

DeepParse: Hybrid Log Parsing with LLM-Synthesized Regex Masks

Amir Shetaia
Electrical and Computer Engineering
Queen’s University
Ontario, Canada
a.shetaia@queensu.ca

Sean Kauffman
Electrical and Computer Engineering
Queen’s University
Ontario, Canada
sean.k@queensu.ca

Abstract

Modern distributed systems produce massive, heterogeneous logs essential for reliability, security, and anomaly detection. Converting these free-form messages into structured templates (log parsing) is challenging due to evolving formats and limited labeled data. Machine-learning-based parsers like *Drain* are fast but accuracy often degrades on complex variables, while Large Language Models (LLMs) offer better generalization but incur prohibitive inference costs. This paper presents DeepParse, a hybrid framework that automatically mines reusable variable patterns from small log samples using an LLM, then applies them deterministically through the *Drain* algorithm. By separating the reasoning phase from execution, DeepParse enables accurate, scalable, and cost-efficient log structuring without relying on brittle handcrafted rules or per-line neural inference. Across 16 benchmark datasets, DeepParse achieves higher accuracy in variable extraction (97.6% average Parsing Accuracy) and better consistency than both heuristic and LLM-only baselines. Integrating DeepParse into an anomaly detection pipeline reduced false alarms by over 30% and reduced inference latency by 36% compared to heuristic baselines.

1 Introduction

Modern software and cyber-physical platforms emit a continuous stream of diagnostic logs that record events across distributed components, user interac-

tions, and infrastructure services. These records are indispensable for site reliability engineering (SRE), security monitoring, and post-incident analysis because they provide the only chronological evidence of what a system actually did in production. However, the rapid adoption of microservices, cloud-native orchestration, and machine learning-driven services has dramatically increased both the volume and heterogeneity of logs [2]. Operators must therefore interpret log messages and recover their structure so that downstream analytics can query and reason about them. Automating this transformation, referred to as *log parsing*, is the central problem addressed in this work.

Although logs are usually generated from structured templates in source code, this structure is often not available at analysis time because the source code, formatting rules, or template definitions may be inaccessible or inconsistently maintained across teams. In practice, analysts often see only the final text emitted by the system, not the underlying template, which makes it necessary to automatically infer the structure from the logs themselves.

Standard approaches to log analysis often rely on manually written regular expressions for each subsystem [12]. While effective for small, static environments, this method quickly becomes impractical in modern distributed systems. Each service, version, and configuration introduces new patterns that must be continuously updated, and even a single misplaced character can cause silent parsing failures. Furthermore, manually authored expressions are rarely documented or version-controlled, making it difficult to

reproduce analyses or audit system behavior after an incident [12].

Traditional log parsers such as **Drain** [11], **Logram** [6], and **LogCluster** [22] are generally effective at grouping similar log messages into event clusters but frequently fail to accurately identify the variable fields within each message. In other words, while such parsers can detect that two messages belong to the same event type (achieving high *Grouping Accuracy*), they often do not correctly extract or generalize the variable components such as user names, IP addresses, or numeric values within those messages. This distinction between correct grouping and precise variable identification is critical: accurate variable extraction directly impacts downstream tasks such as anomaly detection, root cause analysis, and system monitoring.

Recent studies have demonstrated that Large Language Models (LLMs) can accurately infer the latent structure of semi-structured text [23, 17]. However, naïvely relying on stochastic LLM decoding for every log line reintroduces variability and cost, violates data governance constraints, and incurs prohibitive runtime expenses if deployed at scale. Prior LLM-based approaches to log parsing, such as **LLMParse** [23] and **LogPPT** [17], have explored this potential but encountered specific challenges. **LogPPT** requires manually labeling each word in training data, which is labor-intensive. **LLMParse**, while highly effective at identifying log template structures, often struggles to consistently group semantically similar messages due to the probabilistic variability of LLM-generated outputs.

This paper introduces **DeepParse**, a hybrid framework that addresses these limitations by confining the stochastic reasoning of an LLM to an offline *synthesis phase*, delegating the bulk parsing to a deterministic engine (**Drain**). This combination reconciles accuracy, efficiency, and privacy.

1.1 Problem Statement

Modern distributed systems generate vast amounts of heterogeneous logs that must be transformed into structured representations to enable automated analysis. However, organizations face practical constraints

that complicate this process. Many production environments impose strict data-governance and privacy policies that prohibit uploading operational logs to third-party cloud services. Engineering teams also require solutions that function locally on existing infrastructure without recurring costs or dependence on proprietary APIs. At the same time, the diversity and evolving nature of log formats make manual rule maintenance infeasible, while limited computational budgets restrict the continuous use of large models for inference.

Within this context, the central research problem is to enable accurate and consistent log structuring under realistic operational constraints. This work investigates the following guiding questions:

- Can log parsing remain reliable when formats exhibit inconsistent structure, missing fields, or previously unseen patterns across software releases?
- Can a log analysis framework balance accuracy with computational and economic efficiency under local-execution and cost-sensitive environments?
- Can a parsing system ensure reproducible results without continuous manual intervention or reliance on external services?
- Can limitations in log structuring accuracy significantly influence downstream tasks such as anomaly detection, monitoring, and incident response?

Addressing these questions requires theoretical grounding, empirical validation, and extensive experimentation across diverse operational datasets.

1.2 Contributions

This paper introduces **DeepParse**, a hybrid log parsing framework that unifies deterministic rule mining with neural language modeling to achieve accurate, efficient, and reproducible log structuring at scale. Our contributions are:

1. **Hybrid Architecture:** We design a pipeline that decouples stochastic mask synthesis from deterministic parsing. By separating the reasoning phase (LLM) from the execution phase (**Drain**), we eliminate runtime stochasticity and guarantee

that identical log lines always receive identical templates.

2. **Data Efficiency:** We introduce an entropy-greedy sampling strategy that enables DeepParse to achieve state-of-the-art accuracy from only 50 log examples per system during checkpoint preparation. At deployment time, users run the pre-trained checkpoint on an unlabeled sample without any annotation.
3. **Empirical Evaluation:** On 16 LogHub datasets, DeepParse achieves 97.6% average PA and 94.1% GA while running nearly $100\times$ faster than per-line LLM inference, delivering both high accuracy and production-grade throughput.
4. **Downstream Impact Validation:** We show accurate parsing reduces false alarms in anomaly detection systems by over 30% when integrated into LogBERT. This translates improved PA into concrete operational benefits. Additionally, correct variable masking reduces vocabulary size by 38%, decreasing embedding memory consumption by 3.1 GB and inference latency by 36%.
5. **Replicability and Open Science:** We release the complete DeepParse implementation, including sampling algorithms, fine-tuning scripts, evaluation harnesses, and preprocessed dataset splits. This ensures full reproducibility of our results.¹

The rest of the paper is organized as follows. Section 2 provides background on log parsing formalization and evaluation metrics. Section 3 reviews related work in traditional parsing and LLM applications to logs. Section 4 details the DeepParse architecture, sampling strategy, LLM fine-tuning, and Drain integration. Section 5 describes the experimental setup, datasets, baselines, and implementation details. Section 6 presents quantitative results addressing our research questions. Section 7 demonstrates downstream impact through integration with LogBERT. Section 8 discusses threats to validity. Section 9 concludes with future work directions.

