Improving LLM-based Log Parsing by Learning from Errors in Reasoning Traces

Jialai Wang
National University of Singapore
Singapore
wangjialai97@gmail.com

Juncheng Lu
Southeast University
Nanjing, China
213221523@seu.edu.cn

Jie Yang
Wuhan University
Wuhan, China
2022302181128@whu.edu.cn

Junjie Wang
Wuhan University
Wuhan, China
sanowip@gmail.com

Zeyu Gao
Tsinghua University
Beijing, China
gaozy22@mails.tsinghua.edu.cn

Chao Zhang[†]
Tsinghua University
Beijing, China
chaoz@tsinghua.edu.cn

Zhenkai Liang
National University of Singapore
Singapore
liangzk@comp.nus.edu.sg

Ee-Chien Chang
National University of Singapore
Singapore
changec@comp.nus.edu.sg

***Abstract*—Recent advances in reasoning-capable large language models (LLMs) have led to their application in a wide range of tasks, including log parsing. These LLMs generate intermediate reasoning traces during inference, offering a unique opportunity to analyze and improve their performance. In this work, we investigate how reasoning traces can be leveraged to enhance LLM-based log parsers. We propose `TraceDoctor`, a framework that analyzes reasoning traces associated with parsing errors to understand the causes of failure. We categorize these error causes into high-level error types and design targeted log variant generation strategies guided by these high-level error types. The generated variants are then used to fine-tune the LLMs. We instantiate five state-of-the-art (SOTA) reasoning-capable LLMs as log parsers and identify 29 distinct high-level error types. Our approach improves their average parsing accuracy by up to 17.3% and 16.3% on parsing accuracy (PA) and group accuracy (GA), respectively.**

I. Introduction

Large language models (LLMs) with reasoning capabilities have recently demonstrated strong performance across a wide range of downstream tasks, including programming tasks [55], translation tasks [4], [41], and mathematical reasoning [13], [26], [47]. Unlike traditional LLMs that produce answers directly, reasoning LLMs generate intermediate reasoning traces that reflect their step-by-step thought process in natural language. These traces provide insights into how models interpret inputs, decomposes problems, and arrives at final answers. The transparency offered by reasoning traces has led to growing interest in using them not only to explain model decisions, but also to diagnose and improve model behavior [2], [27], [35], [40], [59].

In this work, we focus on enhancing the performance of reasoning LLMs in the context of log parsing. Log parsing converts raw log messages into structured data formats (e.g., templates and parameters), and is a critical first step for many downstream log analysis tasks, such as anomaly detection [18], [58], root cause analysis [1], [16], [17], and system monitoring [5], [48]. Traditional log parsers rely on manually crafted rules or statistical heuristics, which are often brittle and tightly coupled to specific log formats. As a result, they struggle to generalize to unseen log types or evolving systems. Recently, LLMs have emerged as a promising alternative for log parsing [3], [7], [20], [22], [24], [29], [31], [37], [50], [51], [53], [54], [56], [57], [61], [62]. These models, when equipped with reasoning abilities, can handle complex, varied log formats and provide interpretable reasoning traces that expose their parsing logic. However, little is known about how to systematically analyze and improve LLM-based parsers by leveraging these traces.

To fill this gap, we propose a framework `TraceDoctor` that analyzes the reasoning traces of failed log parsing cases to understand why errors occur, and uses this understanding to generate targeted log variants for enhancing LLM-based parsers. Our core insight is inspired by how teachers help students improve: by examining a student's thought process, identifying where and why it went wrong, and providing targeted examples to correct those misunderstandings. Similarly, reasoning traces reveal the thinking process of an LLM during log parsing. By analyzing the traces associated with incorrect outputs, we identify the causes of parsing errors, and generate corresponding log variants to fine-tune the model effectively.

Implementing the above framework presents two main challenges. The first challenge lies in automatically analyzing reasoning traces to identify parsing error causes. Reasoning traces reflect the model's internal thought process but do not explicitly indicate why the reasoning went wrong. Moreover, each parsing error corresponds to a unique trace, and even when the underlying causes are similar, the surface form of these traces can differ significantly. This diversity makes it difficult to generalize across traces. As the number of parsing errors increases, it becomes infeasible for humans to manually inspect and summarize error causes from the reasoning traces. Automating this complex and noisy analysis process is thus a key challenge.

The second challenge is that even after identifying the cause of a parsing error from reasoning traces, it remains unclear

[†] Corresponding author.

how to generate effective log variants based on the analysis. A single error type can be triggered by many different log inputs that vary in structure and wording. In other words, there exist multiple log variants that can reproduce the same error. However, not all of them are equally useful for training. Some variants may reproduce the failure but offer little learning signal due to their low impact on the model's internal representations. Others may be too similar to previously seen examples and fail to encourage generalization. Currently, no existing approach provides a valid method for generating log variants tailored to the log parsing domain.

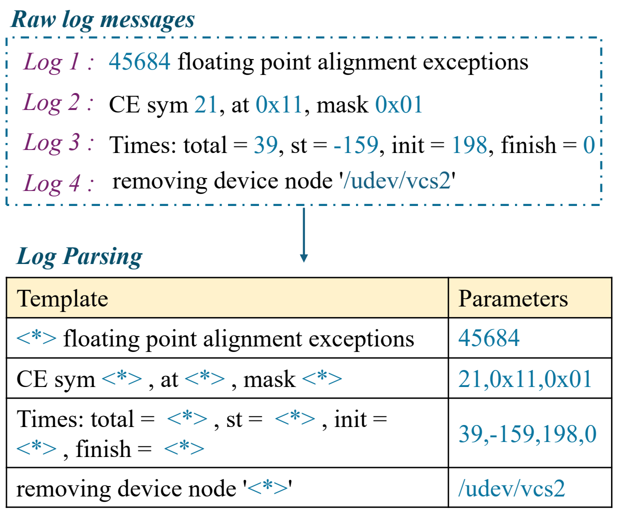
To address the first challenge, we design an automated analysis approach based on LLMs. We use an LLM to analyze the reasoning trace associated with each parsing error and generate a natural language explanation that describes why the error occurs. Although these explanations differ in phrasing, many of them reflect similar underlying problems. To capture these common patterns, we categorize the explanations into a set of high-level error types. This categorization enables a structured understanding of parsing errors and serves as the foundation for subsequent log variant generation.

To address the second challenge, we focus on enhancing the diversity of log variants. For each identified high-level error type, we generate multiple log variants that differ in structure and semantics but preserve the error type. To this end, we design three generation strategies that consider the structural and semantic variation of log fields and support different levels of modification. The first strategy performs variable substitution by modifying lightweight fields such as numeric values or identifiers while preserving the log's meaning. The second strategy applies constant-inclusive rewriting, allowing broader changes to constants while maintaining structural and semantic consistency. The third strategy performs free-form generation by creating entirely new logs guided by the identified error type. This can generate variants without existing erroneous logs, allowing for broader semantic and structure variation.

We implement our framework, `TraceDoctor`, and apply it to improve the log parsing performance of five state-of-the-art (SOTA) reasoning-capable LLMs. Experimental results show that `TraceDoctor` increases PA and GA by 17.3% and 16.3% on average, respectively, and consistently outperforms all baseline methods.

In summary, this work makes the following contributions.

- We propose a framework, `TraceDoctor`, that analyzes the reasoning traces associated with log parsing errors and generates targeted log variants accordingly. `TraceDoctor` significantly improves model performance and outperforms all baselines.
- We identify 29 distinct high-level error types that have not been explored in prior work, potentially inspiring future improvements in model design and training.
- We validate that identifying error types from reasoning traces and using them to guide log variant generation can effectively improve model performance, introducing a new direction for enhancing log parsers.



```
Log 1 :  45684 floating point alignment exceptions
Log 2 :  CE sym 21, at 0x11, mask 0x01
Log 3 :  Times: total = 39, st = -159, init = 198, finish = 0
Log 4 :  removing device node '/udev/vcs2'
```

Template	Parameters
<*> floating point alignment exceptions	45684
CE sym <*> , at <*> , mask <*>	21,0x11,0x01
Times: total = <*> , st = <*> , init = <*> , finish = <*>	39,-159,198,0
removing device node '<*>'	/udev/vcs2

Fig. 1. A simplified example of log parsing.

II. Background and Related Work

A. Log Parsing Definition

Log parsing converts logs into a structured format, typically a template plus parameters. The template corresponds to log constants, and the parameters are log variables (*e.g.*, timestamps and file paths). For example (Figure 1), given the log message `CE sym 21, at 0x11, mask 0x01`, a parser extracts the corresponding template `CE sym ⟨∗⟩, at ⟨∗⟩, mask ⟨∗⟩` and the parameter values `21`, `0x11`, and `0x01`. By structuring logs in this way, log parsing enables developers to apply analytical tools downstream, and log parsing is typically the first step in automated log analysis pipelines, enabling critical tasks like anomaly detection [18], [58], root cause analysis [1], [16], [17], and system monitoring [5], [48].

