Log Parsing with LLMs Featuring Self-Reflection and Continuous Refining

Xiaolei Chen[1 [0009−0000−6136−982X]], Jia Chen✉[1 [0000−0001−9883−6356]], Jie Shi[1 [0009−0007−7413−6735]], Peng Wang[1 [0000−0002−8136−9621]], and Wei Wang[1 [0000−0003−0264−788X]]

School of Computer Science, Fudan University, Shanghai, China
{chenxl21,jshi22}@m.fudan.edu.cn,
{jiachen,pengwang5,weiwang1}@fudan.edu.cn

Abstract. Log parsing is a critical step in analyzing logs from large-scale software systems, enabling tasks like anomaly detection and network failure monitoring. Existing approaches, including statistical and deep learning-based methods, often lack semantic understanding or struggle with unseen data. Recently, large language models (LLMs) have shown promise in high-precision log parsing but face two major challenges: misidentification of constants and variables, and narrow-scope processing that overlooks contextual relationships across logs. To address these challenges, we propose LogReflex, an online log parsing framework leveraging LLMs. LogReflex introduces a self-reflection mechanism to correct misidentified constants and variables within extracted templates and a template refining mechanism to incorporate historical templates into the LLM query, enhancing contextual understanding while maintaining online parsing capabilities. The framework integrates five key components, including lightweight modules to improve accuracy with minimal system overhead. Extensive experiments on benchmark datasets demonstrate that LogReflex outperforms state-of-the-art approaches.

Keywords: Log Parsing · Log Analysis · Large Language Model

1 Introduction

Large-scale software systems are integral to modern society, making their reliability and stability critical. Logs, containing key insights into software operations, are essential for tasks like anomaly detection and failure monitoring. However, their semi-structured nature complicates direct use. Log parsing addresses this by converting logs into structured templates [14] and variables.

Approaches to log parsing include offline ones like IPLoM [11], which process logs in batches, and online ones like Drain [3], which parse logs sequentially. These rely on statistical features like token length and frequency [8] but lack semantic understanding, causing inaccuracies [5]. Deep learning-based approaches [8] enhance semantic recognition but require large labeled datasets and struggle with unseen data [13].

Recently, large language models (LLMs) have showcased exceptional capabilities. Through extensive training, they have amassed vast knowledge, enabling groundbreaking advancements across various domains. This opens new possibilities for leveraging LLMs in log parsing. Approaches like DivLog [13] and LILAC [6] harness LLMs' strengths, such as in-context learning, to achieve high-precision parsing. By providing few-shot examples, these methods guide LLMs to parse logs effectively in a single query.

However, large language models do not always generate the best output on their first try like humans [10]. Despite the strong contextual understanding capabilities, large language models may still mistakenly identify some constants as variables or some variables as constants during parsing, due to the diversity and semantic richness of logs. Therefore, existing LLM-based approaches possess the following two major problems.

- *Misidentification problem.* They use large language models to directly parse logs in a single query. If the extracted template contains misidentified constants or variables, they cannot provide further detailed corrections.
- *Narrow-scope problem.* They parse each log entry in isolation, ignoring the context provided by previously parsed logs. This limitation prevents the model from leveraging inter-log relationships, which can result in inconsistencies and reduced accuracy in capturing patterns or dependencies.

To tackle the aforementioned problems, we propose LogReflex, an online log parsing framework based on the large language model. For the misidentification problem, we design a self reflection mechanism, which allows in-depth reflection and correction of misidentified constants and variables in an extracted template. For the narrow-scope problem, we design a template refining mechanism that allows LogReflex to incorporate historical logs into the LLM query during the parsing process while maintaining online parsing capabilities.

Specifically, LogReflex comprises five main components: template matching, initial parsing, template validating, self-reflection, and template refining. Template matching directly infers the template for logs based on previously extracted templates, improving parsing efficiency. Initial parsing generates an initial template for a log using the large language model. Following this, we designed three lightweight modules to improve parsing accuracy while maintaining online capabilities and introducing minimal additional system overhead. Template validating rectifies discrepancies between the log and its initial template. Self-reflection reviews whether the extracted template contains misidentified constants and variables. Template refining broadens the scope of the LLM query by incorporating historical templates to provide a more comprehensive context. In summary, the following contributions are made in this paper.

