InferLog: Accelerating LLM Inference for Online Log Parsing via ICL-oriented Prefix Caching

Yilun Wang
Sun Yat-sen University, China
wangylun6@mail2.sysu.edu.cn

Pengfei Chen*
Sun Yat-sen University, China
chenpf7@mail.sysu.edu.cn

Haiyu Huang, Zilong He, Gou Tan
Sun Yat-sen University, China
{huanghy95,hezlong,tang29}@mail2.sysu.edu.cn

Chuanfu Zhang
Sun Yat-sen University, China
zhangchf9@mail.sysu.edu.cn

Jingkai He
Sun Yat-sen University, China
hejk25@mail2.sysu.edu.cn

Zibin Zheng
Sun Yat-sen University, China
zhzibin@mail.sysu.edu.cn

ABSTRACT

Modern software systems generate massive volumes of runtime logs, necessitating efficient and accurate log parsing to enable critical downstream tasks such as anomaly detection and root cause analysis. Recently, large language models (LLMs) have achieved advanced accuracy on log parsing, but their deployment in production environments faces two major limitations: (i) the privacy risks associated with commercial LLMs, driving the adoption of local deployment, and (ii) the stringent latency and throughput requirements imposed by high-volume log streams, which existing LLM-based parsers fail to meet. Although recent efforts have reduced the number of LLM queries, they overlook the high latency of the LLM invocations, where concurrent log parsing requests can cause serve performance degradation of LLM inference system.

In this study, we present *InferLog*, the first LLM inference optimization method for online log parsing. Our key insight is that the inference efficiency emerges as the vital bottleneck in LLM-based online log parsing, rather than parsing accuracy. *InferLog* accelerates inference by designing (i) A Prefix-aware ICL Refinement policy to refine the examples and permutation of in-context learning to improve the prefix caching efficiency. (ii) A rapid and task-specific configuration tuning pipeline based on meta-learning to find the optimal LLM scheduling-related configuration for dynamic log parsing workloads. The experimental results based on Loghub dataset and vLLM demonstrate that *InferLog* significantly outperforms existing inference optimization methods and markedly accelerates the state-of-the-art LLM-based log parser without compromising parsing accuracy.

KEYWORDS

Log Parsing, Large Language Model, Prefix Caching, Configuration Tuning

1 INTRODUCTION

Modern software systems generate massive volumes of runtime logs to record system states, events, and anomalies. Automatically parsing these semi-structured logs into structured templates, a process known as log parsing, is a foundational task in software engineering. Specifically, this task requires separating log messages into two distinct components: *static templates* - constant patterns explicitly defined in logging statement and *dynamic variables* - runtime-specific values reflecting system states. It enables critical downstream applications such as anomaly detection, fault localization and root cause analysis, as shown in Fig.1.

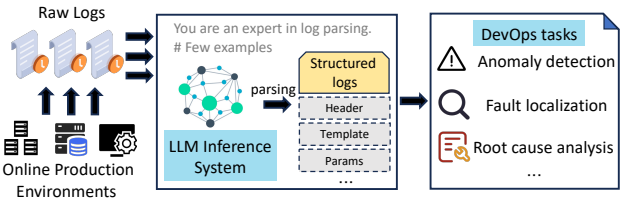


Figure 1. LLM-based online log parsing system.

In recent years, large language models (LLMs) have revolutionized log parsing by leveraging their semantic understanding and few-shot learning capabilities[23, 37, 38, 55, 60, 67]. Current LLM-based parsers achieve state-of-the-art (SOTA) accuracy in template extraction, even for unseen log formats. This advancement stands in contrast to traditional parsing methods, which typically rely on manually designed heuristics and syntactic analysis[11, 18, 24, 28, 40, 46, 62]. Nevertheless, a critical gap persists between existing methods and modern software system deployment: First, commercial LLMs pose significant privacy risks for log parsing due to potential unauthorized data retention and exposure vulnerabilities[38]. Consequently, organizations increasingly adopt locally deployable open-source LLMs to ensure confidential log data remains secure under strict compliance standards. Second, a modern software system can produce log data at rates of several GB per second, imposing stringent requirements on processing *latency* and *throughput*[19, 53]. Despite recent works have achieved promising parsing accuracy, the inference performance is far from the Service Level Objectives (SLOs) in production environments. In our experiments using the vLLM system, we observed that the p95 inference latency for concurrent Loghub-2k log parsing requests was in the tens of seconds, it leads to cascading delays in downstream tasks, ultimately compromising the reliability of large-scale software systems. So, it is crucial to develop an efficient local LLM inference system to handle the massive log parsing requests and meet the SLOs.

Recent works[6, 25, 66] have highlighted the use of *prefix caching* (PC) technique to enhance the performance of LLM inference systems. By sharing and reusing intermediate results (i.e., KV Cache) across requests, it effectively accelerates time-to-first-token (TTFT) and reduces computation costs. While PC benefits general LLM workloads, the effectiveness in log parsing tasks remains unexplored. We conduct an empirical study of LLM-based online log

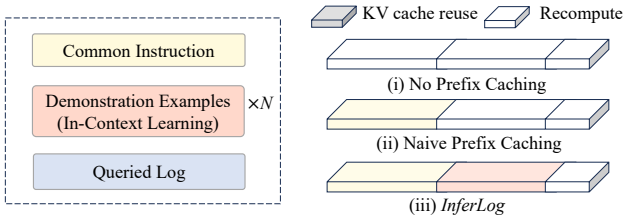
parsing workloads under concurrent scenarios on vLLM[27], three critical performance issues were identified: (i) The *prefill phase* (processing input sequence before generating outputs) dominates inference latency, causing 85.4% of delays distribution, violating production SLOs. (ii) Traditional prefix caching are ineffective in bypassing computations for tokens in in-context learning (ICL) examples, which make up a significant portion of the context in the prompt. (iii) LLM system parameters, such as maximum token and batch thresholds, significantly impact inference performance, through automated configuration tuning, it can lead to substantial reductions in latency compared with default setting. A detailed analysis can be found in § 2.2.