B. LLM-based Parsing

Recently, many LLM-based solutions have been proposed and shown promising parsing performance compared to traditional parsers. We categorize LLM-based solutions into non-fine-tuning solutions and fine-tuning-based solutions.

Non-fine-tuning solutions. Many non-fine-tuning methods [3], [22], [24], [29], [53], [54] adopt in-context learning, where a small number of log–template pairs are embedded in prompts as demonstrations. For example, DivLog [54] selects five examples from a candidate pool by maximizing structural diversity. It measures token-level differences between log messages to select demonstrations with varying formats and variable structures. By exposing models to diverse log patterns, DivLog improves model performance.

To improve demonstration quality, some methods [22], [37], [50] incorporate retrieval mechanisms into the in-context learning pipeline. When given a log to parse, these methods dynamically retrieve similar historical logs or log–template pairs based on structural similarity, allowing the model to condition on structurally aligned examples. Retrieved candidates are often organized into clusters, trees, or buckets to facilitate efficient selection. For instance, LILAC [22] maintains a cache of previously parsed templates and selects relevant examples to construct prompts. To reduce redundant LLM invocations, some solutions [22], [50], [61], [62] cache templates in structured formats such as prefix trees or similarity-based clusters. When a new log arrives, these systems first attempt to match it against cached templates and invoke LLMs only when no match is found. For example, LogParser-LLM [62] builds a syntax-aware prefix parse tree to organize log clusters and resorts to LLM inference only when both strict and loose matching fail.

In contrast to the aforementioned methods that rely on log–template pairs as demonstrations, some work [20], [31], [51], [56] adopt demonstration-free prompting strategies. These approaches eliminate the need for labeled templates. They exploit the inherent structure and regularity of logs, such as shared prefixes or repeated patterns, to infer templates without any labeled examples. LUNAR [20], for instance, clusters logs into groups with similar structures and constructs prompts from each group to guide a frozen LLM. LogBatcher [51] also feeds batches of similar logs into the LLM, enabling it to learn common structure across inputs without explicit guidance. Other solutions [31], [56] incorporate refinement mechanisms like self-reflection and template caching to improve parsing accuracy and reduce redundant computation. These approaches eliminate the need for labeled templates while retaining high scalability and efficiency.

Fine-tuning-based solutions. To tailor pretrained language models to log parsing tasks, several studies fine-tune LLMs on task-specific data [21], [25], [30], [32]. LLMParser [30] formulates parsing as a sequence-to-sequence task and fine-tunes a decoder-only model to map raw logs directly to structured templates. This approach captures format-specific patterns through supervised learning. Mehrabi et al. [32] further show that even compact models, when fine-tuned effectively, can deliver strong performance with lower inference cost. Instead of generating full templates end-to-end, LogPPT [25] introduces a prompt-programming framework that decomposes log parsing into interpretable subtasks, such as identifying variable spans and reconstructing templates. LogLM [28] integrates multiple log analysis tasks such as anomaly detection and log parsing, into a unified instruction-based training framework. SuperLog [21] takes a complementary direction by constructing a large-scale Q&A dataset, where each question corresponds to a real-world log and each answer provides a human-readable explanation. Training on this dataset enables models to acquire domain knowledge useful not only for parsing, but also for downstream tasks such as anomaly detection and root cause analysis.

C. Divergence from Related Work

Divergence from existing log parsing solutions. While our method follows the fine-tuning paradigm, it fundamentally differs from existing fine-tuning-based log parsing approaches. Prior work does not incorporate reasoning traces into error analysis or generate training data based on parsing errors. In contrast, `TraceDoctor` analyzes reasoning traces to identify the causes of parsing errors and generates log variants accordingly. Our approach is also orthogonal to non-fine-tuning methods, which rely on frozen LLMs without updating model parameters. Although conceptually distinct, the two paradigms are complementary. For example, fine-tuned models produced by `TraceDoctor` can be integrated into non-fine-tuning pipelines to improve their parsing quality. We validate this in our experiments (Section IV-D), showing that `TraceDoctor` enhances the performance of state-of-the-art non-fine-tuning methods.

Divergence from data augmentation and adversarial training. In other domains, *e.g.*, computer vision [10], [44], data augmentation [36], [42], [46], [49], [52], [63] and adversarial training [11], [12], [23], [43], [45], [64] have been used to generate data for fine-tuning. However, these approaches differ fundamentally from ours. Neither data augmentation nor adversarial training leverages reasoning trace analysis to guide the generation of training data. In our approach, by contrast, data generation is explicitly driven by analyzing reasoning traces. For example, data augmentation techniques typically apply random perturbations to inputs, without analyzing model errors or internal reasoning processes.

Divergence from reasoning error analysis in other domains. Recent studies [6], [27], [35] have investigated the analysis of reasoning trace errors to improve model performance. However, these solutions primarily target mathematical tasks, which differ fundamentally from log parsing. In mathematical tasks, the reasoning trace typically consists of multi-step logical derivations, and existing methods rely on identifying faulty steps and repairing them incrementally to reconstruct a correct reasoning path. In contrast, log parsing is not a deductive task and does not require such fine-grained logical decomposition. Our approach avoids step-by-step reasoning trace repair entirely. Instead, we summarize high-level error types and use them to guide the generation of diverse log variants. These variants serve as effective training examples without the need to decompose or explicitly fix the original reasoning traces.

III. Methodology

A. Overview

`TraceDoctor` consists of two modules: `Analyzer` and `Generator`. The `Analyzer` module analyzes the reasoning traces generated by the model for failed parsing cases and summarizes the cause of each parsing error. It then categorize diverse error explanations into a set of high-level error types based on their underlying semantics, providing a basis for generating log variants. Building on the high-level error types identified by `Analyzer`, the `Generator` module generates diverse log variants through three complementary strategies. Each strategy modifies logs at a different structural or semantic level, capturing varied expressions of each error type. These variants are then used to fine-tune models. We show the whole workflow in Figure 2.

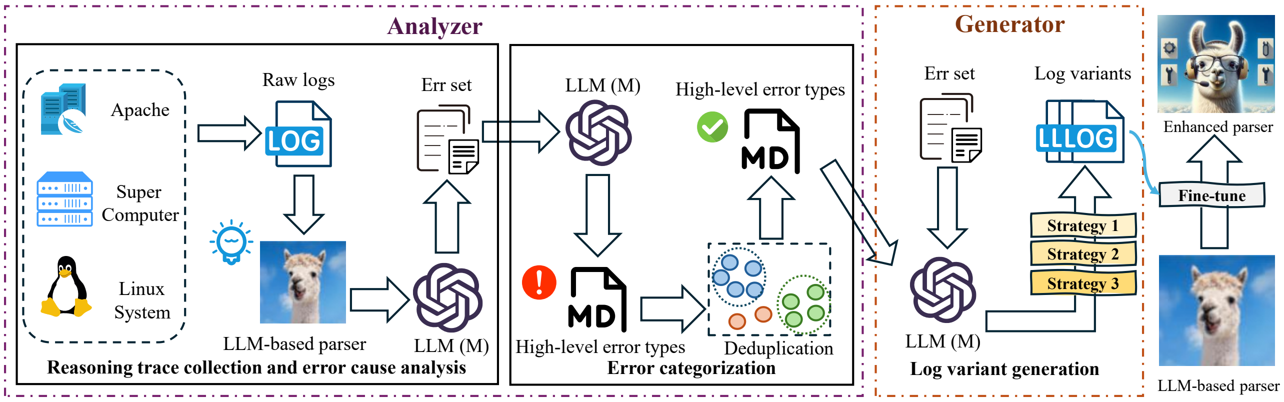


Fig. 2. The general workflow of `TraceDoctor`.

B. Analyzer

We design `Analyzer` to analyze parsing errors by identifying their causes from the associated reasoning traces. For each parsing error, `Analyzer` prompts an LLM to generate a natural language explanation of why the error occurs. After collecting all explanations, `Analyzer` categorizes them into a set of high-level error types. This process abstracts diverse error causes into a set of high-level error types, similar to how a teacher identifies common misconceptions among students to design more effective instruction. In addition, this categorization helps mitigate data imbalance during log variant generation. Some error types occur frequently, while others are rare. If we generate log variants for each parsing error individually, the resulting dataset may be skewed toward dominant error types. By grouping errors into categories, we can control the number of variants generated per type, ensuring balanced coverage across error types.

