Variable-Aware Log Abstraction. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 830–842. <https://doi.org/10.1109/ICSE48619.2023.00078>
- [19] Yilun Liu, Shimin Tao, Weibin Meng, Jingyu Wang, Wenbing Ma, Yanqing Zhao, Yuhang Chen, Hao Yang, Yanfei Jiang, and Xun Chen. 2023. LogPrompt: Prompt Engineering Towards Zero-Shot and Interpretable Log Analysis. <https://doi.org/10.48550/arXiv.2308.07610> arXiv:2308.07610 [cs]
- [20] Yilun Liu, Shimin Tao, Weibin Meng, Jingyu Wang, Wenbing Ma, Yanqing Zhao, Yuhang Chen, Hao Yang, Yanfei Jiang, and Xun Chen. 2024. Interpretable Online Log Analysis Using Large Language Models with Prompt Strategies. <https://doi.org/10.48550/arXiv.2308.07610> arXiv:2308.07610 [cs]
- [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. In *Proceedings of the ACM Web Conference 2022*. 1893–1901. <https://doi.org/10.1145/3485447.3511993> arXiv:2202.06569 [cs]
- [22] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. LLMParse: An Exploratory Study on Using Large Language Models for Log Parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. <https://doi.org/10.1145/3597503.3639150>
- [23] Priyanka Mudgal and Rita Wouhaybi. 2023. An Assessment of ChatGPT on Log Data. <https://doi.org/10.48550/arXiv.2309.07938> arXiv:2309.07938 [cs]
- [24] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2021. Self-Supervised Log Parsing. *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track* 12460 (2021), 122–138. [https://doi.org/10.1007/978-3-03-67667-4\\_8](https://doi.org/10.1007/978-3-03-67667-4_8)
- [25] Stefan Petrescu, Floris den Hengst, Alexandru Uta, and Jan S. Rellermeyer. 2023. Log Parsing Evaluation in the Era of Modern Software Systems. arXiv:2308.09003 [cs]
- [26] Jiaxing Qi, Shaohan Huang, Zhongzhi Luan, Carol Fung, Hailong Yang, and Depei Qian. 2023. LogGPT: Exploring ChatGPT for Log-Based Anomaly Detection. <https://doi.org/10.48550/arXiv.2309.01189> arXiv:2309.01189 [cs]
- [27] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaojing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémie Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs]
- [28] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esioubi, Jude Fernandes, Jeremy Fu, Wenjin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madiam Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molbyog, Yixin Nie, Andrew Poultney, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Ilyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs]
- [29] Lewis Tunstall, Edward Beeching, Nathan Lambert, Nazneen Rajani, Kashif Rasul, Younes Belkada, Shengyi Huang, Leandro von Werra, Clémentine Fourrier, Nathan Habib, Nathan Sarrazin, Omar Sanseviero, Alexander M. Rush, and Thomas Wolf. 2023. Zephyr: Direct Distillation of LM Alignment. arXiv:2310.16944 [cs]
- [30] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2024. DivLog: Log Parsing with Prompt Enhanced In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–12. <https://doi.org/10.1145/3597503.3639155>
- [31] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. 2023. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. arXiv:2008.06448 [cs]
- [32] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2019. Tools and Benchmarks for Automated Log Parsing. In *ICSE: Software Engineering in Practice (ICSE-SEIP)*. IEEE, Montreal, QC, Canada, 121–130. <https://doi.org/10.1109/icse-seip.2019.00021>