¹<https://github.com/NightBaRron1412/DeepParse>

2 Background

This section formalizes log parsing, defines evaluation metrics, and discusses data governance constraints relevant to our hybrid approach.

2.1 Formal Definition of Log Parsing

Let Σ denote a finite *character alphabet*, and Σ^* the set of all finite strings over Σ . A single log message is a string $\ell \in \Sigma^*$, and a *log corpus* is an ordered sequence of N log lines:

$$D = \langle \ell_1, \ell_2, \dots, \ell_N \rangle \in (\Sigma^*)^*.$$

Each log line can be tokenized into a sequence of words or symbols from a vocabulary $\mathcal{V}$:

$$\text{tok} : \Sigma^* \rightarrow \mathcal{V}^*, \quad \text{tok}(\ell) = \langle w_1, w_2, \dots, w_m \rangle.$$

We denote the full corpus token stream as:

$$\text{Log}(D) = \text{tok}(\ell_1) \cdot \text{tok}(\ell_2) \cdots \text{tok}(\ell_N).$$

A special placeholder token $\langle * \rangle \notin \mathcal{V}$ is used to denote variable fields. Let $\mathcal{T} = (\mathcal{V} \cup \{ \langle * \rangle \})^*$ denote the set of all valid log templates. A *log template* $t \in \mathcal{T}$ represents the invariant structure of similar log messages. Parsing a log line ℓ produces a pair $(t, \mathbf{v})$ where t is the identified template and $\mathbf{v} = \langle v_1, \dots, v_k \rangle$ are the values of the k variable fields (with k depending on t).

We define:

$$\pi : \Sigma^* \rightarrow \mathcal{T} \times (\Sigma^*)^*, \quad g : \Sigma^* \rightarrow \mathbb{N},$$

where π is the parsing function and g assigns each line to a template cluster identifier.

2.2 Evaluation Metrics: Grouping vs. Parsing Accuracy

Log parsing quality is typically assessed using two complementary metrics: *Grouping Accuracy (GA)* and *Parsing Accuracy (PA)* [6, 15].

Grouping Accuracy (GA) measures whether log messages are correctly clustered under the same template. Given a set of log messages with ground-truth group assignments, GA is computed as the proportion of logs assigned to their correct group:

$$GA = \frac{\# \text{ of correctly grouped logs}}{\text{Total number of logs}}.$$

GA evaluates whether the parser recognizes that two messages are structurally similar, regardless of whether it identifies the exact variable boundaries.

Parsing Accuracy (PA) evaluates whether the extracted log template exactly matches the ground truth, including the correct identification of all dynamic variables. PA is computed as:

$$PA = \frac{\# \text{ of logs with perfectly parsed templates}}{\text{Total number of logs}}.$$

PA is therefore a stricter metric than GA. A parser can achieve high GA by clustering messages correctly while still failing to identify variable boundaries precisely, resulting in low PA. Conversely, high PA implies that the parser not only groups messages correctly but also accurately distinguishes static text from dynamic variables.

PA is more indicative of the parser’s usefulness for downstream analyses such as anomaly detection or root cause analysis, where precise variable extraction is essential.

2.3 Data Governance and Privacy Constraints

Growing legislative scrutiny (GDPR, CCPA) and sector-specific standards (HIPAA) impose constraints on how logs are handled. Best-practice frameworks recommend minimizing retained personal data and preventing organizations from uploading operational logs to third-party cloud services. Hybrid parsers must therefore function locally and preserve deterministic behavior even when inputs contain redaction placeholders. Caching synthesized masks locally allows compliance teams to audit exactly which patterns were used to process specific time ranges,

satisfying evidentiary requirements during incident post-mortems.

3 Related Work

Log parsing is a well-researched area with approaches spanning traditional algorithmic techniques and modern LLM-based methods. This section reviews both families and positions DeepParse within the current landscape.

3.1 Traditional Log Pattern Mining Approaches

Traditional log parsers have historically aimed to convert raw log messages into structured formats using algorithmic techniques. These approaches typically fall into three main categories: frequent pattern mining, log clustering, and parsing trees. Techniques such as SLCT [21], LogCluster [22], and Logram [6] identify static text and variables by counting the recurrence of patterns or sequences within log data. SLCT uses token frequency to distinguish constants from variables, assuming that frequently occurring tokens are static. Logram extends this by building an n-gram dictionary to capture multi-token patterns. While computationally efficient, these methods struggle with rare events and diverse variable types, as infrequent tokens may be incorrectly classified as static or variable. Methods including LKE [9], LogSig [20], and LenMa [19] group similar log messages together using clustering algorithms, thereby categorizing logs into distinct groups. LKE employs k-means clustering on log message embeddings, while LogSig uses a weighted edit distance metric. These approaches often achieve high Grouping Accuracy (GA) but may fail to precisely identify variable boundaries, resulting in lower Parsing Accuracy (PA). Parsing tree methods, exemplified by Drain [11], construct a fixed-depth parse tree to guide the log group search process, efficiently identifying patterns and abstracting dynamic values. Drain is recognized for its efficiency and widespread adoption in industry [4]. It operates in linear time $O(N)$ with respect to the number of log messages, making it suitable for high-throughput environments. However,

Drain relies on simple heuristics (e.g., “tokens with digits are variables”) that often under-mask complex identifiers or over-mask version numbers. Despite their advancements and widespread use, recent studies have highlighted a critical limitation: while these methods often achieve high Grouping Accuracy (GA), they frequently fail to accurately identify all dynamic variables within log messages [23]. This “variable identification gap” directly hinders the effectiveness of subsequent log analysis tasks, potentially leading to missed anomalies or inaccurate insights.

3.2 LLMs for Log Understanding Beyond Parsing

Beyond the specific task of log parsing, Large Language Models have demonstrated considerable success in various log-related tasks, leveraging their natural language processing and code generation capabilities [23]. These applications underscore the potential of LLMs to understand the semi-structured nature of logs, which uniquely combine natural language and code-like elements. LLMs have been employed to identify unusual patterns in log data. LAnoBERT [18] utilizes Bidirectional Encoder Representations from Transformers (BERT) for masked word prediction, where significant differences between actual and predicted words indicate anomalies. Studies involving ChatGPT have also shown its effectiveness in anomaly detection [8], with findings partially consistent with on-call engineers, suggesting potential for reducing manual verification efforts. LLMs have been integrated into systems for automating root cause analysis of cloud incidents. RCACopilot, for example, incorporates OpenAI’s GPT models to assist in this complex task [5]. These diverse applications highlight the versatility of LLMs in processing and deriving meaning from log data, extending their utility beyond mere parsing to more complex analytical and diagnostic functions.