- We introduce a self reflection mechanism that corrects misidentified constants and variables in extracted templates.
- We design a template refining mechanism that incorporates historical templates to improve parsing scope.
- Experiments on benchmark datasets show LogReflex surpasses state-of-the-art methods in accuracy and efficiency.

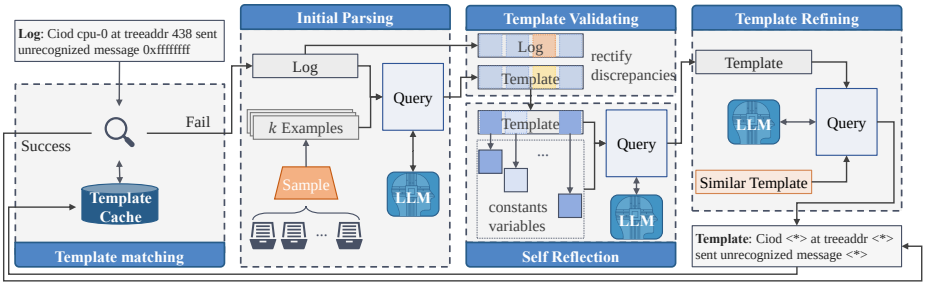

Fig. 1. The Overview of LogReflex

2 Approach

2.1 Preliminary and Overview

A **log sequence** consists of log messages $\mathcal{D} = \{e_1, e_2, \ldots, e_n\}$, where each e_i is a log message. **Constants** are fixed keywords in log print statements, while **variables** are dynamic values or parameters. The **log template** of a message e, denoted as $T(e)$, retains constants while replacing variables with wildcards and is identified by a unique template ID. **Log parsing** converts a log sequence $\mathcal{D}$ into corresponding templates $\{T(e_1), T(e_2), \ldots, T(e_n)\}$ in a streaming fashion, where multiple logs may share the same template, differing only in variable values.

As shown in Fig. 1, **LogReflex** comprises five components: *template matching, initial parsing, template validating, self-reflection,* and *template refining*, enabling a streaming log parsing process. Extracted templates are stored in $\mathcal{T} = \{T_1, T_2, \ldots\}$, where each template corresponds to multiple logs. For a given log e, **template matching** efficiently assigns an existing template from $\mathcal{T}$ using a prefix tree. If no match is found, **initial parsing** generates an initial template $T^0(e)$ using an LLM. Due to potential parsing errors (e.g., missing characters, incorrect capitalization), **template validation** refines $T^0(e)$ by aligning it with e, producing $T^1(e)$. Since LLMs may misclassify constants and variables [10], **self-reflection** reviews and corrects $T^1(e)$, yielding an improved template $T^2(e)$. Finally, **template refining** enhances $T^2(e)$ by incorporating historical templates from $\mathcal{T}$, broadening the LLM's query scope and ensuring better parsing accuracy. The final template $T(e)$ is then assigned.

2.2 Template Matching

Efficient online log parsing is crucial for real-world applications. To enhance efficiency, motivated by existing approaches [6] LogReflex employs a template matching mechanism that quickly matches logs with previously extracted templates in $\mathcal{T}$. For a log e, if it matches a template T in $\mathcal{T}$, we directly assign $T(e) = T$, skipping further steps. Since log templates are far fewer than logs [4], this approach minimizes redundant parsing. Leveraging a prefix tree [2] as a template cache, LogReflex indexes all templates in $\mathcal{T}$, matching logs whose token

sequences align with a path from root to leaf. This mechanism ensures efficient, accurate parsing while reducing reliance on the LLM.