Generally, There are mainly two challenges in designing the inference optimization techniques for log parsing workloads. ***C1: Prefix caching imposes strict requirements on same prefix tokens***, as the KV cache contains unique positional embeddings within the original statement and the dependencies between the tokens of the sentence. However, log parsing prompts are dynamic across requests, characterized by varying in-context learning (ICL) examples and permutation[60], which often leads to prefix cache misses. Existing methods[35, 66] attempt to mitigate this by reordering requests to group those with identical prefixes for batch processing, however, these approach is not suitable for online requests. ***C2: Rapid configuration adaptation for dynamic log parsing workloads***. Logs generated by different software systems exhibit distinct characteristics and token distributions, necessitating tailored configurations for optimal performance. Existing tuning methods such as Bayesian Optimization[3, 7, 64] and Reinforcement Learning[9, 30, 63] demand extensive online trials. This renders them impractical for online inference tasks that demand swift configuration tuning[69].

InferLog Approach. Our key insight is that the inference efficiency emerges as the vital bottleneck in LLM-based online log parsing, which is ignored in previous works[20, 23, 37, 55, 57, 60]. To this end, we propose *InferLog*, a novel framework designed to optimize the LLM ***Infer****ence* performance for ***Log*** parsing tasks, focusing on reducing latency and improving throughput in online environments. *InferLog* proposes *Prefix-aware ICL Refinement*(PAIR) policy to refine the log parsing prompt by adjusting the contents and order of ICL examples. Specifically, for each current request, *InferLog* first identifies the examples of historical requests with the highest prefix cache hit rate probability, then performs modifying and reordering operations to update the current examples to align with it. By doing so, the ICL component can significantly boost the prefix cache hit rate(**solution to C1**). To achieve a fast and tailored configuration optimization, we introduce *Attention mechinism*[49] in *Model-Agnostic Meta-Learning* (MAML)[14] to identify optimal initial parameters of the model for the target tuning tasks and then adopt *Sequential Model-based Optimization* (SMBO)[13] to update the meta-model through few-shot learning with newly collected data to rapidly recommend best configuration in a few iterations(**solution to C2**).

Generally, we make the following contributions.

- We propose *InferLog*, the first LLM inference acceleration framework for online LLM-based log parsing, which boosts LLM inference by *prefix caching* and *configuration tuning*.



(a) Prompt Structure **(b)** Comparison of Prefix Caching

Figure 2. (a) The prompt structure of log parsing and (b) prefix caching for prompt, colored cubes represent KV cache reuse, colorless cubes represent recompute. *InferLog* explores the prefix cache hit rate improvement of ICL part, which constitute a significant portion of prompt tokens.

- We identify the inefficiency of the ICL prefix cache reuse in LLM-based log parsing. To address this issue, we refine ICL demonstrations by matching, modifying and reordering to improve the prefix caching hit rate.
- We introduce a fast and efficient configuration tuning pipeline that leverages attention-based MAML and SMBO to optimize LLM inference system performance.
- We implement *InferLog* based on vLLM. The evaluations of *InferLog* on the Loghub-2k dataset, demonstrate that *InferLog* significantly reduces the p95 latency by 43.02% and increases throughput by 2.14× compared to other inference acceleration baselines and is compatible to current LLM-based log parsers.

2 BACKGROUND AND MOTIVATIONS

2.1 Background

2.1.1 Log Parsing. Log parsing is the process of transforming semi-structured log entries into structured data by extracting both the static elements (i.e., log templates) and the variable components (i.e., log parameters) from the messages. Existing log parsers can be categorized into two main types: syntax-based and semantic-based. Syntax-based log parsers[11, 18, 24, 40, 62] typically employ manually designed heuristics or analyze syntactic characteristics to extract log templates. However, their accuracy tends to decline when dealing with logs that do not adhere to established formats. In contrast, semantic-based log parsers[20, 23, 28, 36, 37, 57, 60] utilize neural networks or language models to discern log templates and parameters by understanding the underlying meanings of log messages. For example, LogPPT[28] utilizes a RoBERTa model for identifying log templates and parameters through prompt-based few-shot learning. Recently, the emergence of large language models has led to the development of various LLM-based log parsers that offer enhanced log parsing capabilities and demonstrate strong adaptability to various log formats. These LLM-based parsers employ techniques such as *fine-tuning*[37], *prompt engineering*[23, 55, 60, 67] to optimize LLMs specifically for log parsing tasks, resulting in impressive performance.

ICL-based Prompt Engineering. *Fine-tuning* LLMs demands substantial computational resources and high-quality labeled data. Also, the high update frequency of software systems leads to frequent changes in log templates[53], making fine-tuning less viable. *In-context learning*[8], conversely, has emerged as a popular method for leveraging LLMs in downstream tasks without

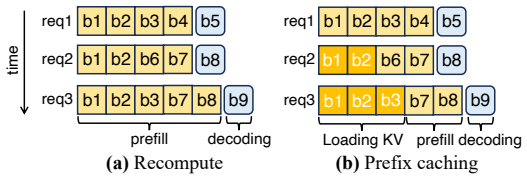


Figure 3. The comparison of recompute and PC.