Reasoning trace collection and error cause analysis. `Analyzer` begins by collecting reasoning traces associated with parsing errors. Let f denote an LLM-based parser, and let $\mathcal{D} = \{x_i\}_{i=1}^n$ be a log dataset, where each log $x \in \mathcal{D}$ is paired with a ground-truth template $T(x)$. For each log x, `Analyzer` queries f to obtain both the predicted template $f(x)$ and the corresponding reasoning trace $r(x)$. If the prediction does not match the ground truth, i.e., $f(x) \neq T(x)$, we consider this a parsing error. `Analyzer` then records the tuple $(x, r(x), T(x))$ as an error instance. After iterating over the dataset, `Analyzer` constructs the full set of parsing errors:

$$E = \{(x, r(x), T(x)) \mid f(x) \neq T(x),\ x \in \mathcal{D}\}. \tag{1}$$

Although each reasoning trace $r(x)$ records how the parser arrives at its output, it does not directly indicate why a parsing error occurs. To uncover the cause of a parsing error, we prompt a separate LLM M to analyze $r(x)$ and summarize the cause of the error as reflected in the trace. For each error instance $(x, r(x), T(x)) \in E$, M produces a natural language explanation $\text{err}(x)$ that describes why the parsing error occurs. We then update the error set E to:

$$E = \{(x, \text{err}(x), T(x)) \mid f(x) \neq T(x),\ x \in \mathcal{D}\}, \tag{2}$$

where $\text{err}(x)$ serves as a concise explanation of the parsing error derived from the reasoning trace.

Error categorization. `TraceDoctor` categorizes all error explanations in E into a set of high-level error types. A key challenge is that the number of error types is unknown beforehand. When prompted without a predefined category count, LLMs tend to generate semantically redundant types that differ in phrasing. This redundancy compromises the clarity and utility of the categorization. On the other hand, enforcing a fixed number of categories may lead M to ignore infrequent yet meaningful error types. To address this, we design a two-stage categorization strategy. First, we prompt M to freely propose candidate error types. Then, we perform embedding-based similarity analysis to identify and remove semantically overlapping types, thereby reducing redundancy while preserving distinct error types.

First, we prompt the LLM M with the full set of error reasons $\{\text{err}(x) \mid x \in E\}$, instructing it to categorize them into a set of initial high-level error types, denoted by $H_{\text{init}} = \{h_1, h_2, \dots, h_n\}$. Importantly, we do not specify the number of categories to allow the model to surface rare or unexpected patterns. This generation step yields broad coverage, but often includes overlapping error types.

To reduce redundancy in H_{init}, we aim to select a subset $H \subseteq H_{\text{init}}$ that contains only semantically distinct categories. More specifically, we begin by encoding each type $h_i \in H_{\text{init}}$ into a dense vector representation $v(h_i) \in \mathbb{R}^d$ using a pre-trained sentence embedding model. The semantic similarity between any two types h_i and h_j is then measured using cosine similarity:

$$\text{sim}(h_i, h_j) = \cos(v(h_i), v(h_j)). \tag{3}$$

To enforce distinctiveness, we define a subset $H \subseteq H_{\text{init}}$ such that for any $h_i \in H$, its similarity to every other type already in H does not exceed a predefined threshold $\delta \in (0, 1)$:

$$H = \{h_i \in H_{\text{init}} \mid \forall h_j \in H,\ h_j \neq h_i \Rightarrow \text{sim}(h_i, h_j) \leq \delta\}\,. \tag{4}$$

Here, we set $\delta = 0.9$ in our experiments. The operator $\Rightarrow$ indicates logical implication: for any pair of distinct types, the similarity must fall below the threshold δ, ensuring that each retained type is semantically distinct from others.

Since optimally selecting the largest possible subset under this pairwise constraint is NP-hard, we adopt a frequency-guided greedy approximation. First, we compute the usage frequency of each type in H_{init}. To do this, we prompt the LLM to assign each explanation $\text{err}(x)$ to its most appropriate type $h(x) \in H_{\text{init}}$, and then compute:

$$\text{freq}(h_i) = |\{x \in E \mid h(x) = h_i\}|\,. \tag{5}$$

We then apply a greedy selection algorithm. All candidate types in H_{init} are sorted in descending order based on their frequencies:

$$[h_1, h_2, \ldots, h_n] = \texttt{SortByFreq}(H_{\text{init}}), \tag{6}$$

where $\texttt{SortByFreq}(\cdot)$ sorts by $\text{freq}(h_i)$. The final set H is initialized as empty. We sequentially examine each type h_i in the sorted list, adding it to H only if its similarity with all previously selected types in H is less than or equal to δ:

$$\forall h_j \in H,\ \text{sim}(h_i, h_j) \leq \delta. \tag{7}$$

If this condition is met, the type is added to the set:

$$H \leftarrow H \cup \{h_i\}. \tag{8}$$

Finally, `Analyzer` yields the refined set of high-level error types H, in which all pairwise similarities are below the threshold δ. To offer an intuitive understanding of high-level error types, we present several representative examples.

Examples of high-level error types

Log structure modification: The model alters the log structure based on its own interpretation, mistakenly adding symbols that did not originally exist.

Over-complication of analysis: Introduce unnecessary complexity when parsing simple log patterns

Non-standard placeholder notation: Use custom or specific variable markers instead of the required generic <*> placeholder

Variable over-segmentation: Identifying compound identifiers as multiple variables for replacement, rather than treating them as a whole for a single variable replacement.

C. Generator

Based on the high-level error types identified, we design `Generator` to generate log variants for fine-tuning models. `Generator` leverages M to synthesize variants guided by these error types. For each high-level error type $h_i \in H$, `Generator` randomly selects a tuple $(x, y, \text{err}(x)) \in E$, where the error explanation $\text{err}(x)$ is associated with h_i. This tuple serves as an anchor for generation. Specifically, the original log x provides the base input, the label y offers ground-truth supervision context, and $\text{err}(x)$ illustrates a concrete instance of error cause captured by h_i. These components are encoded into the prompt for M, enabling it to generate log variants that reflect the specific failure mode captured by h_i.

However, it is unclear what specific form of log variants can effectively facilitate model fine-tuning. To address this challenge, we propose three strategies to generate diverse log variants, thus increasing the likelihood of covering effective log variants. During log variant generation, we explicitly include a selected strategy in the prompt. This steers M to perform the corresponding strategy. Our three strategies are as follows. (1) *Variable substitution*. Generating log variants by modifying variables. (2) *Constant-inclusive rewriting*. Generating log variants by modifying constants and variables. (3) *Free-form generation*. Generating log variants without logs.

Variable substitution. This strategy modifies only variables in a log while preserving its overall structure and semantics. Given a log x and its template $T(x)$ from E, $T(x)$ specifies the locations of variables. According to $T(x)$, we extract all variables $V = \{v_1, v_2, \ldots, v_n\}$, where each v_i represents a concrete instance of a variable in x, (*e.g.*, an IP address or timestamp). To determine which variables to modify, we sample a binary mask over the variable set using independent Bernoulli trials. For each variable $v_i \in V$, we sample a binary indicator $z_i \in \{0, 1\}$ from a Bernoulli distribution, $z_i \sim \text{Bernoulli}(p)$, where $p \in [0, 1]$ is a hyperparameter (we set $p = 0.5$) that controls the expected proportion of variables to be replaced. The set of selected variables is then defined as:

$$V' = \{v_i \in V \mid z_i = 1\}. \tag{9}$$

For each selected variable $v_i \in V'$, we prompt an LLM to generate a replacement value v_i'. The replacement conforms to the semantic and syntactic constraints of the original variable. For example, if a variable corresponds to an IP address, the modified variable should still be an IP address. The generated log variant x' is obtained by replacing each $v_i \in V'$ in x with the corresponding v_i', while keeping all other fields unchanged. This strategy introduces controlled variability through lightweight, localized substitutions, enabling targeted exploration of the input space without disrupting the structural or semantic integrity of the log.

Constant-inclusive rewriting. In addition to modifying variables, this strategy supports rewriting constants within logs. This strategy samples several constants according to their semantic importance, and applies an LLM to conduct semantic-preserving rewriting, such as rewording articles or auxiliaries. More specifically, given a log, we first estimate the semantic importance of each constant using its surprisal under an LLM. We then sample a subset of constants for rewriting, assigning lower rewriting probabilities to those deemed more semantically critical. Finally, we invoke an LLM to perform semantic-preserving rewriting for the selected constants, introducing lexical variation while preserving the original semantics of the log.

We estimate the semantic importance of each constant based on its surprisal [14], [38] under an LLM [19]. Specifically, the surprisal of a constant is defined as the negative log-probability of the token given its preceding context, measuring how unexpected it is in that position. Intuitively, highly predictable constants (e.g., articles or prepositions) carry little semantic content, whereas less predictable ones (e.g., specific function names such as `handleRequest`) tend to be more informative and semantically specific. Surprisal thus provides a model-grounded and cognitively motivated estimate of the semantic importance of log constants.