2.3 Initial Parsing

If no matching template is found, a new template must be generated for e. We first perform **initial parsing** using an LLM to extract $T^0(e)$. While various strategies like fine-tuning or In-Context Learning (ICL) are applicable, we adopt the ICL-based approach from state-of-the-art methods [6, 13]. Following LogPPT [8], we construct a reference set RS of log-template pairs by iteratively selecting diverse samples. For a new log e, we identify its K-Nearest Neighbors (KNN) from RS to form an example set $ES = \{p_1, p_2, \ldots, p_k\}$, where each $p_i = (e_i, T(e_i))$. This set is used to construct an LLM prompt for generating $T^0(e)$. To ensure clean output, logs and templates are enclosed within <START> and <END> tags [13].

Algorithm 1: Template Validation

Input: log e, initial template $T^0(e)$
Output: validated template $T^1(e)$

Find the different parts between e and $T^0(e)$ based on LCS algorithm, denoted as a triplet list TL.
$T^1(e) = T^0(e)$
for *(str_e^i, str_t^i, pos_i) in TL* **do**
    if *<*> in str_t^i* **then** `// Case 1`
        Replace str_t with <*> in $T^1(e)$ at pos_i.
    else `// Case 2`
        Replace str_t with str_e in $T^1(e)$ at pos_i.
return $T^1(e)$

2.4 Template Validating

Since we utilize the LLM for initial parsing, discrepancies between $T^0(e)$ and e may occur due to the generative nature of the model. Other factors, such as complex log structures, contextual nuances, and potential noise and variability within the logs, can also contribute to these discrepancies. Therefore, we propose a rectification mechanism to address these discrepancies, called template validating. The core idea is to identify the differences between $T^0(e)$ and e and rectify the erroneous ones.

Specifically, as demonstrated in Algorithm 1, given a log e and its initial template $T^0(e)$, we first find all the different parts between e and $T^0(e)$ based on Longest Common Sub-sequence algorithm (LCS). Then we record them in a triplet list, denoted as $TL = \{(str_e^1, str_t^1, pos_1), (str_e^2, str_t^2, pos_2), \ldots\}$, where

str_e^i is a substring of e, str_t^i is the corresponding substring of $T^0(e)$, and pos_i is their location (line 1). Next, we initialize the content of the validated template $T^1(e)$ to $T^0(e)$ (line 2). For each triplet (str_e^i, str_t^i, pos_i) in TL (line 3), we rectify $T^1(e)$ according to e based on the following Cases:

- Case 1: The LLM identifies str_e^i as a variable, but it may incorrectly generate additional characters not present in e (line 4-5).
- Case 2: The LLM recognizes str_e^i as a constant, but it incorrectly output the content of str_e^i (line 6-7).

It is worth noting that for contents where no discrepancy occurs, the str_t^i is exactly a wildcard. In such cases, the processed template according to Case 1 remains the same as the template before rectifying.

2.5 Self Reflection

Given the diversity and semantic complexity of logs, the model, despite its strong contextual understanding capabilities, may still misclassify some constants as variables and vice versa during initial parsing. Additionally, the general LLM lacks a deep understanding of log parsing [13]. Moreover, similar to humans, LLMs do not always produce the optimal output on their first attempt [10]. During the initial parsing, we rely solely on a few-shot approach for log parsing, which may still result in misidentification of variables and constants. To address this, given a validated template $T^1(e)$ with its raw log e, we further instruct the LLM to reflect on the accuracy of identified constants and variables based on several principles. For a given template, we instruct the LLM to reflect on each constant and variable. Specifically, we first summarize specific characteristics of variables and constants within logs, as follows.

- Variables are typically nouns dynamically populated in logs, representing specific objects, entities, or data values that may change as more system activities are logged, rather than prepositions, conjunctions, or similar.
- Constants typically carry important context and define the interpretation or contextualization, whose value ranges remain unchanged as more system activities are logged.

Based on these characteristics, we prompt the model to analyze both constants and variables with finer granularity in $T^1(e)$. We precisely extract the output by limiting the LLM to reply 'Yes' or 'No'.

To reduce time overhead, we design a pruning strategy to avoid excessive LLM-queries. In general, content within a log composed of pure letters is more likely to be a constant, whereas content composed of pure digits is more likely to be a variable. Therefore, we conduct self reflection only for the following three cases. (1) Variables consist of pure letters. (2) Constants consist of pure digits. (3) Parts of constants that match common variable formats (e.g., IP, path). In detail, given $T^1(e)$ and e, we first apply the pruning strategy to remove unnecessary contents, leaving $\{str_1, str_2, \ldots\}$. For each str_i, we query the LLM with e to determine if it is a constant or a variable, then adjust $T^1(e)$ to $T^2(e)$.