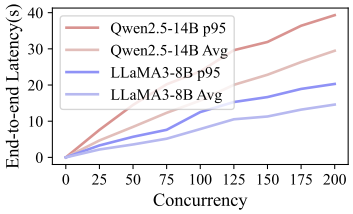


Figure 4. Inference time under different concurrent workloads.

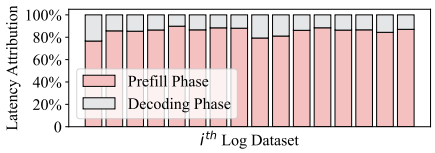


Figure 5. LLM inference latency attribution.

finetuning[23, 26, 31, 59, 60, 67], it makes predictions based on contexts augmented with a few examples, where the selection of examples is based on the similarity to the target query. This enables LLMs to adapt quickly to new log formats and templates and reduces the dependency on training data. Every example consisted of a labeled {*Log*, *Template*} pair. For individual log parsing, the following prompt format has been widely adopted: *Common Instruction*: ⟨...⟩, *Demonstration Examples*: ⟨...⟩, *Queried Log*: ⟨...⟩, which as shown in Fig.2(a).

2.1.2 LLM Inference System. The inference of LLMs is usually divided into prefill and decode phases. The prefill phases involves feeding all prompt tokens into the model in a single, parallel forward pass, generating the initial output tokens. In contrast, the decode process incrementally generates subsequent tokens based on the previously generated tokens and the input context.

KV Cache and Prefix Caching. The prefill stage is particularly time-consuming, as it requires to compute the entire input sequence's key (K) and value (V) tensors. Due to the autoregressive nature of model generation, each generated token depends solely on the previously generated tokens. The KV tensors computed by the attention layers are cached, and subsequent tokens use only the last generated token and the KV cache as inputs for the model's forward pass. Prefix caching (PC)[6] is an efficient technique for cross-request KV cache reuse by allowing the common KV context of prompt to be computed once and reused across different queries that share the same prefix. Specifically, if a new coming query shares the same prefix as existing one, the corresponding KV cache can be reused, allowing the new query to bypass computation for the shared tokens. Due to memory capacity, KV cache blocks are evicted based on an least recently used (LRU) policy.

Despite these advantages, PC imposes strict requirements on same prefix tokens, underscoring the order-dependence of KV tensors. Take Fig.3 as an example, three requests arrive in order, `req2` and `req3` have some same prefix as `req1` so they can load KV cache without recomputation(i.e., `[b1,b2]` in `req2` and `[b1,b2,b3]` in `req3`), but even if `req3` and `req2` have the same logical block `[b7,b8]` in the same position, `req3` can not reuse `req2`'s KV cache of `[b7,b8]` due to missing prefix block `b6`. Without a rational management of requests can lead a poor KV cache hit[25, 66].

LLM System Parameters. The presence of numerous configuration parameters offers flexibility for optimizing performance in inference engines[51]. For example, in vLLM[27], `max-num-seqs` and `max-num-batched-tokens` defines the maximum number of requests and tokens the model can process simultaneously in a single inference step. Besides, `enable-prefix-caching` and `enable-chunked-prefill` can be employed to activate respective features, which can provide significant advantages in certain scenarios. Recent study[7] proposes optimizing parameters for vLLM engine using BO, however, this approach is inefficient for online log parsing task due to the large search space and diverse token distributions. Besides, it requires much online tuning steps, and can not easily adapt to diverse workloads. *InferLog* targets at the optimization of *scheduling-related* parameters, which play a critical role in request scheduling and resource utilization optimization in the environments with dynamic high-concurrency requests.

2.2 Motivations

In this section, we show our empirical studies on LLM inference performance on online log parsing and introduce our motivations. The studies in conducted on the public dataset from Loghub-2k[68] under vLLM[27] inference system. Detailed information of experimental setup is provided in § 4.1. Our study aims to answer the following research questions (RQs):

- **RQ1**: Does the LLM-based online log parsing workloads meet the requirements of SLOs under concurrent requests?
- **RQ2**: Does the traditional prefix caching technique achieves optimal KV cache reuse for log parsing workloads?
- **RQ3**: Is the default configuration setting of LLM inference system suitable for log parsing workloads?

2.2.1 LLM Inference Performance for Online Log Parsing. With the increasing complexity of software technology and distributed systems, production systems generate massive amounts of logs daily[19]. Ensuring low latency to meet SLOs for online inference is crucial for downstream tasks. However, the high volume of concurrent invocations presents significant challenges to LLM inference performance. As depicted in Fig.4, performance deteriorates significantly with an increase in concurrent requests. The average and p95 latency at a concurrency of 200 are 6.29× and 5.17× higher than at a concurrency of 25 in `Qwen2.5-14B`, and 6.70× and 6.15× in `LLaMA3-8B`, even reaching several tens of seconds. Moreover, we observe that the inference latency is predominantly attributed to the prefill phase, which accounts for an average of 85.4% of the latency across 16 datasets in Fig.5. The reason is that log parsing task involves complete instruction and ICL demonstrations in prompt, leading to the token length far exceeding the decode length, where decoding stage only outputs static log templates. So there is a pressing need to reduce prefill latency to enhance inference performance for online log parsing.

Motivation1: Massive concurrent invocations in LLM-based online log parsing scenario pose a substantial challenge to LLM inference performance, with the primary performance bottleneck occurring during the prefill stage.