Formally, given a log x and its corresponding template $T(x)$, the template specifies the locations of constants within the log. Based on $T(x)$, we extract all constants $C = \{c_1, c_2, \ldots, c_m\}$, where each $c_j \in C$ denotes a constant in the log. To compute the surprisal of a constant, we leverage an LLM to obtain token-level probabilities. Since LLMs operate on subword tokens rather than raw constants, each constant is first tokenized into one or more subword tokens. To address this granularity mismatch, we define the surprisal of a constant as the average surprisal of its constituent tokens.

Let $\mathbf{t} = (\tau_1, \tau_2, \ldots, \tau_n)$ denote the token sequence produced by tokenizing the entire log x. For each constant c_j, we define $\text{tokens}(c_j) \subseteq \mathbf{t}$ as the subsequence of tokens that constitute c_j. We define the surprisal of a token τ_i as:

$$\phi(\tau_i) = -\log P(\tau_i \mid \tau_1, \ldots, \tau_{i-1}), \tag{10}$$

where $P(\tau_i \mid \tau_1, \ldots, \tau_{i-1})$ denotes the conditional probability of token τ_i given its preceding context, computed by the LLM via next-token prediction. The surprisal of a constant c_j is then computed as the average surprisal of its constituent tokens:

$$\phi(c_j) = \frac{1}{|\text{tokens}(c_j)|} \sum_{\tau_i \in \text{tokens}(c_j)} \phi(\tau_i). \tag{11}$$

We sample constants for rewriting based on their semantic importance scores defined in Eq. (11). We convert surprisal of constants into rewriting probabilities using a softmax-style transformation:

$$\pi_j = \frac{\exp(-\lambda \cdot \phi(c_j))}{\sum_{c_k \in C} \exp(-\lambda \cdot \phi(c_k))}, \tag{12}$$

where $\lambda > 0$ is a temperature parameter that controls the sharpness of the distribution, and we set it to 0.8 to obtain a relatively smooth distribution. Lower-surprisal (i.e., less informative) constants receive higher rewriting probabilities.

We then perform independent Bernoulli sampling to select constants for rewriting. For each $c_j \in C$, we draw a binary variable $z_j \sim \text{Bernoulli}(\pi_j)$, where $z_j = 1$ indicates that c_j is selected. We subsequently prompt an LLM to perform semantic-preserving rewriting for the selected constants and generate a rewritten log x'. Since the rewritten constants may differ in lexical form or token length, we update the corresponding parts of the original template $T(x)$ to obtain a new template $T(x')$ for the generated log variant.

Semantic rewriting. While variable substitution and constant-inclusive rewriting introduce localized perturbations, they are inherently limited by the structure and semantics of the original log x. To enable broader exploration of error-triggering patterns, we introduce a semantic rewriting strategy that allows changing log semantics to generate log variants, whereas the variants should still can trigger a target error type. Semantic rewriting contains two modes:

- *Error-preserving rewriting.* Given a known error type h_i and a log x that triggers h_i, we prompt an LLM to rewrite x into a variant that remains capable of inducing the same error. This strategy does not enforce strict semantic equivalence, thus enabling broad exploration of diverse variants. To steer generation, we use a prompt that describes the error type h_i and instruct the LLM to generate a variant capable of eliciting the same error type.
- *Error-inducing rewriting.* Given a target error type h_i and an arbitrary log $\hat{x}$, where $\hat{x}$ may or may not trigger any error, this strategy rewrites $\hat{x}$ into a new variant x' that is expected to trigger h_i. Unlike error-preserving rewriting, this mode does not assume that the original log belongs to the error set E. Instead, it enables the transformation of benign or unrelated logs into counterfactual examples aligned with the target error type. This indicates log variants can be generated from new anchors, but not limited to logs in E, which can bring more diversity. To guide the rewriting process, we prompt an LLM with a description of h_i and instruct it to modify the input log accordingly.

Generated variants may not always successfully trigger the intended error type. To address this, each generated log variant is re-evaluated by the target parser. We then reuse our `Analyzer` module to determine whether the variant exhibits the expected error type. For variants belong to corresponding error types, the variants are retained. Otherwise, variants are discarded, and we will regenerate variants until we succeed.

IV. Evaluation

We conduct extensive experiments on the publicly available LogHub [65] dataset to answer the following research questions. (1) **RQ1.** We evaluate the effectiveness of `TraceDoctor` and compare with baselines. (Section IV-A). (2) **RQ2.** We reveal common and distinct high-level error types across different models and datasets (Section IV-B). (3) **RQ3.** We conduct an ablation study to verify the effectiveness of key components in `TraceDoctor` (Section IV-C). (4) **RQ4.** We validate that `TraceDoctor` can be used to enhance non-fine-tuning solutions (Section IV-D).

A. *Comparison with Fine-tuning-based Solutions*

We compare our `TraceDoctor` with SOTA fine-tuning-based log parsing methods. Overall, experimental results show that, on 5 models and 14 datasets, our method consistently achieves better fine-tuning performance. Notably, `TraceDoctor` requires fewer labeled log instances to reach superior parsing accuracy.

Baselines and hyperparameters. We compare `TraceDoctor` against three SOTA fine-tuning-based solutions: LLMParser [30], Superlog [21], and LogLM [28]. We reproduce these baseline solutions using their open-source implementations. For fair comparison, all baselines extract 200 logs and templates from LogHub-2k [65] dataset to construct their fine-tuning sets with their data selection strategies. Other hyperparameters follow the default settings in their open-source repositories.

Our `TraceDoctor` first extracts the error set from LogHub-2k and categorizes the logs into high-level error types. We then randomly sample 50 seed logs and their corresponding templates from the error set, ensuring a roughly balanced number of samples across different high-level error types. For each seed log, we randomly select three variant generation strategies and apply each once to generate three distinct log variants. This results in a total of 150 log variants, which are combined with the original 50 logs to form a fine-tuning dataset containing 200 log-template pairs. The LLM (M) model selected is DeepSeek-V3-0324 [8], and the embedding model selected is text-embedding-ada-002 by Open-AI [34].

Datasets and models. To avoid overlap with the fine-tuning data, we evaluate all methods on LogHub-2.0 [65], a large benchmark containing over 40 million log entries across 14 diverse datasets. We evaluate all solutions on five SOTA open-source reasoning models, *i.e.,* DeepSeek-R1-Distill-Llama-8B [9], DeepSeek-R1-Distill-Qwen-7B [9], Qwen3-14B [39], Phi-4-reasoning-plus [33], and Skywork-o1-Open-Llama-3.1-8B [15]. All models are fine-tuned using the default hyperparameters provided by the LLaMA-Factory framework [60], a widely used training infrastructure for LLMs.

Evaluation metrics. Following previous art [30], [20], we choose two widely used comparison metrics:

- *Parsing Accuracy (PA)*: Parsing accuracy within the log template should match the ground truth template. All variables in the logs must be correctly identified.
- *Group Accuracy (GA)*: Group accuracy does not directly measure the accuracy of log parsing. Instead, it evaluates the correctness of the log grouping. It focuses on verifying whether log messages with the same template are accurately grouped together.

Results. Table I presents the comparison results between `TraceDoctor` and existing baselines. Across a wide range of reasoning models and log datasets, `TraceDoctor` consistently outperforms prior finetuning-based log parsers. In terms of average performance (denoted as Avg in the table), our approach achieves over 0.880 PA and 0.829 GA on each model, whereas baselines typically fall below 0.780 PA and 0.73 GA. More specifically, on the HPC dataset with DeepSeek-R1-Distill-Llama-8B, `TraceDoctor` achieves a PA of 0.988, while LogParser, SuperLog, and LogLM achieve only 0.955, 0.883, and 0.942, respectively. On DeepSeek-R1-Distill-Qwen-7B, our method achieves average PA and GA scores of 0.880 and 0.829, while the best-performing baseline (LogParser) only reaches 0.771 PA and 0.692 GA. These results validate the effectiveness of our approach.

We also observe that in rare cases, baselines slightly outperform `TraceDoctor`. For example, on Apache, LogParser achieves 0.988 PA, while our method yields 0.966. We attribute this to the relatively low diversity in Apache logs, which allows LogParser to memorize key structures via its uniform sampling strategy during training. However, in most other cases where log diversity is higher, memorization becomes difficult and our method significantly outperforms LogParser.