For a given log e, relying solely on the insight of this single log, LogReflex breaks the task of template extraction into two stages. The first stage, initial parsing, uses a few-shot approach to enable the LLM to learn and parse the log automatically. In the second stage, we provide the LLM with expert knowledge to reflect on the results of the first stage. This process of parsing logs from coarse (log level in initial parsing) to fine (content level in self reflection) granularity progressively enhances identification accuracy.

Admittedly, in the ICL-based initial parsing process, expert knowledge could be directly embedded into the initial prompt. However, we chose not to take this approach for two reasons. First, we aim to decouple the initial parsing stage from subsequent steps, ensuring greater flexibility. Second, existing research [12] suggests that breaking down a complex task into smaller sub-tasks can significantly improve overall effectiveness. Consequently, we adopted a multi-stage approach to balance flexibility and performance.

2.6 Template Refining

Up to now, LogReflex processes logs on its own, without any awareness of previous logs and templates. Therefore, $T^2(e)$ may exhibit inconsistencies with templates in $\mathcal{T}$. For example, given $T^2(e)$ "service $<*>$ has been closed by user *admin* from $<*>$", it is ambiguous whether "admin" is a constant or a variable based solely on itself. However, if we find another template "service $<*>$ has been closed by user *test* from $<*>$" in $\mathcal{T}$, we can confirm that "admin" and "test" are the same type of variable and obtain a refined template "service $<*>$ has been closed by user $<*>$ from $<*>$".

The existing LLM-based approach [6] addresses such inconsistencies through a post-processing mechanism: after parsing the current log, it employs longest common subsequence (LCS) similarity to determine whether two templates should be merged when updating the cache. However, this approach has two key limitations. First, it does not provide the LLM with a broader context during the parsing of the current log, as the LLM's perspective remains limited to a single log at a time. Second, the post-processing step relies on a traditional similarity threshold-based approach, which is at odds with the semantic parsing capabilities of LLMs, thereby reducing semantic precision.

Considering this, we propose a template refining process to address the inconsistencies. First, we identify the template T_{sim} in $\mathcal{T}$ that is most similar to the $T^2(e)$. Then, we instruct the LLM with specific principles to identify inconsistencies between $T^2(e)$ and T_{sim}. These inconsistencies could result in two logs, which should share the same template, being extracted different templates due to incomplete information resulting from a narrow insight. When such inconsistencies occur and two templates should be merged, the two templates typically share a similar format and context. Considering this, we instruct the LLM to determine if $T^2(e)$ and T_{sim} should be merged into a single template. If the LLM deems it necessary, we merge $T^2(e)$ with T_{sim} to generate $T(e)$. If not, we consider $T^2(e)$ as a new template. We also limit the LLM to reply 'Yes' or 'No'.

In detail, for the template $T^2(e)$, we find its most similar template T_{sim} in $\mathcal{T}$. Their similarity can be calculated by the Jaccard similarity $sim_{Jaccard}(T, T') = \frac{|bow(T) \bigcap bow(T')|}{|bow(T) \bigcup bow(T')|}$, where $bow(T)$ means the bag of words within template T. Then, we use the aforementioned prompt to let the LLM to determine whether to merge $T^2(e)$ with T_{sim}. If the model consider it necessary, we merge the $T^2(e)$ and T_{sim} to generate $T(e)$. Then we update T_{sim} with $T(e)$ both in template cache and $\mathcal{T}$. Conversely, if not, $T(e)$ is exactly the same as $T^2(e)$. We insert $T(e)$ into $\mathcal{T}$ as a new template, and insert it into the template cache. This process allows us to provide a broader perspective for the already extracted templates and continuously refine them based on the current log.

3 Experiment

3.1 Experiment Setup

In this section, we introduce the experiment setup.