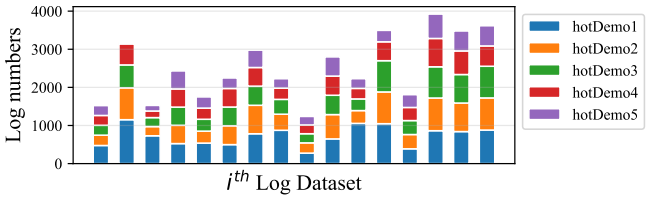


Figure 6. The number of queried logs that select top 5 hotspot demonstrations for ICL.

2.2.2 Insufficient Prefix KV Cache Reuse. Traditional prefix caching technique, as depicted in Fig.2(b)(ii), optimize LLM inference by reusing KV cache for static prompt components like common instructions. However, PC can not efficiently reuse KV cache for ICL part due to two reasons: (i) ***Dynamic examples selection***. ICL examples are chosen based on k-Nearest Neighbor (kNN). For instance, LILAC[23] adopts Jaccard similarity and DivLog[60] uses cosine distance to calculate the similarity between demonstration candidates and target log, leading to example sequence variations of ICL across requests. (ii) ***Specific permutation***. Existing methods[23, 60] typically order the demonstrations based on their ascending similarity to the queried log, positioning the most similar demonstrations closer to it. This leads to the situation where, even if different log messages select the same examples, their order within the entire demonstration set will vary. Overall, these common practices in LLM-based log parsing impose token sequence mismatches, rendering traditional prefix caching ineffective for ICL tokens. Our experiments across 16 datasets show that for naive PC, the average prefix cache hit rate[1] is 55.17% when parsing distinct log messages.

However, although log messages are diverse, we identify the selection of ICL examples across different requests exhibits a notable degree of overlap, creating an opportunity for ICL to reuse historical KV cache. Specifically, (i) ***The candidate numbers for ICL demonstrations is limited***. For instance, prior studies[23, 67] constrain the number of candidates to a fixed small size. (ii) ***The phenomenon of hotspot demonstrations***. We find that specific templates from the candidate set are frequently selected during the examples selection process. As illustrated in Fig. 6, the top five hotspot demonstration templates were selected by average 36.2%, 27.2%, 24.7%, 21.2%, and 17.1% in each dataset[2], respectively. For instance, across 16 datasets, an average of $2000 \times 36.2\% = 724$ independent log messages chose *hotDemo1* as their demonstration template.

Motivation2: Traditional prefix caching can not efficiently reuse prefix KV cache for ICL due to dynamic examples selection and specific permutation. The presence of hot demonstrations suggests that refining the ICL examples could enable it to benefit from prefix caching.

2.2.3 Suboptimal LLM Inference Parameter Configurations Setting and Inefficient Tuning Process. Configuration parameters of LLM engines significantly impacts the inference performance, particularly for *scheduling-related* parameters, which control the maximum batch token size and scheduling intervals, thereby determining the

[1]The metric of vllm:gpu_prefix_cache_hit_rate exposed by vllm.
[2]The x-axis ticks of Figure 5 and Figure 6 are the same as Figure 11.

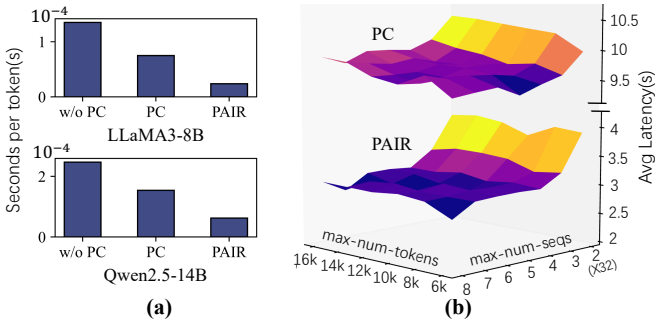


Figure 7. (a) shows the performance of different prefix caching strategies. (b) shows the performance surface of two parameters under PC and PAIR. The modification of PC strategy leads to changes in the configuration performance.

scheduling strategy. It is necessary to perform configuration optimization due to the following reasons. (i) ***Default parameters underperform for specific workloads***. Default parameter settings are typically designed for general use, offering a one-size-fits-all approach that may not be optimal for specific tasks. For instance, the default `max-num-batched-tokens` is often set in line with the `max-position-embeddings` of LLM, yet this setting is usually excessive for medium or short context tasks. Our experiments in § 5.4.2 show optimized configurations can cut end-to-end latency by an average of 34.91%, and up to 57.6% compared to default settings. (ii) ***Impact of PC strategy on configuration governance***. Under identical configurations, diverse PC strategies demonstrate distinct performance characteristics, with PAIR surpassing PC by an average of ~6 seconds latency as illustrated in Fig.7(b). In particular, during the prefill phase, PAIR lowers average per-token processing time due to the improved KV cache reuse. This reduction suggests permitting more simultaneous requests or tokens to enhance GPU utilization without additional latency. Besides, the optimal configuration setting for PC are suboptimal within the context of PAIR.

Existing methods employ online learning techniques such as BO and RL to recommend configurations[3, 9, 63]. However, these approaches are time-consuming, requiring hundreds of tuning steps. This is infeasible for online inference, where the tuning time should be minimized[44, 64]. Consequently, there is a need to leverage historical knowledge to expedite the tuning process.

Motivation3: The appropriate parameters setting is crucial for ensuring SLOs and enable an efficient LLM inference. However, existing tuning methods are time-consuming and cannot effectively suggest configurations for online inference workloads in a timely manner.