3.3 LLM-Based Log Parsing

The exceptional capabilities of LLMs in text understanding, generation, and code-related tasks make

them intuitively suitable for log parsing. While traditional log parsing works well for simple, structured logs, logs containing natural language elements like error descriptions, user actions, or debugging messages are harder to parse yet often contain the most critical diagnostic information. LogPPT [17] relies on prompt-based few-shot learning using pre-trained language models for log parsing. It leverages AI models that predict missing words and requires manually labeling each word in the training data as either a constant or a parameter, which is a labor-intensive process. While LogPPT demonstrates improved parsing accuracy over traditional methods, the manual annotation overhead and the need for carefully designed prompts limit its scalability. LLMParser [23] explores using various LLMs (including T5, ChatGLM, and LLaMA) for direct log parsing. While highly effective at identifying log template structures, LLMParser often struggles to consistently group semantically similar messages together due to the probabilistic variability of LLM-generated outputs. The authors also revealed that simply increasing model complexity or pre-training on logs from other systems might not universally improve accuracy and could even degrade it for certain models. These observations indicate that a direct, unoptimized application of general LLMs or simple cross-system pre-training is not a definitive solution.

Pure LLM parsing is constrained by (i) cost and latency when invoked per log line, (ii) non-determinism from stochastic decoding that breaks auditability, and (iii) data-governance constraints that often prohibit sending operational logs to third-party APIs.

3.4 Positioning of DeepParse

DeepParse addresses the limitations of both traditional and LLM-based parsers through a hybrid architecture that decouples stochastic reasoning from deterministic execution.

By confining LLM invocation to an offline synthesis phase, DeepParse achieves:

- **High Accuracy:** LLM-synthesized regexes capture semantic variable boundaries that heuristics miss, improving PA.
- **Deterministic Execution:** Drain applies fixed

regex masks, ensuring reproducibility and linear-time complexity.

- **Data Efficiency:** Entropy-greedy sampling enables high accuracy with only 50 labeled examples per system.
- **Privacy Compliance:** Local execution and cached masks satisfy data governance requirements.
- **Cost Efficiency:** One-time LLM synthesis amortizes computational costs, avoiding per-line inference expenses.

This design reconciles the semantic understanding of LLMs with the speed and determinism of traditional parsers, positioning DeepParse as a practical solution for production log analysis pipelines.

4 The DeepParse Framework

This section presents the design of the DeepParse framework, detailing its hybrid architecture, sampling strategy, LLM configuration, and integration with the Drain parser.

4.1 Design Overview and Architecture

DeepParse is engineered as a modular pipeline that separates stochastic reasoning from deterministic execution. The framework has three layers: an **Ingestion Layer** for dataset loading, sampling, and pre-processing; a **Synthesis Layer** for LLM fine-tuning and prompt orchestration that maps raw-log batches to regex bundles with provenance metadata; and an **Execution Layer** comprising the deterministic parser (extended **Drain**) and output adapters.

This separation allows the potentially expensive and non-deterministic LLM to run only once during an offline *Installation Phase*, producing a fixed artifact (the mask bundle) that drives the high-speed *Execution Phase*.

We distinguish four phases: **design time** (choosing architecture, datasets, and training), **installation time** (one-off deployment actions such as offline auto-synthesis on a small local sample), **execution time** (steady-state deterministic parsing with a fixed mask

bundle), and **maintenance time** (refreshing masks when schemas drift).

Figure 1 illustrates the DeepParse workflow. An LLM synthesizes regex masks from a small sample of logs offline; these masks then guide a deterministic Drain parser online.

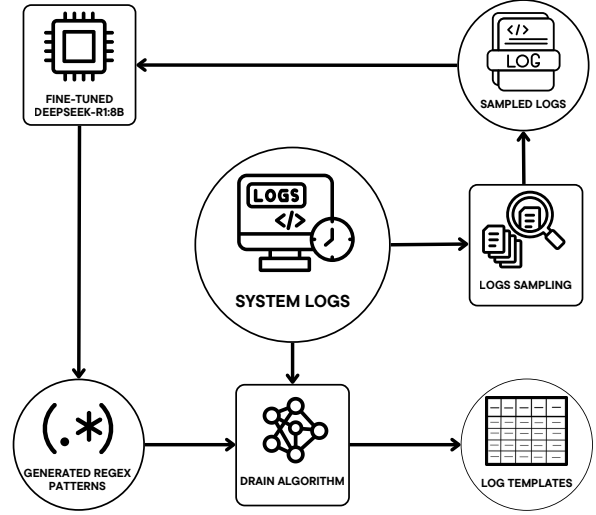


Figure 1: The DeepParse workflow. An LLM synthesizes regex masks from a small sample of logs (offline); these masks guide a deterministic Drain parser (online).

4.2 User Workflow

DeepParse provides a one-click *auto-synthesis* step that invokes a locally-hosted **DeepSeek-R1:8B** checkpoint to emit a regex *mask list* from a small, automatically sampled subset of logs. This step is performed once as an **offline pre-processing phase** prior to deployment. **Users do not label data or fine-tune models**; they invoke a pre-trained checkpoint that we release (Section 4.4 describes the one-time fine-tuning performed by the authors to produce this checkpoint). The full integration workflow is illustrated in Listing 1. In the figure, **synth_masks** performs the offline auto-synthesis, producing patterns loaded into a Drain instance with the **load_masks** method.

Listing 1: User integration: auto-synthesis (fixed checkpoint) to Drain masks

```
# Auto-synthesize regex mask list once (no
  labels, no training)
patterns = synth_masks(sys_logs,
                        sample_size=50,
                        temperature=0,
                        max_length=512)

# Load into Drain as masks and parse
drain = Drain()
drain.load_masks(patterns)
parsed = drain.parse_all(sys_logs)
```

DeepParse first performs **offline auto-synthesis** once: it samples k log lines (default $k=50$) and calls the fixed DeepSeek-R1:8B checkpoint to produce a regex mask list $\mathcal{R}_{\text{auto}}$, which can be cached and reused. At **runtime**, the system loads $\mathcal{R}_{\text{auto}}$ into **Drain** and parses all logs in linear time, $O(N)$. Under **schema drift**, users re-run the same offline auto-synthesis to regenerate $\mathcal{R}_{\text{auto}}$ (no labeling and no model fine-tuning).

4.3 Entropy-Greedy Sampling

Log datasets are often highly skewed, causing random sampling to miss rare but structurally complex messages. We therefore use entropy-greedy sampling to select a small, diverse subset of logs for mask synthesis.