Baselines: We compare LogReflex with **Drain** [3], a traditional search tree-based approach, **LogPPT** [8], a deep-learning approach, and **DivLog** [13] and **LILAC** [6], both LLM-based with in-context learning, with LILAC incorporating a parsing cache. All implementations are publicly available [6, 8, 14].

Datasets: We use benchmark log datasets from Loghub [4] that include 16 labeled benchmark datasets (2000 logs). Most existing log parsing approaches use these 16 labeled datasets. Meanwhile, some studies [7,9] point out that there are some errors and fix them. Additionally, to conduct efficiency experiments, we selected four complete datasets of varying sizes for efficiency experiments.

Metric: We evaluate effectiveness using prior survey metrics [7, 14]. **Group Accuracy (GA)** measures correctly grouped logs, $GA = N_c/N$. **Parsing Accuracy (PA)** assesses correct variable and constant identification, $PA = N'_c/N$. **Precision Template Accuracy (PTA)** is $PTA = N'_{Tc}/N_{Tp}$, where templates are correct if they satisfy GA and PA. **Recall Template Accuracy (RTA)** is $RTA = N'_{Tc}/N_{Tg}$, with N_{Tg} as ground truth templates.

Environment: We locally deploy the open-source LLM Qwen2-72B [1]. The model runs on an Ubuntu server with 4 NVIDIA A40 (48GB) GPUs and an Intel Xeon Platinum 8358 CPU @ 2.60GHz. Since LogPPT requires a GPU, we run its experiments on the same server. Other experiments are conducted on a Windows machine with an AMD Ryzen 7 4800U CPU @ 1.80GHz, 16GB RAM, connected to the Ubuntu server over a network for LLM invocation.

Implementation: We use logs and ground truth templates from 16 public benchmark datasets, randomly splitting them into two groups of 8 to construct the reference set RS. Evaluation ensures no data leakage by sampling RS from the other group, following the same setup for LogPPT, DivLog, and LILAC. This setup reflects real-world challenges, where collecting annotated RS is labor-intensive, and unseen logs frequently appear. For fair comparison, we use parameter values recommended in prior works. In LogReflex, we set RS to 32 logs, the example set ES to 3 logs, and the LLM temperature to 0 to eliminate

Table 1. Accuracy Comparison on log level metric

Dataset	Drain		LogPPT		DivLog		LILAC		LogReflex	
	GA	PA	GA	PA	GA	PA	GA	PA	GA	PA
HDFS	99.8	99.7	41.2	84.3	5.9	86.2	**100.0**	85.4	**100.0**	**100.0**
Hadoop	89.0	52.1	54.9	74.3	48.7	89.3	93.3	84.4	**97.8**	**93.3**
Spark	92.0	56.2	43.3	91.9	35.9	82.8	**98.1**	**94.0**	92.2	92.1
Zookeeper	96.7	95.8	75.6	92.3	48.4	97.4	98.7	66.3	**99.5**	**99.3**
OpenStack	73.3	1.9	39.6	**81.5**	38.0	34.3	48.1	45.0	**100.0**	**49.2**
BGL	96.3	78.9	40.9	84.0	33.0	90.1	93.9	89.4	**95.5**	**93.1**
HPC	88.7	65.4	74.3	79.0	48.2	69.8	87.7	87.2	**90.6**	**89.7**
Thunderbird	95.5	33.8	65.0	79.3	64.6	86.9	95.1	**88.8**	**96.8**	82.8
Windows	99.7	46.2	71.3	75.2	40.2	52.3	69.2	55.4	**99.5**	**86.1**
Linux	87.6	18.4	17.7	70.0	29.4	83.6	75.1	67.5	**93.8**	**92.9**
Mac	78.7	29.6	76.3	50.7	55.8	59.7	86.6	51.5	**86.7**	**60.7**
Android	83.1	55.0	87.7	78.0	72.7	59.3	**95.4**	58.7	95.0	**87.8**
HealthApp	78.0	35.4	**100.0**	**89.1**	63.2	88.7	90.1	88.1	90.3	88.7
Apache	100.0	69.4	58.2	30.7	56.0	69.6	**100.0**	**98.4**	**100.0**	71.6
Proxifier	52.7	50.4	0.1	38.0	0.0	56.6	52.5	52.5	**100.0**	**100.0**
OpenSSH	78.9	50.8	33.2	44.8	17.6	50.7	68.7	61.9	**80.1**	**80.1**
Average	86.9	52.4	54.9	71.4	41.1	72.3	84.5	73.4	**94.8**	**85.4**

randomness. All experiments use Python 3.10. The LogReflex code is available at `https://github.com/Chen-XiaoLei/LogReflex`.