3 METHODOLOGY

3.1 Overview

This paper presents *InferLog*, a novel framework designed to optimize the inference performance of LLM-based log parsing. The architecture, as illustrated in Fig.8, integrates two synergistic components: an ICL-oriented KV cache reusing optimization technique to improve prefix caching efficiency (§ 3.2), and a fast inference

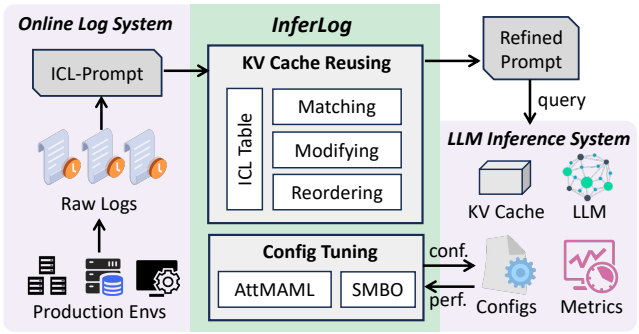


Figure 8. The architecture of *InferLog*.

configuration tuning pipeline to identify optimal inference system configuration parameters to enhance performance(§ 3.3).

Specifically, the online log system generates production logs, which are processed with ICL-based prompt engineering to obtain requests. Upon requests arriving, *InferLog* match the historical ICL with the highest potential prefix cache hit rate from the ICL Table, then dynamically refines prompts by modifying and reordering the demonstrations of ICL to hit the prefix KV cache. To fast recommend optimal configurations, *InferLog* proposes an attention-based meta-learning algorithm where AttMAML trains a good initial regression model using historical tuning data, enabling rapid adaptation to new workloads. Employing AttMAML as the surrogate model, SMBO iteratively explores the parameter space, guided by the meta-learned priors to recommend near-optimal configurations within few iterations.

3.2 ICL-oriented KV Cache Reusing

3.2.1 Prefix-Aware ICL Refinement. As mentioned in § 2.2.2, there is substantial potential for ICL to benefit from prefix caching. To support this, we propose *Prefix-aware ICL Refinement* (PAIR), which refines the demonstration examples to enhance the prefix cache hit rate of the ICL part for a more efficient LLM inference. The primary operations of the PAIR strategy include three main components: ① *Prefix-based Matching*, ② *Modifying* and ③ *Reordering*.

Prefix-based Matching. By maintaining the *ICL Table* to store historical ICL of each request(i.e., the request has completed inference and its KV cache has been stored in the system), which contains a demonstration set: $DS = \{D_1, D_2, ..., D_N\}$, where D is an example consisting of a labeled $\{Log, Template\}$ pair. In PAIR, we first traverse the ICL Table for finding the most matching DS with maximum *prefix-matching count* (PMC), as this is considered to achieve optimal KV cache reuse for the current DS. For example, consider the scenario of Fig.9, although both DS_1 and DS_2 share two examples with the same template with the $DS_{current}$. From the aspect of *prefix*, the first D in DS_1 does not appear in DS_{j+1}, so the PMC for DS_1 is 0, indicating no possibility for prefix cache reuse of DS_1. In contrast, the first two D in DS_2 appear in current DS (i.e, `IPV4 Addr: <*>` and `ARPT: <*>`), resulting in a PMC of 2. Therefore, we mark DS_2 as the DS_{target} for prefix sharing and align the current DS with DS_2.

Modifying. Although DS_{target} and $DS_{current}$ share similar examples with same $Template$, they do not strictly adhere to identical prefix tokens due to variations in Log. Hence, it is essential to

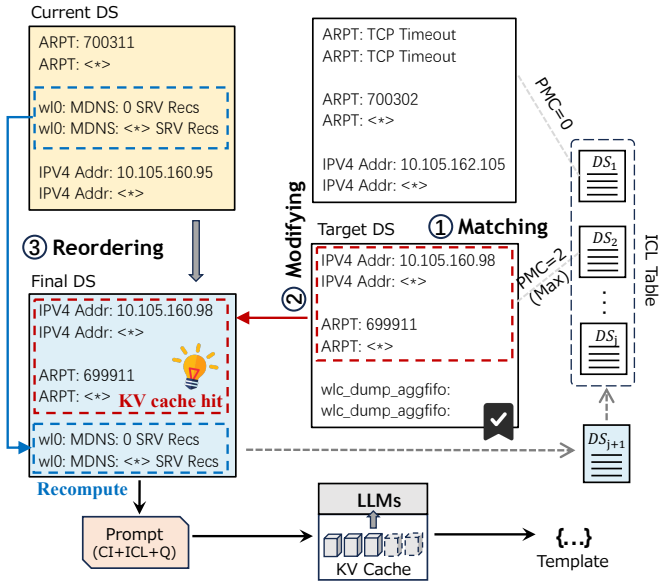


Figure 9. The demonstration cases of matching, modifying and reordering operations in PAIR.

modify the current examples based on DS_{target} to ensure a complete cache hit. For example, the current DS includes the log: `ARPT: 700311` and `IPV4 Addr: 10.105.160.95`, while DS_2 contains log: `ARPT: 699911` and `IPV4 Addr: 10.105.160.98`. These logs adhere to the template `ARPT: <*>` and `IPV4 Addr: <*>`. Since DS_2 already has the KV cache in the memory of the LLM system, we modify the current example to align with the example from DS_2 that shares the same template (indicated by the red dashed box). This operation enables examples with the same template to share the KV cache.