Algorithm 1 implements entropy-greedy sampling. The **normalization step** (Line 1) replaces digits and hexadecimal tokens with placeholder symbols so that rare numeric identifiers do not dominate entropy scores; this ensures entropy reflects structural diversity rather than incidental numeric variation. Crucially, normalization affects only sampling selection—the original (un-normalized) logs are passed to the LLM for mask synthesis, preserving all structural signals.

We define **entropy** as the Shannon entropy over the token-frequency distribution of each normalized

Algorithm 1 Entropy-Greedy Sampling

Require: Log corpus L , target sample size k

- 1: Normalize digits and hexadecimal tokens in L to obtain L'
 - 2: Compute token-frequency distributions for each log in L'
 - 3: Compute entropy score $H(\ell)$ for every $\ell \in L'$
 - 4: Initialize sample set $S \leftarrow \emptyset$
 - 5: **while** $|S| < k$ **do**
 - 6: Select log $\hat{\ell}$ from L' with maximum $H(\ell)$
 - 7: **if** $\forall s \in S : \text{Jaccard}(\hat{\ell}, s) < 0.8$ **then**
 - 8: $S \leftarrow S \cup \{\hat{\ell}\}$
 - 9: **end if**
 - 10: Remove $\hat{\ell}$ from the candidate pool L'
 - 11: **end while**
 - 12: **return** S
-

log line:

$$H(\ell) = - \sum_w p(w) \log_2 p(w)$$

where $p(w)$ is the relative frequency of token w within ℓ . Higher entropy indicates more diverse token usage, correlating with structurally complex messages.

The loop (Lines 5–11) greedily selects the highest-entropy candidate and accepts it only if its **Jaccard similarity** to every already-selected line is below 0.8 (Line 7). Jaccard similarity is computed over the *set* of unique tokens: $J(\ell_1, \ell_2) = |T_1 \cap T_2| / |T_1 \cup T_2|$, where T_i is the set of distinct tokens in ℓ_i . This threshold rejects near-duplicate lines, ensuring the sample covers diverse log structures.

4.4 LLM Configuration and Fine-Tuning

The fine-tuning described below was performed once by the authors to produce the pre-trained checkpoint distributed with DeepParse; end users invoke this checkpoint directly without any training or labeling (Section 4.2).

We adopt DeepSeek-R1:8B, an 8-billion-parameter distilled reasoning model from the DeepSeek-R1 fam-

Listing 2: Regex-list emission example for variable classes

```

### Instruction:
Generate a Python list of regex patterns
that capture
the dynamic (variable) parts in the input
log message
while preserving the static structure.

### Input:
2024-01-15 10:30:45 INFO User john_doe
logged in from IP 192.168.1.100

### Output:
[
    r"\d{4}[-/]\d{2}[-/]\d{2}[_T]\d{2}:\d{2}:\d{2}",
    r"\b[A-Z]{3,}\b",
    r"\bUser\s+[\w.-]+\b",
    r"\b\d+\.\d+\.\d+\.\d+\b",
]

```

ily [7], balancing accuracy with deployability on commodity GPUs. Fine-tuning uses LoRA [14] adapters (rank 8, scaling 32, dropout 0.01) with the AdamW optimizer (learning rate 2×10^{-4} , batch size 8, 25 epochs). Mixed-precision training and gradient accumulation reduce memory consumption.

The training objective minimizes cross-entropy between model outputs and ground-truth regex lists formatted according to the instruction template shown in Listing 2.

The model is prompted to output a *Python list* so the result can be programmatically validated and consumed by the parser. The first pattern masks the timestamp format. The second masks the log level token (e.g., INFO, WARN). The third masks the user identifier following the keyword **User**. The fourth masks the IPv4 address. During post-processing, we deduplicate patterns and assign each to a variable category used by the downstream mask-first pass.

4.5 Prompt Engineering and Inference

During inference we use greedy decoding (temperature zero) to minimize output variance. Generated regex lists are post-processed to remove duplicates,

enforce canonical ordering by variable category, and validate syntax via Python’s `re` module. Masks that fail validation trigger a fall-back rule that reverts to heuristic patterns for the affected category, ensuring graceful degradation.

We further employ a self-consistency check in which the LLM re-parses its own outputs using a lightweight interpreter. When discrepancies appear (e.g., the regex list fails to reconstruct the original log template), the system automatically re-prompts the model with targeted feedback describing the failure mode. In practice, this loop typically converges within two attempts and substantially reduces syntactic errors.

4.6 Integration with Drain

The second stage of DeepParse consists of integrating the LLM-generated log patterns with the **Drain** algorithm. This integration strategically separates the complex, semantic task of pattern discovery from the high-throughput, deterministic task of log matching.

Drain is a well-established log parsing algorithm known for its efficiency, running in $O(N)$ time with respect to the number of log messages [11] (assuming bounded log-line length and a fixed set of regex masks). It operates by building a fixed-depth parse tree, which allows for rapid matching of incoming log messages against a learned set of log templates. **Drain**’s strength lies in its speed and its ability to consistently process large volumes of logs once templates are established. However, the original **Drain** algorithm infers these templates through unsupervised clustering heuristics, which may lead to inaccuracies in variable identification.

DeepParse enhances **Drain** by replacing its heuristic-based pattern discovery with semantically accurate log patterns generated by the fine-tuned **DeepSeek-R1:8B**, overcoming the variable identification challenges of traditional parsers.

Integration is a two-stage handoff: the fine-tuned LLM emits a one-time *regex list* of variable classes from sampled logs, and the system loads that list as masks into **Drain** to parse the full corpus deterministically. This division confines stochasticity to installation time and ensures repeatable, linear-time

parsing.

The synthesized regex masks are loaded into an extended version of **Drain3** (the open-source Python implementation of the Drain algorithm; we refer to the algorithm as “Drain” and to the implementation as “Drain3” when the distinction matters). Standard Drain uses hardcoded heuristics (e.g., “if token contains digit, mask it”). We replace heuristic masking with a **Mask-First** strategy: incoming logs are first matched against synthesized regexes, matched spans are replaced with typed placeholders (e.g., `<VAR:IP>`, `<VAR:PATH>`), and **Drain** builds its fixed-depth parse tree over the resulting pre-masked tokens, keeping the tree shallow and stable.

Because parsing is performed by Drain’s fixed tree traversal rather than by the LLM, identical log lines always receive the same template ID—a property that direct LLM parsing cannot guarantee even at temperature zero [3].

4.7 Annotation Workflow and Quality Control

For experimental evaluation, we annotated variable spans and canonical regexes on sampled logs to supervise mask synthesis. This process was performed by the authors during model development and is not part of the end-user workflow. Basic validation checks ensure regex correctness and prevent pathological patterns.