B. *High-level Error Types Identification*

This section characterizes the high-level error types from two aspects. First, we quantify the number of high-level error types identified for each model on each dataset. This provides a basic overview of how error types are distributed across different model–dataset combinations. Second, we aggregate all error types identified across models and datasets and remove duplicates to obtain the set of unique error types. Based on the resulting set, we analyze the frequency of each error type to identify which categories occur most frequently. This analysis helps reveal common failure patterns that are shared across models and datasets, offering insights for future improvements.

Results. Table II reports the number of distinct error types identified for each model on each dataset. For example, on the DeepSeek-R1-Distill-Llama-8B model and Linux dataset, `TraceDoctor` identifies 10 high-level error types. Further, `TraceDoctor` identifies, in total, 29 distinct error types across all models and datasets. Figure 3 shows the distribution of the most frequent error types. The top six high-level error types are:

- **Variable segmentation errors**: Models typically decompose compound single variables into multiple variables, such as treating IP addresses and port numbers as separate variables rather than as a single entity.
- **Contextual over-analysis**: Models excessively interpret certain nouns or numbers, identifying them as variables or constants while disregarding their semantic context.
- **Over-specification of variable names**: During output, models parse numerical variables as specific pronouns, such as interpreting IP addresses as `<IP>` rather than `<*>`.
- **Key-value template mishandling**: When encountering key-value pair variables, models often treat the entire pair as a variable, such as mistakenly identifying `port:400` as a single variable.
- **Non-standard output format**: When outputting parsed logs, models may disrupt the original log structure, for

TABLE I

A comparison of the PA and GA across different systems and models. The highest values of PA and GA are indicated in bold.

Method	Metric	Apache	HDFS	SSH	Mac	Stack	Linux	Spark	Health	Proxifier	BGL	Hadoop	Zookeep	HPC	Thund	Avg
DeepSeek-R1-Distill-Llama-8B																
LogParser	PA	0.990	0.999	0.705	0.512	0.999	0.118	0.944	0.947	0.711	0.954	0.775	0.985	0.955	0.359	0.782
	GA	0.862	0.800	0.751	0.540	0.913	0.669	0.677	0.861	0.592	0.773	0.684	0.708	0.785	0.543	0.726
SuperLog	PA	0.921	0.919	0.652	0.476	0.929	0.110	0.878	0.876	0.654	0.883	0.717	0.906	0.883	0.335	0.724
	GA	0.701	0.648	0.615	0.443	0.749	0.552	0.562	0.706	0.528	0.634	0.561	0.581	0.643	0.450	0.598
LogLM	PA	0.960	**0.999**	0.646	0.673	0.986	0.084	0.933	0.918	0.702	0.955	0.694	0.983	0.942	0.337	0.772
	GA	0.862	0.900	0.688	0.556	0.870	0.683	0.638	0.782	0.546	0.753	0.602	0.708	0.723	0.569	0.706
`TraceDoctor`	PA	**0.991**	0.992	**0.819**	**0.721**	**0.999**	**0.825**	**0.969**	**0.951**	**0.742**	**0.960**	**0.803**	**0.989**	**0.988**	**0.692**	**0.889**
	GA	**0.902**	**0.925**	**0.761**	**0.684**	**0.959**	**0.804**	**0.805**	**0.874**	**0.868**	**0.893**	**0.742**	**0.959**	**0.899**	**0.814**	**0.849**
DeepSeek-R1-Distill-Qwen-7B																
LogParser	PA	0.989	0.999	0.633	0.492	0.988	0.089	0.959	0.949	0.715	0.925	0.772	0.985	0.955	0.350	0.771
	GA	0.690	0.768	0.594	0.513	0.913	0.648	0.654	0.868	0.676	0.781	0.610	0.719	0.785	0.528	0.692
SuperLog	PA	0.930	0.946	0.669	0.487	0.952	0.112	0.896	0.898	0.667	0.898	0.732	0.932	0.905	0.343	0.741
	GA	0.737	0.706	0.664	0.478	0.816	0.596	0.613	0.763	0.573	0.685	0.606	0.633	0.697	0.491	0.647
LogLM	PA	0.958	0.999	0.589	0.644	0.975	0.079	0.931	0.885	0.693	0.944	0.654	0.984	0.947	0.211	0.750
	GA	0.655	0.700	0.469	0.487	0.848	0.639	0.567	0.729	0.636	0.700	0.511	0.697	0.739	0.389	0.626
`TraceDoctor`	PA	**0.989**	**0.999**	**0.813**	**0.717**	**0.998**	**0.730**	**0.962**	**0.954**	**0.736**	**0.953**	**0.797**	**0.995**	**0.980**	**0.688**	**0.880**
	GA	**0.878**	**0.897**	**0.731**	**0.666**	**0.935**	**0.784**	**0.785**	**0.894**	**0.844**	**0.869**	**0.724**	**0.934**	**0.876**	**0.794**	**0.829**
Qwen3-14b																
LogParser	PA	**0.988**	0.999	0.699	0.580	0.999	0.093	0.964	0.947	0.704	0.913	0.784	0.983	0.954	0.418	0.787
	GA	0.793	0.900	0.750	0.628	0.913	0.763	0.748	0.907	0.652	0.858	0.740	0.888	0.785	0.655	0.784
SuperLog	PA	0.858	0.852	0.602	0.438	0.857	0.101	0.806	0.808	0.600	0.809	0.658	0.838	0.815	0.309	0.668
	GA	0.683	0.691	0.598	0.430	0.734	0.536	0.551	0.686	0.516	0.622	0.545	0.570	0.627	0.442	0.588
LogLM	PA	0.882	0.999	0.661	0.682	0.983	0.091	0.969	0.938	0.701	0.949	0.744	0.986	0.946	0.397	0.780
	GA	0.621	0.800	0.688	0.590	0.913	0.678	0.748	0.894	0.455	0.850	0.667	0.764	0.692	0.683	0.717
`TraceDoctor`	PA	0.966	**0.999**	**0.844**	**0.735**	**0.999**	**0.849**	**0.980**	**0.980**	**0.765**	**0.949**	**0.787**	**0.993**	**0.962**	**0.674**	**0.891**
	GA	**0.874**	**0.950**	**0.812**	**0.720**	**0.942**	**0.764**	**0.846**	**0.912**	**0.824**	**0.858**	**0.774**	**0.935**	**0.944**	**0.854**	**0.858**
Phi-4-reasoning-plus																
LogParser	PA	0.958	0.999	0.698	0.506	0.977	0.095	**0.984**	0.938	0.731	0.940	0.680	0.984	0.952	0.391	0.774
	GA	0.789	0.800	0.750	0.490	0.870	0.678	0.724	0.815	0.641	0.773	0.684	0.719	0.800	0.549	0.720
SuperLog	PA	0.967	0.913	0.642	0.467	0.940	0.117	0.860	0.862	0.640	0.862	0.702	0.894	0.941	0.361	0.726
	GA	0.773	0.685	0.684	0.459	0.857	0.626	0.631	0.732	0.550	0.714	0.576	0.658	0.725	0.469	0.653
LogLM	PA	0.976	0.999	0.698	0.507	0.977	0.095	0.984	0.937	0.714	0.942	0.680	0.984	0.952	0.391	0.774
	GA	0.828	0.800	0.750	0.494	0.870	0.689	0.709	0.808	0.636	0.777	0.684	0.742	0.815	0.562	0.726
`TraceDoctor`	PA	**0.996**	**0.999**	**0.827**	**0.757**	**0.998**	**0.832**	0.959	**0.990**	**0.780**	**0.960**	**0.771**	**0.993**	**0.981**	**0.688**	**0.895**
	GA	**0.918**	**0.962**	**0.752**	**0.689**	**0.893**	**0.731**	**0.888**	**0.853**	**0.858**	**0.823**	**0.809**	**0.898**	**0.907**	**0.820**	**0.843**
Skywork-o1-Open-Llama-3.1-8B																
LogParser	PA	0.967	0.999	0.696	0.478	0.975	0.095	0.931	0.928	0.724	0.852	0.693	0.985	0.951	0.351	0.759
	GA	0.728	0.950	0.656	0.500	0.848	0.743	0.646	0.775	0.545	0.729	0.602	0.787	0.800	0.554	0.705
SuperLog	PA	0.883	0.965	0.685	0.462	0.895	0.113	0.852	0.850	0.687	0.927	0.690	0.879	0.856	0.352	0.721
	GA	0.729	0.626	0.591	0.456	0.787	0.530	0.541	0.739	0.549	0.609	0.578	0.610	0.675	0.432	0.604
LogLLM	PA	0.988	0.999	0.653	0.608	0.991	0.095	0.937	0.949	0.703	0.852	0.690	0.967	0.955	0.359	0.768
	GA	0.819	0.900	0.688	0.583	0.913	0.719	0.724	0.927	0.636	0.717	0.597	0.764	0.815	0.614	0.744
`TraceDoctor`	PA	**0.992**	**0.999**	**0.852**	**0.772**	**0.998**	**0.815**	**0.940**	**0.970**	**0.818**	**0.979**	**0.810**	**0.992**	**0.961**	**0.715**	**0.901**
	GA	**0.955**	**0.985**	**0.707**	**0.696**	**0.933**	**0.768**	**0.935**	**0.953**	**0.806**	**0.794**	**0.834**	**0.985**	**0.963**	**0.795**	**0.865**

example, by introducing symbols that did not exist in the original log.
- **Structural bracket preservation failure**: When parsing variables such as `(<*>)`, models typically include the parentheses as part of the variable during substitution.