3.2 Effectiveness Comparison

In this experiment, we evaluate the effectiveness of LogReflex, as measured by *GA*, *PA*, *PTA*, and *RTA* compared with other approaches, using 16 benchmark datasets. The results are presented in Table 1 and Table 2. The highest metric values are highlighted in bold. LogReflex achieves the highest values on 12 out of 16 datasets for *GA* metric, 12 for *PA*, 14 for *PTA*, and 14 for *RTA*. Additionally, LogReflex achieves the highest average values across all metrics.

The *GA* metric assesses parser consistency in grouping logs [6]. Table 1 shows that Drain, LILAC, and LogReflex perform well, with Drain benefiting from statistical clustering, while LILAC and LogReflex leverage a prefix tree for matching. LogReflex further improves consistency through template refining. In contrast, LogPPT and DivLog lack consistency as their grouping relies on individual templates, where one differing template can affect an entire group. The *PA* metric measures accuracy in identifying constants and variables. Model-based approaches (LogPPT, DivLog, LILAC, LogReflex) perform better, while Drain, relying on statistical features, scores lower. LogReflex outperforms others by incorporating self-reflection and template refining, enhancing initial templates and correcting errors. DivLog lacks corrections, and LILAC's reliance on heuristic templates reduces semantic accuracy. The *PTA* and *RTA* metrics evaluate template-level effectiveness. Drain's poor variable identification and LogPPT and DivLog's consistency issues result in lower scores. LogReflex achieves higher values due to its accuracy and consistency.

Table 2. Accuracy Comparison on template level metric

Dataset	Drain		LogPPT		DivLog		LILAC		LogReflex	
	PTA	RTA	PTA	RTA	PTA	RTA	PTA	RTA	PTA	RTA
HDFS	75.0	85.7	18.5	35.7	5.4	21.4	92.9	92.9	**100.0**	**100.0**
Hadoop	23.9	28.9	51.3	68.4	53.1	60.5	68.3	60.5	**74.5**	**69.3**
Spark	48.3	38.9	38.0	52.8	26.3	55.6	72.2	**72.2**	**75.8**	69.4
Zookeeper	60.9	56.0	51.6	66.0	24.8	52.0	68.6	70.0	**80.7**	**92.0**
OpenStack	1.0	7.0	31.1	65.1	22.7	46.5	33.3	83.7	**93.0**	**93.0**
BGL	30.6	27.5	34.1	46.7	32.4	55.8	72.3	71.7	**88.7**	**85.0**
HPC	38.3	39.1	38.0	58.7	35.1	56.5	**79.1**	73.9	77.8	**76.1**
Thunderbird	24.2	29.5	42.4	59.7	43.2	61.7	55.8	67.8	**59.9**	**71.1**
Windows	39.6	42.0	53.0	70.0	35.2	62.0	50.0	56.0	**63.0**	**68.0**
Linux	44.6	43.1	68.6	82.8	65.9	76.7	69.8	69.8	**84.2**	**82.8**
Mac	24.4	28.2	40.5	49.0	25.6	38.4	53.1	50.7	**64.3**	**65.1**
Android	53.2	57.6	50.3	62.0	41.9	55.7	70.1	68.4	**75.7**	**72.8**
HealthApp	7.7	32.0	81.3	81.3	50.0	62.7	76.4	73.3	**86.3**	**84.0**
Apache	50.0	50.0	22.2	33.3	11.8	33.3	**83.3**	**83.3**	**83.3**	**83.3**
Proxifier	27.8	62.5	0.0	0.0	0.0	0.0	20.7	75.0	**100.0**	**100.0**
OpenSSH	45.8	42.3	5.1	38.5	11.9	26.9	**76.0**	**73.1**	65.5	**73.1**
Average	37.2	41.9	39.1	54.4	30.3	47.9	65.1	71.4	**79.5**	**80.3**

3.3 validity of each component

In this experiment, we conduct an ablation study of LogReflex to assess the impact of its four core components: self reflection, template refining, template validating, and example sampling. We conduct five experimental variants on the benchmark datasets. By analyzing these variants, we aim to evaluate the individual contributions of each component to the overall performance of LogReflex.