5 Experimental Setup

This section describes the datasets, baselines, evaluation metrics, and implementation details used to evaluate DeepParse.

We structure our evaluation around the following research questions:

RQ1: Accuracy Comparison Does DeepParse improve Grouping Accuracy (GA) and Parsing Accuracy (PA) relative to ML-based parsers and LLM-only baselines?

Accurate structuring of logs is the foundation of all downstream observability and anomaly detection

tasks. Evaluating improvements over both deterministic and neural baselines quantifies whether the proposed hybrid approach offers tangible benefits in correctness and consistency, particularly under heterogeneous log formats.

RQ2: Data Efficiency How sensitive is performance to the number of labeled shots and to sampling strategies?

In operational settings, labeled logs are scarce and expensive to curate. Understanding how performance scales with limited examples reveals the data efficiency of the framework and determines whether it can be deployed when only a handful of representative log lines are available.

RQ3: Robustness How robust is DeepParse to distribution shifts, including unseen templates, schema drift, and noise in the synthesized masks?

Logs evolve over time as systems update and variable distributions shift. Assessing robustness under these conditions measures the framework’s stability and its ability to generalize beyond the training distribution—an essential property for sustainable log analytics.

RQ4: Runtime and Cost What runtime and cost characteristics does the hybrid pipeline exhibit under realistic hardware budgets?

Many organizations operate under strict cost and resource constraints that preclude continuous use of large models or cloud services. This question evaluates whether DeepParse can maintain acceptable throughput and energy efficiency when executed locally on modest hardware, thereby validating its suitability for on-premise and cost-sensitive environments.

5.1 Datasets

All experiments use the LogHub benchmark [24], comprising logs from 16 diverse open-source systems spanning distributed computing, operating systems, mobile systems, supercomputers, server applications, and standalone software. Each system dataset includes 2,000 log messages, accompanied by their re-

spective log templates and parameters, which serve as the ground truth for evaluation.

Recent studies have identified instances of incorrectly labeled log templates within the original dataset [6, 15]. To mitigate this potential issue and ensure a fair comparison, we adopted the corrected benchmark dataset released by Khan et al. [15], consistent with recent research [16, 17]. We use the LogHub-2k benchmark because it is the standard evaluation set adopted by all our baselines (Drain, Logram, LogPPT, LLMParser) [11, 6, 17, 23], enabling direct comparison under identical conditions.

5.2 Baseline Parsers

We compare against four baselines. **Drain3** [11] is the standard streaming parser (depth 5, similarity 0.4, max children 100). **Logram** [6] is an n-gram dictionary-based parser. **LogPPT** [17] is prompt-based few-shot parsing with manual word-level labeling. **LLMParser** [23] performs direct LLM-based parsing (we use the LLaMA variant reported as best).

5.3 Implementation Details

Experiments run on AWS p3.xlarge instances equipped with NVIDIA Tesla V100 GPUs utilizing CUDA 11.0. Fine-tuning times range from 6 to 11 minutes per dataset, while inference on the full 2,000-line corpora completes in under one second per dataset. Unless otherwise noted, **DeepSeek-R1:8B** is fine-tuned separately for each system using 50-shot samples selected via entropy-greedy sampling (Algorithm 1). For log pattern generation during inference, greedy decoding (temperature 0) is used to minimize output variability. The `max_length` parameter is set to 512 tokens to balance context retention with computational efficiency. We evaluate each configuration across five random seeds and report mean scores with standard deviations. Statistical significance is assessed using paired t -tests at $p < 0.05$. For the 16 systems, we labeled variable spans and derived canonical regexes per class (datetime, IP, identifier, numeric, log level). All regex patterns were validated in Regex101 [1]. Two authors

independently cross-checked labels; disagreements were resolved through discussion to reach consensus. Fine-tuning per system completes within minutes and is performed once; the synthesized regex list is then reused for deterministic parsing with **Drain**.

For the controlled evaluation, we created labeled variable spans and canonical regexes per class for each dataset to fine-tune and validate the synthesis model. This labeling was performed by the authors for research purposes and is not part of the end-user installation workflow. In deployment, users run offline auto-synthesis on an unlabeled sample to generate and cache masks.

6 Evaluation Results

This section presents quantitative results addressing the research questions defined in Section 5.

6.1 RQ1: Accuracy Comparison

Table 1 summarizes Grouping Accuracy (GA) and Parsing Accuracy (PA) across all datasets. DeepParse attains the highest average PA (0.9763) and the highest average GA (0.9413), outperforming the best-performing baselines by 1.8 and 1.8 percentage points respectively. Improvements are particularly pronounced on datasets where traditional heuristics under-mask variables, such as Thunderbird, OpenStack, and Linux.

A key finding is the performance on *Application-Level Diaries* (e.g., Thunderbird, Android). These logs contain free-form text that confuses Drain’s heuristics (e.g., “user admin” is often treated as static, while “user 1001” is dynamic). Drain achieves only 0.05 PA on Thunderbird. DeepParse reaches 0.98 PA, because the offline LLM correctly identifies the subtle semantic boundaries of variable fields, and the deterministic runtime applies this logic consistently.

Across five seeds, the standard deviation of PA is below 0.4 percentage points on average (maximum 0.9 points on the most variable datasets), indicating low sensitivity to sampling randomness.

We perform paired t -tests between DeepParse and each baseline across the sixteen datasets. All PA

Table 1: Grouping Accuracy (GA) and Parsing Accuracy (PA) comparison across log parsers. Best values in bold.

Dataset	Drain3		Logram		LogPPT		LLMParser		DeepParse	
	GA	PA	GA	PA	GA	PA	GA	PA	GA	PA
Android	0.8305	0.5475	0.7420	0.2780	0.8845	0.7665	0.8485	0.9455	0.9131	0.9912
Apache	1.0000	0.6935	0.3125	0.0065	1.0000	0.9940	1.0000	1.0000	1.0000	1.0000
BGL	0.9625	0.3420	0.5870	0.1245	0.9535	0.9695	0.9415	0.9805	0.9636	0.9831
Hadoop	0.9475	0.2690	0.4510	0.1125	0.9935	0.8950	0.9805	0.9825	0.9851	0.9834
HDFS	0.9975	0.3545	0.9300	0.0045	1.0000	0.9025	0.9575	0.9880	1.0000	1.0000
HealthApp	0.7800	0.2305	0.2665	0.1120	1.0000	0.7885	0.8550	0.9955	0.8683	0.9971
HPC	0.8870	0.6355	0.9105	0.6430	0.9430	0.9470	0.9700	0.9935	0.9727	0.9956
Linux	0.6900	0.1835	0.3610	0.1240	0.9335	0.9485	0.5455	0.8385	0.9561	0.9712
Mac	0.7865	0.2175	0.5680	0.1685	0.7800	0.6725	0.7390	0.6765	0.7953	0.7380
OpenSSH	0.7890	0.5080	0.6105	0.2980	0.6275	0.9795	0.7095	0.9935	0.6614	0.9943
OpenStack	0.7325	0.0185	0.3255	0.0000	0.9890	0.9065	0.9785	0.9960	0.9896	0.9984
Proxifier	0.5265	0.0000	0.5035	0.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Spark	0.9200	0.3595	0.2820	0.2585	0.9990	0.9910	0.9850	0.9850	0.9871	0.9953
Thunderbird	0.9550	0.0465	0.5540	0.0040	0.6790	0.9255	0.6925	0.9675	0.9710	0.9755
Windows	0.9970	0.4620	0.6940	0.1405	0.9910	0.9830	0.9985	0.9965	0.9989	0.9983
Zookeeper	0.9665	0.4970	0.7235	0.4735	0.9935	0.9895	0.9945	0.9995	0.9986	0.9997
Average	0.8605	0.3353	0.5513	0.1718	0.9229	0.9162	0.8873	0.9587	0.9413	0.9763