This operation is predicated on the assumption that demonstrations with the same template exhibit high similarity. This is because they provide semantic information that enables the LLMs to distinguish between the template and variable while remaining insensitive to specific variable values, as noted in [41]. This observation is further supported by the hidden representations, which yield an ultra-high cosine similarity of more than 0.99 for the representations of logs with the same template in most cases, as discussed in § 5.4.1.

Reordering. Subsequently, we align the $DS_{current}$ with DS_{target} by reordering ICL examples to achieve prefix cache reuse. As shown in Fig.9, the original DS $[D_1, D_2, D_3]$ is changed to $[D_3, D_1, D_2]$. The former portion of the final DS contains parts that can reuse KV cache, indicated by the red dashed box, whereas the latter part consists of segments that require recompute, indicated by the blue dashed box. The Final DS ensures that a significant portion of the demonstration benefit from prefix caching by directly reusing the LLM's KV cache in memory without additional computation.

3.2.2 ICL Table Management. Currently, *InferLog* has refined the ICL to concatenate the common instruction and the queried log, forming a final prompt(Fig. 2(a)), it then send to the LLM to perform a rapid inference to generate the log template, meanwhile, the KV cache for the cache-missing tokens is calculated and stored in

memory. Due to the high cost and limited capacity of GPU memory, we can not retain the KV cache in memory permanently. So another important design of PAIR is the ICL Table, which serves as a bridge between ICL tokens and KV cache in LLM inference system. Specifically, we employ the *OrderedDict* in Python to store the DS of historical requests due to its ability to maintain element order and support efficient reordering, and maintaining the ICL Table through a *variant of LRU strategy*.

When handling a new request, we first try to perform *Prefix-based Matching* to find DS_{target}, then update the elements in the ICL Table after processing the request. We have established the following effective rules to refine the ICL and update the ICL table in different cases:

(1) If the *PMC* equals *N* (the number of demonstrations), represents $DS_{current}$ and DS_{target} use the entirely same example templates, the $DS_{current}$ is directly modified to DS_{target} and the DS_{target} is relocated to the ICL table's tail to signify recent utilization.

(2) If the *PMC* equals 0, represents no historical DS that uses the same example template as the $DS_{current}$, the $DS_{current}$ remains unchanged due to KV cache unavailability and the $DS_{current}$ is directly appended to the ICL table's tail.

(3) If the *PMC* fall between 1 and *N*, represents there are partially identical templates between the DS_{target} and the $DS_{current}$. The refinement process of $DS_{current}$, as previously shown, will go through the methods of *modifying* and *reordering* to form final DS. The final DS is appended to the ICL Table's tail, while the positioning of DS_{target} in ICL Table remains unchanged.

Whenever the ICL Table reaches its capacity, the eviction of the head element is carried out based on the LRU policy. Notably, *InferLog* offline determines the table length based on the profiling of distribution of requested tokens and remaining GPU memory which can be easily obtained in the inference system[27].

3.3 Fast Inference Config Tuning

3.3.1 Problem Formulation. Formally, given a tuning task (log parsing workload in our scenario), our goal is to find the optimal LLM inference configurations (c) that minimize performance goals (f). The optimization objective is:

$$c^* = arg\min_{c} f(c),\ subject\ \ to\ \ \sum_{k=1}^{K} f(c_k) \le T_{budget}. \tag{1}$$

$f(c)$ is a black-box function, the value can be observed based on a complete workload replay. While increasing the sampling number is benefit for capture configuration-performance plane to identify the optimal configuration, considering the requirement for near real-time responsiveness in online tuning scenario[44, 64], the search process for sampling configurations $\{c_1, c_2, ..., c_k\}$ is under budget constraints T_{budget}. Although many metrics exist to evaluate performance in LLM inference systems like TTFT and TPOT[7]. This paper focuses on optimizing total completion time as a single-objective optimization process because it is easy to measure and optimize.

3.3.2 Attention MAML. Meta-learning, also known as "learning to learn"[48, 50], aims to acquire meta-knowledge from experiences across various tasks to mitigate the issue of requiring a large number of training samples. This paper innovatively designs the Attention MAML algorithm(AttMAML) based on the insight that tasks with similar workload features can provide the meta-learner with more valuable information[10, 64].

Model-Agnostic Meta-Learning. MAML[14] is a popular meta-learning framework that provides good weight initialization of a model that can achieve strong generalization to adapt to new tasks after a few gradient steps. Formally, consider a set of source tasks $T = \{T_1, T_2, ..., T_M\}$ and initial parameters θ^m, for each task T_i and its associated training and validation datasets $(\mathcal{D}^i_{train}, \mathcal{D}^i_{val})$. In the *inner-loop update* process, the task-specific parameters θ_i can be trained by one or more gradient descent steps to fit T_i as follows:

$$\theta_i \leftarrow \theta^m - \alpha \nabla_{\theta^m} \mathcal{L}^{train}_{T_i}\left(f_{\theta^m}\right), \tag{2}$$

Here α is the base learning rate of the learner. The loss function $\mathcal{L}_{T_i}$ is defined as MSE between the predicted performance and the real one.

In the *outer-loop update*, we further calculate the expected meta-loss across all tasks according to the post-update parameters θ_i, this meta-loss from validation data is minimized to optimize the initial parameter value θ^m:

$$\theta^m \leftarrow \theta^m - \beta \nabla_{\theta^m} \frac{1}{M} \sum_{i=1}^{M} \mathcal{L}^{val}_{T_i}\left(f_{\theta_i}\right), \tag{3}$$

Where β is the meta learning rate, M is the number of source tasks. MAML finds the optimal initialization weight that contains the across-task knowledge. After a few steps of fine-tune on a few-shot dataset, the network will perform well on the unseen tasks.