Table 3. Ablation experiment

	GA	PA	PTA	RTA
LogReflex	94.8	85.4	79.5	80.3
LogReflex w/o self reflection	88.9	77.1	67.7	68.3
LogReflex w/o template refining	91.9	82.4	74.5	80.0
LogReflex w/o template validating	85.7	76.0	75.7	73.1
LogReflex w/o all	79.1	70.8	67.8	69.8

As shown in Table 3, the ablation study demonstrates the critical importance of each core component in the performance of LogReflex. Removing any one of the components, self reflection, template refining, template validation, or example sampling, leads to a decline in the effectiveness of the parser. This indicates that these components work synergistically to enhance the robustness and precision of LogReflex. Particularly, the complete absence of all components results in the most pronounced performance drop, underscoring their collective contribution to the overall performance of the parser.

4 Conclusion

In this paper, we introduced LogReflex, an online log parsing framework addressing misidentification and narrow-scope issues in LLM-based approaches. The self-reflection mechanism corrects misclassified constants and variables, while the template refining mechanism expands parsing context using historical templates. These improvements enhance accuracy and consistency. Experiments on benchmark datasets show that LogReflex achieves state-of-the-art accuracy.

References

1. Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., et al.: Qwen technical report. arXiv preprint arXiv:2309.16609 (2023)
2. Chen, X., Shi, J., Chen, J., Wang, P., Wang, W.: High-precision online log parsing with large language models. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (2024)
3. He, P., Zhu, J., Zheng, Z., Lyu, M.R.: Drain: An online log parsing approach with fixed depth tree. In: 2017 IEEE international conference on web services (2017)
4. He, S., Zhu, J., He, P., Lyu, M.R.: Loghub: a large collection of system log datasets towards automated log analytics. arXiv preprint arXiv:2008.06448 (2020)
5. Huo, Y., Su, Y., Lee, C., Lyu, M.R.: Semparser: A semantic parser for log analytics. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 881–893. IEEE (2023)
6. Jiang, Z., Liu, J., Chen, Z., Li, Y., Huang, J., Huo, Y., He, P., Gu, J., Lyu, M.R.: Lilac: Log parsing using llms with adaptive parsing cache. Proceedings of the ACM on Software Engineering **1**(FSE), 137–160 (2024)
7. Khan, Z.A., Shin, D., Bianculli, D., Briand, L.: Guidelines for assessing the accuracy of log message template identification techniques. In: Proceedings of the 44th International Conference on Software Engineering (2022)
8. Le, V.H., Zhang, H.: Log parsing with prompt-based few-shot learning. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (2023)
9. Liu, Y., Zhang, X., He, S., Zhang, H., Li, L., Kang, Y., Xu, Y., Ma, M., Lin, Q., Dang, Y., et al.: Uniparser: A unified log parser for heterogeneous log data. In: Proceedings of the ACM Web Conference 2022. pp. 1893–1901 (2022)
10. Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al.: Self-refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems **36** (2024)
11. Makanju, A.A., Zincir-Heywood, A.N., Milios, E.E.: Clustering event logs using iterative partitioning. In: Proceedings of the 15th international conference on Knowledge discovery and data mining (2009)
12. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D., et al.: Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems **35**, 24824–24837 (2022)
13. Xu, J., Yang, R., Huo, Y., Zhang, C., He, P.: Divlog: Log parsing with prompt enhanced in-context learning. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–12 (2024)
14. Zhu Jieming, He Shilin, L.J.H.P.X.Q.Z.Z., R, L.M.: Tools and benchmarks for automated log parsing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (2019)