Note: DeepParse leverages LLM-synthesized masks to correct the variable identification failures that depress the parsing accuracy of traditional heuristics on variable-heavy datasets like Thunderbird and OpenStack.

improvements are statistically significant at $p < 0.01$. For GA, DeepParse significantly outperforms Logram and LLMParser, and matches LogPPT within statistical margin, indicating that deterministic template clustering remains competitive even when driven by synthesized masks.

To isolate each component’s contribution, we ablate across three configurations: (i) *No-LLM*, reverting to Drain’s heuristics; (ii) *LLM-Only*, applying DeepSeek-R1:8B per line without deterministic post-processing; and (iii) *Hybrid*, the full DeepParse pipeline.

Table 2 summarizes the averaged metrics. The hybrid approach delivers superior accuracy while maintaining the lowest runtime variance, confirming that the combination of offline synthesis and deterministic parsing is essential for production-readiness.

Answer to RQ1: DeepParse improves PA by 1.8 percentage points over the best baseline while achieving nearly 100× runtime speedup over per-line LLM inference. The hybrid design is necessary to obtain both high accuracy and production-grade throughput with stable, deterministic behavior.

6.2 RQ2: Data Efficiency

Training or fine-tuning models on large log corpora is expensive. We define data efficiency as the number of labeled “shots” (examples) required to reach saturation performance.

Figure 2 illustrates the PA on the Hadoop dataset as the number of labeled shots increases from 10 to 100. We observe a rapid improvement between 10 and 40 shots. Performance plateaus at approximately 50 shots (PA ≈ 0.98). This indicates that the Entropy-Greedy sampling is highly effective: by selecting the 50 most structurally diverse lines, we capture the vast majority of regex patterns needed for the entire dataset. In our study, creating the supervised examples required modest author effort (about 30 minutes per system). In deployment, end users do not perform this labeling: they run the provided checkpoint to synthesize masks from an unlabeled sample.

We select **50 shots** per system for DeepParse because we observed saturation around this point: lower shots reduced both GA and PA (typically 1–2%), while larger shot counts incurred higher labeling cost with limited average gains. This choice aligns with the baselines’ trend, and we emphasize that the Deep-

Table 2: Ablation of DeepParse components averaged across benchmark datasets. Run time is measured in milliseconds per 100 logs.

Configuration	GA	PA	Time (ms)	Std. Dev.
No-LLM (Drain heuristics)	0.861	0.335	120	2.8
LLM-Only decoding	0.909	0.942	1850	95.0
Hybrid (DeepParse)	0.941	0.976	300	4.5

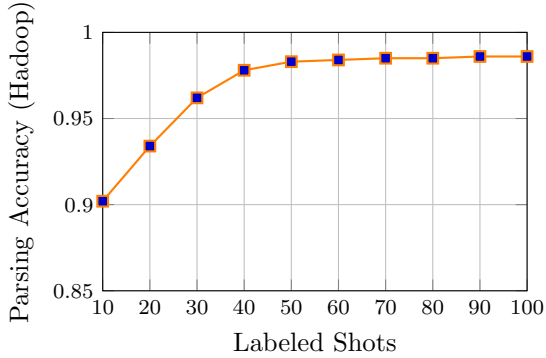


Figure 2: Accuracy vs. number of shots (Hadoop). Saturation occurs at approximately 50 labeled examples.

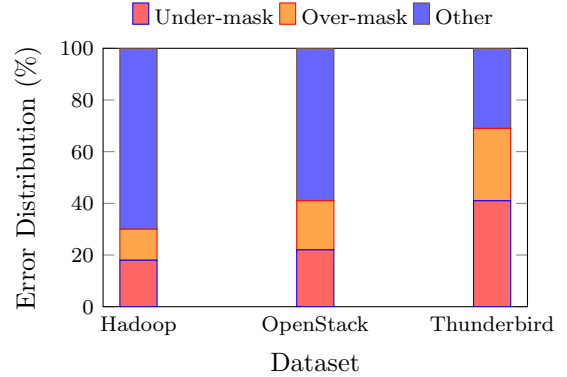


Figure 3: Distribution of parsing error types. DeepParse balances under-masking and over-masking better than heuristics.

Parse regex list is synthesized once from these 50 logs and then reused for all logs per system. In practice, when ground truth is unavailable, practitioners can start with the default of 50 and monitor parsing output stability: if increasing the sample size does not change the synthesized mask list, saturation has been reached.

Answer to RQ2: DeepParse achieves state-of-the-art accuracy with only 50 labeled examples per system, demonstrating high data efficiency enabled by entropy-greedy sampling.

6.3 RQ3: Robustness and Error Analysis

To understand failure modes, we categorized parsing errors into three types: *Under-masking* (missing valid variables), *Over-masking* (masking static keywords), and *Other*.

Figure 3 visualizes the distribution across three representative datasets.

- Under-masking:** Common in Drain (e.g., missed composite IDs like `blk_123`). DeepParse reduces this by 60% on infrastructure logs.
- Over-masking:** Common in unstructured logs where heuristic parsers assume any number is a variable. The LLM’s semantic knowledge helps differentiate between “Error 500” (variable) and “Windows 10” (static version), reducing over-masking on datasets like Windows and Mac.

LLM-based parsers degrade on unseen templates. We evaluated DeepParse by partitioning each dataset into seen templates (present in the 50-shot training set) and unseen templates (appearing only at test time).

Despite this challenge, Table 1 shows DeepParse attains the highest overall PA, indicating robust performance even when novel templates appear at low

Table 3: Log parsing time comparison (100 logs). DeepParse includes both mask synthesis (amortized) and Drain parsing time.