These results uncover both shared and divergent patterns of parsing errors across different models and datasets, offering insights into recurring failure modes. For example, the identification of high-frequency error types may inform future efforts to design dataset- or model-specific improvements, such as prioritizing robustness in areas where parsing errors are most concentrated.

TABLE II
COUNTS OF DISTINCT ERROR TYPES.

Dataset	Ds-llama3-8b	Ds-qwen-7b	Qwen3-14b	Phi-4-plus	Skywork
Apache	6	6	8	6	6
BGL	10	7	6	6	8
Hadoop	8	6	6	7	6
HDFS	6	7	5	4	7
HealthApp	8	7	6	7	8
HPC	7	6	7	5	7
Linux	10	6	6	5	7
Mac	8	7	6	7	7
OpenSSH	8	6	6	7	6
OpenStack	8	7	6	6	8
Proxifier	7	6	6	5	6
Spark	10	8	8	5	6
Thunderbird	6	6	5	5	5
Zookeeper	7	7	5	5	6

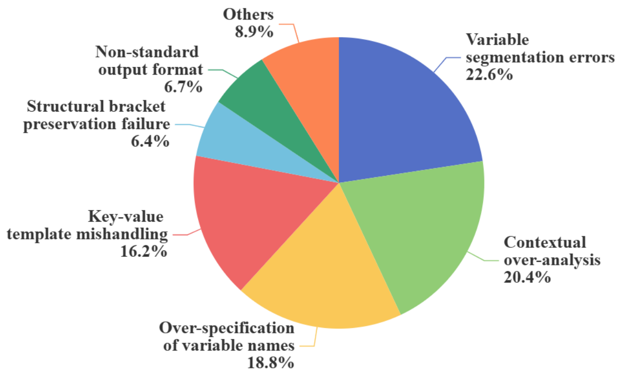


Fig. 3. Proportion of Error Types.

C. Ablation Study

We conduct an ablation study and use the DeepSeek-R1-Distill-Llama-8B model as an example to verify the effectiveness of key components in `TraceDoctor`. We present the average GA and PA across different datasets.

Effectiveness of reasoning-guided generation. We first verify the effectiveness of analyzing reasoning traces to guide log variant generation. To this end, we compare `TraceDoctor` with a non-guided setting, denoted as **Non-analysis**, where the model directly uses misparsed logs to prompt the LLM for variant generation, without analyzing the associated reasoning traces. This comparison highlights the value of understanding the causes of parsing errors rather than relying on surface-level parsing errors.

Effectiveness of variant generation strategies. We then verify the effectiveness of the three variant generation strategies in the `Generator` module. We consider the following four configurations: (1) **Aug1-only**, where only the variable substitution strategy is enabled, (2) **Aug2-only**, where only the constant-inclusive rewriting strategy is enabled, (3) **Aug3-only**, where only the semantic rewriting strategy is enabled, (4) **Non-Aug**, where all augmentation strategies are disabled and no variants are generated.

Results. The experimental results are shown in Table III, show that the complete `TraceDoctor` framework achieves the best performance, confirming that each component plays an important and complementary role. In particular, removing reasoning-based analysis (Non-analysis) causes the largest performance drop. This result shows that analyzing the model's reasoning traces is crucial for identifying the true causes of parsing errors. Without such analysis, the generated variants tend to imitate surface-level patterns and fail to address the actual reasoning flaws.

We also evaluate the impact of the three variant generation strategies. Each individual strategy brings noticeable improvement over the setting without augmentation (Non-Aug). Variable substitution (Aug1-only) achieves the highest gain among the three. Constant-inclusive rewriting (Aug2-only) introduces controlled variation in constants, which helps expose different log patterns. Semantic rewriting (Aug3-only) makes larger changes to the log while preserving its meaning, further increasing diversity. Although semantic rewriting alone brings slightly smaller improvement, it complements the other two strategies well. When all three strategies are used together, the model achieves the highest overall performance. This confirms that combining the strategies leads to better generalization than using any single one alone.

D. Enhancing Non-fine-tuning Solutions

To evaluate whether `TraceDoctor` can enhance non-fine-tuning solutions, we select four SOTA non-fine-tuning solutions as representative baselines: Divlog [54], LUNAR [20], LogBatcher [51], and LILAC [22]. We conduct experiments using DeepSeek-R1-Distill-Llama-8B as the base model. For each solution, we evaluate its log parsing performance in two settings: (1) **Base**, where the solution is deployed with the non-fine-tuned model, and (2) **Improvement**, where the solution is deployed with the enhanced model that has been fine-tuned by `TraceDoctor`.

Table IV shows that `TraceDoctor` significantly boosts the performance of non-fine-tuning solutions. This indicates that our approach is orthogonal and complementary to existing non-fine-tuning solutions. In particular, we report the performance gains as ΔPA and ΔGA, defined as the PA and GA improvements achieved when switching from the base model to the enhanced model. Across all datasets and solutions, we observe consistent improvements. For instance, on the OpenStack dataset, `TraceDoctor` improves PA by 0.189 and GA by 0.209 over LILAC. Taking another example, on the Thunderbird dataset, `TraceDoctor` improves PA by 0.156 and GA by 0.386 over DivLog. The average improvements across all solutions range from 0.133 to 0.304 in PA and from 0.035 to 0.371 in GA. These results demonstrate that `TraceDoctor` can be effectively combined with existing non-fine-tuning solutions to enhance log parsing performance.

TABLE III
ABLATION STUDY: ANALYZING THE IMPACT OF DIFFERENT COMPONENTS.

Methods	Ds-llama3-8b		Ds-qwen-7b		Qwen3-14b		Phi-4-plus		Skywork	
	PA	GA	PA	GA	PA	GA	PA	GA	PA	GA
`TraceDoctor`	**0.721**	**0.684**	**0.717**	**0.666**	**0.735**	**0.720**	**0.757**	**0.689**	**0.772**	**0.696**
Non-analysis	0.510	0.593	0.521	0.599	0.524	0.607	0.515	0.610	0.527	0.601
Aug1-only	0.703	0.669	0.695	0.645	0.713	0.684	0.721	0.668	0.727	0.675
Aug2-only	0.648	0.627	0.662	0.638	0.667	0.643	0.655	0.641	0.669	0.635
Aug3-only	0.621	0.605	0.635	0.617	0.639	0.621	0.633	0.619	0.642	0.612
Non-Aug	0.619	0.573	0.632	0.586	0.635	0.591	0.628	0.589	0.641	0.577

TABLE IV
A COMPARISON OF THE PA AND GA ACROSS THE BASELINE MODEL AND THE ENHANCED MODEL.