Weighted MAML. MAML assumes equal weights to all tasks, thus failing to discern their relative importance for the target tasks. In real-world scenarios, a target task may be similar to only a few of the source tasks or even out-of-distribution, applying equal weighting to all sources during meta-training can be detrimental[22]. To address this issue, the accuracy of meta-loss can be enhanced by employing a weighted averaging, which captures different importance between tuning tasks. The *weighted meta-loss* is:

$$\mathcal{L}_{meta}(\theta^m) = \frac{1}{W} \sum_{i=1}^{M} w^i \mathcal{L}^{val}_{T_i}\left(f_{\theta_i}\right), W = \sum_{i=1}^{M} w^i. \tag{4}$$

Automatic Weights Learning. Inspired by prior work[17, 54], *InferLog* implements the *attention mechanism*[49] to automatically learn optimal weight distributions, indicating the importance of meta-tasks in relation to target tasks. Specifically, it explores the potential impact relationships between meta-tasks and the target task by learning the dynamic attention among workload feature vectors s, thereby facilitating task-specific model initialization for the target domain. To acquire adaptive weights within the context of MAML, we assign a separate attention module with learnable parameters θ^{att}, the attention weight w^i between target task t and meta task i compares the workload feature vector s_t and s_k through a Query-Key system:

$$w^i \propto softmax(s_t W_q W_k^T s_i^T), \tag{5}$$

where Query matrix W_q transforms s_j into a query and Key matrix W_k transforms s_k into a key, the matching is then scaled to prevent vanishing gradients and ensure the stability of the learning process. The parameters of the attention network θ^{att} are updated in accordance with the *outer-loop update* process using the *weighted*

meta-loss in Eq.4:

$$\theta^{att} \leftarrow \theta^{att} - \beta \nabla_{\theta^{att}} \mathcal{L}_{meta}(\theta^{m}, \theta^{att}), \tag{6}$$

3.3.3 Enhancing SMBO with AttMAML. Sequential Model-based Optimization (SMBO) is a powerful method for configuration tuning[13, 52]. The standard workflow of SMBO is: 1) Utilizing the *surrogate model* and the *acquisition function* to determine the next configuration. 2) Evaluating the newly generated configuration to obtain corresponding performance. 3) Adding the new $\langle configuration, performance \rangle$ sample into the observation dataset and then updating the *surrogate model*. This process is repeated until the optimal solutions are identified or the search cost exhaustion. the workflow of configuration tuning in *InferLog* is shown in Fig. 10.

AttMAML as Surrogate Model. Traditional surrogates like Gaussian Processes (GP)[43] and Random Forest are constrained to the present task and cannot integrate insights from historical tasks, this restricts their effectiveness across diverse applications. *InferLog* have learned a proficient initialization using AttMAML algorithm that enables the model to accurately predict the performance of configurations for new tasks after a few steps of gradient descent. Therefore, *InferLog* employs the offline trained AttMAML as the surrogate model, leveraging the tuning experience from the similar history tasks and transfering the accumulated knowledge to accelerate the tuning process of the new tasks.

Besides, following previous work[10, 12], we adopt the expected improvement(EI)[5] acquisition function due to its effective balance between exploration and exploitation at a low computational cost.

MC Dropout for Uncertainty Approximation. In classic BO, the widely used GP[43] provides a measure of uncertainty for the predictions of unsampled data points, this uncertainty information is crucial to the optimization process as it helps balance exploration of uncertain areas with exploitation of known good regions.

To provide uncertainty analysis for SMBO, we employ the *Monte Carlo Dropout* (MC Dropout)[15] strategy to obtain mean and variance information. This approach is an effective method for uncertainty quantification in neural networks. Specifically, during the prediction phase of AttMAML model, we enable dropout, which allows us to perform multiple inferences while randomly activating different neurons each time. By calculating the mean μ and variance σ^2 of the predicted results, we can effectively assess the meta-model's uncertainty:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} \hat{y}^{(i)}, \sigma^2 = \frac{1}{N}\sum_{i=1}^{N} (\hat{y}^{(i)} - \mu)^2. \tag{7}$$

where N is the number of inferences performed and $\hat{y}$ presents predicted result obtained with dropout.

Warm Strat. To accelerate the convergence during the initial phase and effectively leverage successful historical knowledge, the meta-learning module proposes initial configurations by assessing task similarity, rather than relying on random sampling. In particular, we choose three samples that represent the best configuration of top-3 most similar source tasks as initial points follow prior works[32, 58] to make the fast adaption to the new tuning task. This not only enables the meta-model to quickly adapt to new environments using few-shot learning, but also allows for more efficient exploration guidance of the search space in SMBO process.

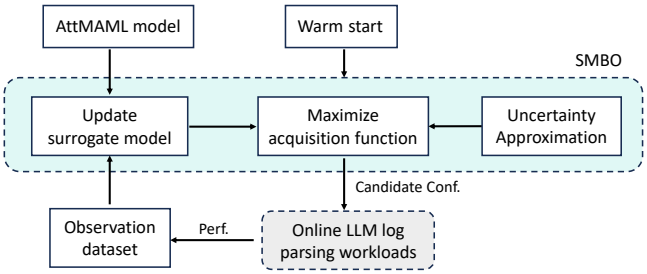


Figure 10. *InferLog*'s configuration tuning workflow.