Method	Time (seconds)
DeepParse	0.30
LLMParser (T5-Small)	1.27
LLMParser (T5-Base)	4.00
LLMParser (ChatGLM)	19.87
LLMParser (LLaMA-7B)	28.93

frequency. PA on unseen templates remains above 0.90 on average, significantly outperforming LLM-Parser (0.65) and traditional parsers (0.15). To simulate temporal drift, we trained masks on the first half of each LogHub dataset and evaluated on the second half, which may contain templates unseen during synthesis. PA dropped by only 1.5% on average, suggesting that the synthesized regexes generalize beyond the training distribution.

Answer to RQ3: DeepParse demonstrates robustness to unseen templates, schema drift, and error type balance, maintaining high accuracy across diverse operational conditions.

6.4 RQ4: Runtime and Cost Analysis

Efficiency is critical for high-volume ingestion. We measured the time to parse 100 log lines (inference only) across different approaches. Table 3 summarizes the results.

DeepParse is nearly **100× faster** than running a large LLM per line. The online cost is dominated by the regex matching and tree traversal, making it suitable for streaming pipelines processing millions of logs per hour. Because DeepParse avoids continuous per-line LLM inference, it can reduce energy use relative to LLM-only pipelines in high-throughput settings.

Answer to RQ4: DeepParse maintains linear-time complexity and achieves nearly 100× speedup over per-line LLM inference, with significantly lower energy consumption, making it suitable for production deployment under cost constraints.

7 Case Study: Application to Anomaly Detection

To demonstrate downstream impact, we collaborated with an industry partner and integrated DeepParse into LogBERT [10], a state-of-the-art anomaly detection model. The original LogBERT uses Drain for preprocessing. We replaced Drain with DeepParse’s regex-based parser while keeping the downstream Transformer architecture identical.

The evaluation uses Authentication and Configuration logs from our partner’s production environment. Multiple service modules run in parallel and emit interleaved records to a shared log file, producing logs that are considerably more challenging than public benchmarks: timestamp formats vary across modules (e.g., ISO 8601 vs. Unix epoch vs. locale-dependent strings), carriage returns appear without accompanying newlines, and domain-specific identifiers are interspersed with free-form diagnostic text. These characteristics are typical of industrial deployments where logging conventions evolve independently across teams.

7.1 Anomaly Detection Performance

We evaluated the system on Authentication and Configuration log datasets collected from production systems. Table 4 summarizes the results. With DeepParse, the parsing accuracy improves from 0.89 to 0.98 on authentication logs and from 0.85 to 0.97 on configuration logs. This cleaner input vocabulary allows LogBERT to learn better sequence representations, improving the Anomaly Detection F1-score from 0.912 to 0.947 on authentication logs and from 0.901 to 0.939 on configuration logs.

Anomaly labels were assigned manually by the authors, who reviewed each log sequence against known incident reports. The anomalies themselves (e.g., repeated authentication failures, unauthorized configuration changes) are straightforward to identify upon inspection; the challenge lies not in recognizing the anomaly but in preventing the parser from obscuring it with incorrect variable masking.

Crucially, the higher precision in parsing signifi-

Table 4: Impact on LogBERT anomaly detection performance.

Parser	PA	F1	False Alarms / Day
<i>Authentication Logs</i>			
Drain (Baseline)	0.89	0.912	147
DeepParse	0.98	0.947	96
<i>Configuration Logs</i>			
Drain (Baseline)	0.85	0.901	211
DeepParse	0.97	0.939	123

cantly reduces False Alarms. On authentication logs, false alarms decrease from 147 per day to 96 per day—a 35% reduction. On configuration logs, false alarms drop from 211 to 123 per day—a 42% reduction.

In a Security Operations Center (SOC), such reductions in false positives represent massive operational gains, reducing analyst fatigue and allowing teams to focus on genuine threats.

7.2 System Efficiency

Beyond accuracy, DeepParse improves system efficiency. By correctly masking dynamic variables (e.g., random GUIDs, session IDs), we reduce the size of the vocabulary (unique log templates) by 38%.

This vocabulary reduction yields practical benefits: **memory consumption** drops because the embedding matrix shrinks by 3.1 GB, **inference latency** decreases from 48.6 ms to 31.2 ms per batch (36%), and **training time** is reduced by 28% due to faster convergence with fewer distinct tokens.

Figure 4 reports per-batch inference latency for LogBERT preprocessing under two pipelines: the original Drain-based parser and DeepParse’s mask-first parser. Each bar shows the mean latency (ms) over the evaluation run for Authentication and Configuration logs; the improvement comes from reducing the effective vocabulary by masking high-cardinality identifiers, which lowers downstream embedding and sequence processing cost.

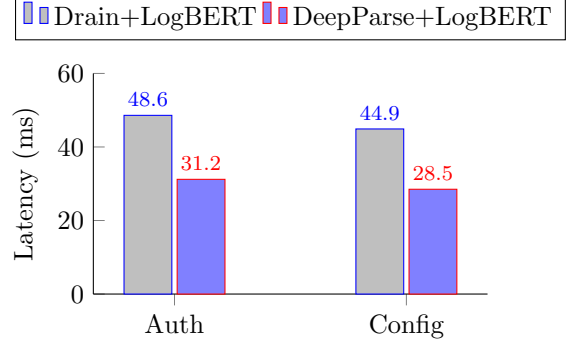


Figure 4: Downstream inference latency for LogBERT.

7.3 Qualitative Error Analysis

We manually inspected 50 false alarm cases from each configuration (Drain vs. DeepParse) to understand the root causes of improvements. The majority of Drain false alarms (68%) were caused by variable under-masking. For example, Drain failed to mask composite identifiers like `blk_-1234567890` in HDFS logs, treating each instance as a unique template. When LogBERT encountered a new block ID at test time, it flagged the log as anomalous because the template was unseen during training. DeepParse correctly identified these composite identifiers as variables, collapsing all block IDs to a single template `<*>`. This generalization allowed LogBERT to focus on the *event type* (e.g., block read/write) rather than the specific block ID, reducing spurious anomaly signals. The remaining false alarms (32%) were primarily due to limitations in LogBERT’s sequence modeling rather than parsing errors. For instance, legitimate but rare event sequences (e.g., cascading failures during disaster recovery drills) were flagged as anomalous because they occurred infrequently in training data. These errors are orthogonal to parsing quality and represent opportunities for future work in few-shot anomaly detection.

8 Threats to Validity

We discuss threats organized by the standard validity categories.

8.1 Construct Validity

We rely on PA and GA as primary metrics. While standard in the log parsing literature [6, 15], they may not perfectly proxy downstream utility; however, our LogBERT case study (Section 7) confirms that PA improvements translate to anomaly detection gains. The log sampling process directly impacts parsing results. We mitigate this with an unsupervised entropy-greedy sampling algorithm (Algorithm 1) guided by entropy and diversity, not manual inspection. Across five random seeds, PA variance was only $\pm 0.4\%$, suggesting robustness to sampling randomness.