Dataset	DivLog				LUNAR				LogBatcher				LILAC			
	Base		Improved		Base		Improved		Base		Improved		Base		Improved	
	PA	GA	ΔPA	ΔGA	PA	GA	ΔPA	ΔGA	PA	GA	ΔPA	ΔGA	PA	GA	ΔPA	ΔGA
Apache	0.963	0.951	+0.012	+0.013	0.694	0.953	+0.143	+0.011	0.968	0.923	+0.006	-0.038	0.993	0.542	+0.004	+0.458
BGL	0.506	0.119	+0.457	+0.677	0.803	0.900	+0.088	+0.041	0.783	0.827	+0.188	+0.081	0.773	0.852	+0.204	+0.141
Hadoop	0.545	0.222	+0.361	+0.640	0.162	0.921	+0.412	-0.014	0.257	0.855	+0.266	+0.066	0.362	0.890	+0.561	+0.052
HDFS	0.604	0.158	+0.395	+0.789	0.623	0.785	+0.324	+0.162	0.778	0.903	+0.071	+0.030	0.964	0.961	+0.036	+0.039
HealthApp	0.798	0.847	+0.044	+0.044	0.420	0.880	+0.047	+0.084	0.792	0.893	+0.028	+0.065	0.822	0.869	+0.054	+0.128
HPC	0.929	0.882	+0.033	-0.069	0.789	0.891	+0.068	+0.019	0.793	0.811	+0.036	+0.026	0.648	0.989	+0.278	-0.074
Linux	0.162	0.498	+0.811	+0.380	0.139	0.852	+0.750	+0.046	0.103	0.920	+0.316	-0.008	0.157	0.395	+0.807	+0.599
Mac	0.500	0.207	+0.091	+0.206	0.180	0.949	+0.195	-0.080	0.509	0.881	+0.007	+0.028	0.282	0.708	+0.241	+0.052
OpenSSH	0.961	0.227	+0.030	+0.487	0.552	0.815	+0.022	+0.078	0.671	0.827	+0.021	+0.006	0.751	0.682	+0.184	+0.072
OpenStack	0.924	0.048	+0.050	+0.730	0.127	0.907	+0.161	+0.049	0.103	0.878	+0.518	+0.024	0.775	0.784	+0.189	+0.209
Proxifier	0.722	0.091	+0.033	+0.364	0.631	0.875	+0.187	+0.034	0.816	0.772	+0.084	+0.146	0.010	0.001	+0.989	+0.993
Spark	0.919	0.847	+0.003	+0.012	0.327	0.833	+0.360	+0.089	0.541	0.851	+0.042	+0.019	0.966	0.974	+0.025	+0.025
Thunderbird	0.563	0.167	+0.156	+0.386	0.107	0.940	+0.281	-0.006	0.319	0.898	+0.203	+0.008	0.292	0.832	+0.489	+0.069
Zookeeper	0.920	0.431	+0.073	+0.533	0.450	0.860	+0.053	+0.013	0.614	0.953	+0.071	+0.033	0.565	0.962	+0.196	+0.035
Average	0.715	0.407	+0.182	+0.371	0.429	0.883	+0.221	+0.038	0.575	0.871	+0.133	+0.035	0.598	0.746	+0.304	+0.200

V. DISCUSSION

Robustness to model stochasticity. A potential threat to the stability of our experimental results is the stochastic nature of LLMs. In particular, our `Analyzer` module relies on DeepSeek-V3-0324 to acquire high-level error types. However, LLMs often produce different outputs for the same prompt due to inherent randomness, raising concerns about the consistency of the identified error types. To this end, we conduct five repeated runs of the error-type extraction process with randomness enabled. We then compute the pairwise cosine similarity between the resulting high-level error types by embedding each type using the text-embedding-ada-002 embedding model. The average semantic similarity across these runs ranged from 0.92 to 0.97, indicating that although surface-level phrasing might differ, the underlying semantics remained highly consistent. To further assess whether this variance affects downstream outcomes, we also repeat the fine-tuning and evaluation processes. We observe that both PA and GA metrics vary by less than 5% across runs, indicating that our overall framework exhibits strong robustness.

As for the `Generator` module, although it also employs DeepSeek-V3-0324 to generate log variants, it is inherently designed to introduce variation through random sampling of examples and prompting strategies. Therefore, it naturally tolerates stochastic behavior, and all results reported in this work are averaged over five runs to ensure fairness and reliability.

Reliability of generated templates. In the `Generator` module, each generated log variant is paired with a corresponding template to enable supervised fine-tuning. Since templates are produced by an LLM, the generated templates may occasionally be inaccurate, and no ground truth is available for verification. However, our objective is not to ensure the correctness of every template, but to improve model performance by exposing it to diverse error patterns. Our experiments validate that the generated templates are sufficient to support effective model enhancement. We leave enhancing the accuracy of template generation as future work.

VI. Conclusion

This work proposes `TraceDoctor`, a framework that systematically analyzes reasoning traces associated with LLM log parsing errors and leverages this analysis to generate log variants for fine-tuning. By identifying high-level error types and designing corresponding generation strategies, `TraceDoctor` enhances parsing accuracy across five state-of-the-art reasoning LLMs. Our results validate the effectiveness of reasoning-trace-guided error analysis and variant generation in improving LLM-based log parsers. This work suggests a new perspective for enhancing LLM performance in structured prediction tasks through reasoning-aware error analysis.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (U24A20337); by the National Research Foundation, Singapore, through the National Cybersecurity R&D Lab at the National University of Singapore under its National Cybersecurity R&D Programme (Award No. NCR25-NCL P3-0001); and by the Joint Research Center for System Security (JCSS), Tsinghua University (Institute for Network Sciences and Cyberspace) – Science City (Guangzhou) Digital Technology Group Co., Ltd. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and National Cybersecurity R&D Lab at the National University of Singapore.

References

[1] A. Amar and P. C. Rigby, "Mining historical test logs to predict bugs and localize faults in the test logs," in *Proceedings of the 41st International Conference on Software Engineering, ICSE*. IEEE / ACM, 2019.

[2] S. An, Z. Ma, Z. Lin, N. Zheng, J. Lou, and W. Chen, "Learning from mistakes makes LLM better reasoner," *CoRR*, vol. abs/2310.20689, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.20689

[3] M. Astekin, M. Hort, and L. Moonen, "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*, X. Franch, M. Daneva, S. Martínez-Fernández, and L. Quaranta, Eds. ACM, 2024.

[4] A. Chen, Y. Song, W. Zhu, K. Chen, M. Yang, T. Zhao, and M. Zhang, "Evaluating o1-like llms: Unlocking reasoning for translation through comprehensive analysis," *CoRR*, vol. abs/2502.11544, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2502.11544

[5] J. Chen, W. Shang, A. E. Hassan, Y. Wang, and J. Lin, "An experience report of generating load tests using log-recovered workloads at varying granularities of user behaviour," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2019.

[6] S. Chen, B. Li, and D. Niu, "Boosting of thoughts: Trial-and-error problem solving with large language models," in *The Twelfth International Conference on Learning Representations, ICLR*, 2024.

[7] T. Cui, S. Ma, Z. Chen, T. Xiao, S. Tao, Y. Liu, S. Zhang, D. Lin, C. Liu, Y. Cai, W. Meng, Y. Sun, and D. Pei, "Logeval: A comprehensive benchmark suite for large language models in log analysis," *CoRR*, 2024.

[8] DeepSeek-AI, "Deepseek-v3 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2412.19437

[9] ——, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[10] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *9th International Conference on Learning Representations, ICLR*, 2021.

[11] C. Du, H. Sun, J. Wang, Q. Qi, and J. Liao, "Adversarial and domain-aware BERT for cross-domain sentiment analysis," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL*, 2020.

[12] J. Ebrahimi, H. Yang, and W. Zhang, "How does adversarial fine-tuning benefit bert?" *CoRR*, vol. abs/2108.13602, 2021. [Online]. Available: https://arxiv.org/abs/2108.13602

[13] K. Gao, H. Cai, Q. Shuai, D. Gong, and Z. Li, "Embedding self-correction as an inherent ability in large language models for enhanced mathematical reasoning," *CoRR*, vol. abs/2410.10735, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2410.10735

[14] A. Goodkind and K. Bicknell, "Predictive power of word surprisal for reading times is a linear function of language model quality," in *Proceedings of the 8th Workshop on Cognitive Modeling and Computational Linguistics, CMCL*, 2018.

[15] J. He, T. Wei, R. Yan, J. Liu, C. Wang, Y. Gan, S. Tu, C. Y. Liu, L. Zeng, X. Wang, B. Wang, Y. Li, F. Zhang, J. Xu, B. An, Y. Liu, and Y. Zhou, "Skywork-o1 open series," https://huggingface.co/Skywork, November 2024. [Online]. Available: https://huggingface.co/Skywork

[16] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM Comput. Surv.*, 2022.

[17] S. He, Q. Lin, J. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. ACM, 2018.

[18] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *27th IEEE International Symposium on Software Reliability Engineering, ISSRE*. IEEE Computer Society, 2016.

[19] J. Hu and Y. Cong, "Modeling Chinese L2 writing development: The LLM-surprisal perspective," in *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics*, 2025.

[20] J. Huang, Z. Jiang, Z. Chen, and M. R. Lyu, "Lunar: Unsupervised llm-based log parsing," *arXiv preprint arXiv:2406.07174*, 2024.

[21] Y. Ji, Y. Liu, F. Yao, M. He, S. Tao, X. Zhao, C. Su, X. Yang, W. Meng, Y. Xie, B. Chen, and H. Yang, "Adapting large language models to log analysis with interpretable domain knowledge," *CoRR*, 2024.

[22] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu, "LILAC: log parsing using llms with adaptive parsing cache," *Proc. ACM Softw. Eng.*, no. FSE, 2024.

[23] A. Karimi, L. Rossi, and A. Prati, "Adversarial training for aspect-based sentiment analysis with BERT," in *25th International Conference on Pattern Recognition, ICPR 2020, Virtual Event / Milan, Italy, January 10-15, 2021*, 2020.

[24] V. Le and H. Zhang, "Log parsing: How far can chatgpt go?" in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2023.