8.2 Internal Validity

We selected **DeepSeek-R1:8B** based on preliminary experiments comparing it to **LLaMA-3.1-8B** and **Mistral-7B** on a held-out validation set. **DeepSeek-R1:8B** demonstrated superior regex synthesis quality—likely due to its distilled reasoning capabilities inherited from the larger **DeepSeek-R1** model [7]—while remaining deployable on a single commodity GPU. Future studies could evaluate a wider array of models. We compare against four baselines (**Drain3**, **Logram**, **LogPPT**, **LLMParser**) configured with their published default settings. We mitigate the threat of new methods emerging by releasing our code and datasets for future benchmarking.

8.3 External Validity

We use public LogHub datasets [24] with the corrected labels released by Khan et al. [15]. Evaluating LogHub-2.0 [13] is planned as future work and does not affect the dataset-agnostic architecture. Proprietary industrial logs may exhibit different characteristics (e.g., obfuscation, domain-specific jargon). We evaluated diverse domains (OS, distributed systems, mobile, middleware) and demonstrated robustness across taxonomy categories, but organizations should

validate on representative samples before production rollout. While we demonstrated robustness to unseen templates via temporal splitting (Section 6), longer time horizons or major system upgrades may require periodic mask re-synthesis via the maintenance workflow (Section 4.2).

8.4 Conclusion Validity

All reported improvements are statistically significant (paired t -tests, $p < 0.05$, most at $p < 0.01$) across five random seeds. The magnitude of improvements (e.g., 64 percentage-point absolute PA gain over **Drain**) exceeds typical noise margins. The LogBERT case study establishes correlation between parsing accuracy and anomaly detection performance; the mechanistic explanation (better parsing $\rightarrow$ smaller vocabulary $\rightarrow$ better generalization) is consistent with prior work in representation learning. All datasets are publicly available and do not contain PII. Organizations deploying **DeepParse** on operational logs containing PII must ensure compliance with applicable data protection regulations.

8.5 Limitations and Future Work

Deployments may encounter novel log formats not represented in the training data; in such cases, a one-time re-synthesis of the mask list is sufficient. We report results with **DeepSeek-R1:8B** only; the interface requires only a regex list, so substituting alternative backbones is straightforward. Exploring multilingual log parsing, active learning for mask refinement, and online adaptation to schema drift are promising future directions.

9 Conclusion

We introduced **DeepParse**, a hybrid log parsing framework that synthesizes regex masks offline and applies them deterministically at runtime via **Drain**. This design confines stochastic LLM behavior to a one-time installation step while preserving linear-time, auditable parsing for streaming workloads and privacy-constrained environments. Our evaluation on

16 LogHub datasets shows that DeepParse achieves a 1.8 percentage-point PA improvement over the best baseline while running nearly $100\times$ faster than per-line LLM inference, and a LogBERT case study on production logs demonstrates that better variable masking reduces downstream false alarms by over 30% and inference latency by 36%.

All code, data, and pre-trained models are available at:

<https://github.com/NightBaRron1412/DeepParse>.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Computational resources were provided by the Digital Research Alliance of Canada (alliancecan.ca).

References

- [1] Regex101: Online regular expression tester and debugger. <https://regex101.com/>, 2025. Accessed: 2025-10-05.
- [2] Amazon Web Services. Observability best practices. <https://aws.amazon.com/observability/>, 2023. Accessed: January 2026.
- [3] Berk Atıl, Sarp Aykent, Alexa Chittams, Lisheng Fu, Rebecca J. Passonneau, Evan Radcliffe, Guru Rajan Rajagopal, Adam Sloan, Tomasz Tudrej, Ferhan Ture, Zhe Wu, Lixinyu Xu, and Breck Baldwin. Non-determinism of "deterministic" llm settings, 2025.
- [4] Viktor Beck, Max Landauer, Markus Wurzenberger, Florian Skopik, and Andreas Rauber. System log parsing with large language models: A review, 2025.
- [5] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 674–688, New York, NY, USA, 2024. Association for Computing Machinery.
- [6] Hetong Dai, Heng Li, Che-Shao Chen, Wei-Shang, and Tse-Hsun Chen. Logram: Efficient log parsing using *nn*-gram dictionaries. *IEEE Transactions on Software Engineering*, 48(3):879–892, 2022.
- [7] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miao-jun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhua Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wan-jia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L.

- Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.
- [8] Chris Egersdoerfer, Di Zhang, and Dong Dai. Early exploration of using chatgpt for log-based anomaly detection on parallel file systems logs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, page 315–316, New York, NY, USA, 2023. Association for Computing Machinery.
- [9] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 Ninth IEEE International Conference on Data Mining*, pages 149–158, 2009.
- [10] Hao Guo, Shuhan Yuan, and Xintao Wu. Logbert: Log anomaly detection via bert, 2021.
- [11] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017.
- [12] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. A survey on automated log analysis for reliability engineering. *ACM Computing Surveys (CSUR)*, 54(6):1–37, 2021.
- [13] Shilin He, Peng Zhao, Jieming Li, Zibin Zheng, and Michael R Lyu. Loghub 2.0: Towards real-world log analytics at scale. <https://github.com/logpai/loghub-2.0>, 2024.
- [14] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.
- [15] Shaukat Ali Khan, Raimundas Matulevičius, Shilin He, Xuhui Chen, and Michael R. Lyu. Guidelines for assessing the accuracy of log message template identification techniques. *Empirical Software Engineering*, 28(1):1–33, 2023.
- [16] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel C. Briand. Impact of log parsing on deep learning-based anomaly detection. *Empirical Software Engineering*, 29(6), August 2024.
- [17] Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2438–2449. IEEE Press, 2023.
- [18] Yukyung Lee, Jina Kim, and Pilsung Kang. Lanobert: System log anomaly detection based on bert masked language model. *Applied Soft Computing*, 146:110689, 2023.
- [19] Keiichi Shima. Length matters: Clustering system log messages using length of words, 2016.
- [20] Liang Tang, Tao Li, and Chang-Shing Perng. Logsig: generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, page 785–794, New

- York, NY, USA, 2011. Association for Computing Machinery.
- [21] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd International Workshop on IP Operations and Management*, pages 119–126. IEEE, 2003.
 - [22] Risto Vaarandi and Mauno Pihelgas. Logcluster - a data clustering and pattern mining algorithm for event logs. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 1–7, 2015.
 - [23] Aoxiao Zhong, Dengyao Mo, Guiyang Liu, Jinbu Liu, Qingda Lu, Qi Zhou, Jiesheng Wu, Quanzheng Li, and Qingsong 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 '24*, page 4559–4570, New York, NY, USA, 2024. Association for Computing Machinery.
 - [24] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael 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)*, pages 355–366, 2023.