[25] ——, "Log parsing with prompt-based few-shot learning," in *45th IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 2023.

[26] C. Li, G. Dong, M. Xue, R. Peng, X. Wang, and D. Liu, "Dotamath: Decomposition of thought with code assistance and self-correction for mathematical reasoning," *CoRR*, vol. abs/2407.04078, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.04078

[27] X. Li, W. Wang, M. Li, J. Guo, Y. Zhang, and F. Feng, "Evaluating mathematical reasoning of large language models: A focus on error identification and correction," in *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 11 316–11 360. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.673

[28] Y. Liu, Y. Ji, S. Tao, M. He, W. Meng, S. Zhang, Y. Sun, Y. Xie, B. Chen, and H. Yang, "Loglm: From task-based to instruction-based automated log analysis," *arXiv preprint arXiv:2410.09352*, 2024.

[29] Y. Liu, S. Tao, W. Meng, J. Wang, W. Ma, Y. Chen, Y. Zhao, H. Yang, and Y. Jiang, "Interpretable online log analysis using large language models with prompt strategies," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, ICPC*. ACM, 2024.

[30] Z. Ma, A. R. Chen, D. J. Kim, T. Chen, and S. Wang, "Llmparser: An exploratory study on using large language models for log parsing," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE*. ACM, 2024.

[31] Z. Ma, D. J. Kim, and T.-H. Chen, " LibreLog: Accurate and Efficient Unsupervised Log Parsing Using Open-Source Large Language Models ," in *IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025.

[32] M. Mehrabi, A. Hamou-Lhadj, and H. Moosavi, "The effectiveness of compact fine-tuned llms in log parsing," in *IEEE International Conference on Software Maintenance and Evolution, ICSME*. IEEE, 2024.

[33] Microsoft, "Phi-4-reasoning-plus," https://huggingface.co/microsoft/Phi-4-reasoning-plus, 2025, accessed: October 1, 2025.

[34] OpenAI, "Embeddings guide," https://openai.xiniushu.com/docs/guides/embeddings, 2024, openAI Documentation (Chinese Version).

[35] Z. Pan, Y. Li, H. Lin, Q. Pei, Z. Tang, W. Wu, C. Ming, H. V. Zhao, C. He, and L. Wu, "LEMMA: learning from errors for mathematical advancement in llms," *CoRR*, vol. abs/2503.17439, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.17439

[36] D. S. Park, W. Chan, Y. Zhang, C. Chiu, B. Zoph, E. D. Cubuk, and Q. V. Le, "Specaugment: A simple data augmentation method for automatic speech recognition," in *20th Annual Conference of the International Speech Communication Association, Interspeech*, 2019.

[37] C. Pei, Z. Liu, J. Li, E. Zhang, L. Zhang, H. Zhang, W. Chen, D. Pei, and G. Xie, "Self-evolutionary group-wise log parsing based on large language model," in *35th IEEE International Symposium on Software Reliability Engineering, ISSRE*. IEEE, 2024.

[38] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, 1948.

[39] Q. Team, "Qwen3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[40] Y. Tong, D. Li, S. Wang, Y. Wang, F. Teng, and J. Shang, "Can llms learn from previous mistakes? investigating llms' errors to boost for reasoning," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 3065–3080. [Online]. Available: https://doi.org/10.18653/v1/2024.acl-long.169

[41] J. Wang, F. Meng, Y. Liang, and J. Zhou, "Drt-o1: Optimized deep reasoning translation via long chain-of-thought," *CoRR*, vol. abs/2412.17498, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2412.17498

[42] J. Wang, H. Qiu, Y. Rong, H. Ye, Q. Li, Z. Li, and C. Zhang, "BET: black-box efficient testing for convolutional neural networks," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2022.

[43] J. Wang, W. Qu, Y. Rong, H. Qiu, Q. Li, Z. Li, and C. Zhang, "Mpass: Bypassing learning-based static malware detectors," in *60th ACM/IEEE Design Automation Conference, DAC*. IEEE, 2023.

[44] J. Wang, Y. Wu, W. Xu, Y. Huang, C. Zhang, Z. Li, M. Xu, and Z. Liang, "Your scale factors are my weapon: Targeted bit-flip attacks on vision transformers via scale factor manipulation," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR*. Computer Vision Foundation / IEEE, 2025.

[45] J. Wang, C. Zhang, L. Chen, Y. Rong, Y. Wu, H. Wang, W. Tan, Q. Li, and Z. Li, "Improving ml-based binary function similarity detection by assessing and deprioritizing control flow graph features," in *33rd USENIX Security Symposium, USENIX Security*. USENIX Association, 2024.

[46] J. Wang, Z. Zhang, M. Wang, H. Qiu, T. Zhang, Q. Li, Z. Li, T. Wei, and C. Zhang, "Aegis: Mitigating targeted bit-flip attacks against deep neural networks," in *32nd USENIX Security Symposium, USENIX Security*. USENIX Association, 2023.

[47] J. Wang, M. Fang, Z. Wan, M. Wen, J. Zhu, A. Liu, Z. Gong, Y. Song, L. Chen, L. M. Ni, L. Yang, Y. Wen, and W. Zhang, "Openr: An open source framework for advanced reasoning with large language models," *CoRR*, vol. abs/2410.09671, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2410.09671

[48] Z. Wang, H. Zhang, T. P. Chen, and S. Wang, "Would you like a quick peek? providing logging support to monitor data processing in big data applications," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021.

[49] J. W. Wei and K. Zou, "EDA: easy data augmentation techniques for boosting performance on text classification tasks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP*, 2019.

[50] Y. Wu, S. Yu, and Y. Li, "Log parsing with self-generated in-context learning and self-correction," *CoRR*, vol. abs/2406.03376, 2024.

[51] Y. Xiao, V. Le, and H. Zhang, "Demonstration-free: Towards more practical log parsing with large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE*. ACM, 2024.

[52] Q. Xie, Z. Dai, E. H. Hovy, T. Luong, and Q. Le, "Unsupervised data augmentation for consistency training," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS*, 2020.

[53] A. Xu and A. Gau, "HELP: hierarchical embeddings-based log parsing," *CoRR*, vol. abs/2408.08300, 2024.

[54] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He, "Divlog: Log parsing with prompt enhanced in-context learning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE*. ACM, 2024.

[55] D. Yang, T. Liu, D. Zhang, A. Simoulin, X. Liu, Y. Cao, Z. Teng, X. Qian, G. Yang, J. Luo, and J. J. McAuley, "Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms," *CoRR*, vol. abs/2502.19411, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2502.19411

[56] X. Yu, S. Nong, D. He, W. Zheng, T. Ma, N. Liu, J. Li, and G. Xie, "Loggenius: An unsupervised log parsing framework with zero-shot prompt engineering," in *IEEE International Conference on Web Services, ICWS*, 2024.

[57] W. Zhang, H. Guo, A. Le, J. Yang, J. Liu, Z. Li, T. Zheng, S. Xu, R. Zang, L. Zheng, and B. Zhang, "Lemur: Log parsing with entropy sampling and chain-of-thought merging," *CoRR*, 2024.

[58] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. ACM, 2019.

[59] Y. Zhang, M. Khalifa, L. Logeswaran, J. Kim, M. Lee, H. Lee, and L. Wang, "Small language models need strong verifiers to self-correct reasoning," in *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 15 637–15 653. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.924

[60] Y. Zheng, R. Zhang, J. Zhang, Y. Ye, Z. Luo, Z. Feng, and Y. Ma, "Llamafactory: Unified efficient fine-tuning of 100+ language models," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Bangkok, Thailand: Association for Computational Linguistics, 2024. [Online]. Available: http://arxiv.org/abs/2403.13372

[61] C. Zhi, L. Cheng, M. Liu, X. Zhao, Y. Xu, and S. Deng, "Llm-powered zero-shot online log parsing," in *IEEE International Conference on Web Services, ICWS*, 2024.

[62] A. Zhong, D. Mo, G. Liu, J. Liu, Q. Lu, Q. Zhou, J. Wu, Q. Li, and Q. Wen, "Logparser-llm: Advancing efficient log parsing with large language models," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD*. ACM, 2024.

[63] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang, "Random erasing data augmentation," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI*, 2020.

[64] D. Zhu, W. Lin, Y. Zhang, Q. Zhong, G. Zeng, W. Wu, and J. Tang, "AT-BERT: adversarial training BERT for acronym identification winning solution for sdu@aaai-21," in *Proceedings of the Workshop on Scientific*

Document Understanding co-located with 35th AAAI Conference on Artificial Inteligence, SDU@AAAI 2021, Virtual Event, February 9, 2021, 2021.

[65] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, “Loghub: A large collection of system log datasets for ai-driven log analytics,” in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 355–366.