4 EXPERIMENTAL EVALUATION

In this section, we evaluate *InferLog* to answer four questions:

- **RQ1:** How does *InferLog* compare against other inference acceleration methods for online log parsing workloads?
- **RQ2:** How does *InferLog* adapt to different LLMs and dynamic invocations?
- **RQ3:** How does *InferLog* accelerate exsiting LLM-based log parsers?
- **RQ4:** What is the contribution of each design module?

4.1 Experimental Settings

Environment and Implementation. We conduct our experiments on a NVIDIA Tesla A40 48G GPU. The CUDA version on the NVIDIA platform is 12.4. The CPU is Intel(R) Xeon(R) Gold 6326 CPU @2.90GHz, 128 GiB host memory. The operation system is Ubuntu 20.04. We implement *InferLog* with Python 3.11. We adopt vLLM as the LLM inference system, the version is v0.6.3. We use `Qwen2.5-14B`[42] as underlying LLM because it is a relatively small yet powerful model, balancing performance and efficiency in various tasks. In RQ2, we evaluate *InferLog* by replacing `Qwen2.5` with other open-source LLMs.

During the AttMAML training process, we divided the log datasets into two groups: half for meta-training and the remaining half for testing. To obtain prior tuning knowledge, we apply the Latin hypercube sampling(LHS)[39] to uniformly sample 100 points within the configuration space under each meta-training task for training AttMAML. Besides, in training phase, the sizes of the support set and query set in MAML are 15 and 45, the meta learning rate is set to 0.0001 and the base learning rate is set to 0.001. The model consists of two fully connected layers, with 64 neurons each, and the attention matrix has a dimension of 32. We report the performance of optimal configuration within 15 steps.

Datasets. We assess the effectiveness of our method on log datasets from Loghub-2k[68]. It is a widely recognized benchmark in the field of log parsing. It encompasses logs from 16 diverse systems, including distributed systems, supercomputers, operating systems, mobile platforms, server applications, and individual software packages. Each data source includes 2,000 log messages along with their respective log templates.

Configurations and Workloads. We optimize three critical *scheduling-related* configuration parameters in vLLM as listed in Tab.1, where *#Max* represents `max-position-embeddings` of LLM. To create prompt, we samples 200 logs from each log dataset to

Table 1. Parameters to tune.

Configuration Parameter	Type	Range
`max-num-batched-tokens`	Integer	[4000,#*Max*]
`max-num-seqs`	Integer	[64,256]
`scheduler-delay-factor`	Float	[0,2]

construct the candidate set and select 5 labeled examples from the candidate set in ICL following DivLog[60]. We conduct optimization on a fixed batch of 2000 log parsing requests processed with a fixed concurrency level of 100 by default. We use Python asyncio module to simulate concurrency online requests. In RQ2, we evaluate it's adaptability under dynamic concurrency.

Metrics. Considering the low-latency requirements for online log parsing and the high GPU utilization of the inference system, we use the **p95 end-to-end latency** and **throughput** of requests as performance metrics for the LLM inference system. p95 end-to-end latency refers to the time within which 95% of all requests are completed, while throughput refers to the average number of requests completed per second. Besides, following the established metrics outlined in [23, 60] for accuracy evaluation of log parsing, we employ Parsing Accuracy (PA), Precision Template Accuracy (PTA), Recall Template Accuracy (RTA), and Grouping Accuracy (GA) metrics.

Baselines. We compare *InferLog* with the following baselines. (1)***Default***: we treat vLLM[27] with default configurations as the the basic system. (2)***PC***[6] uses default prefix caching technique for optimizing inference latency in vLLM. (3)***PC-Tune***: we adopt random search[4] to optimize inference configuration parameters based on PC. (4)***PC-Chunk***[1] optimizes context filling efficiency by processing input text in chunks, enhancing generation speed and memory usage. (5)***BatchLLM*** [66] designs prefix-sharing group based scheduling and request groups reordering strategies to improve the prefix caching ratio. Beside, it also determines the optimal chunk size of chunked-prefills[1] scheduling policy to increase the throughput. Notably, except for the default, all other approaches have the prefix caching option enabled.

5 RESULTS AND ANALYSIS

5.1 Effectiveness Validation (RQ1)

5.1.1 Inference Performance Improvement. Fig.11 provides a comparative analysis of various inference optimization methods across 16 datasets in terms of latency and throughput. Experimental results show that *InferLog* significantly outperforms other methods. Specifically, *InferLog* achieves a 71.9% 43.9%, 34.6%, 38.0% and 26.7% reduction in p95 latency, as well as a 4.02×, 1.62×, 1.51×, 2.21×, 1.35× speedup on throughput compared to default vLLM[27], PC[6], PC-Tune, PC-Chunk[1] and BatchLLM[66] on average, respectively.

Specifically, the default inference does not employ any optimization techniques, and the system fully executes the request inference process, resulting in the worst inference performance. Due to the fact that numerous log parsing requests with same log templates in log parsing task, they tend to opt for the similar demonstrations in ICL. Therefore, employing PC can lead to a significant acceleration in inference performance. However, PC[6], PC-Tune and PC-Chunk can only strictly reuse the KV cache with identical prefixes, which limits the reuse of KV cache between requests. PC-Chunk[1] utilize chunk-prefill[1] technique to divide requests into chunks and execute them in parallel. However, the chunking operation introduces additional overhead, which is not conducive to the execution of short requests like log parsing, so it leads to a lower throughput than PC and PC-Tune. BatchLLM[66] performs better than other methods because it reorders the requests to improve the prefix cache hit rate. However, BatchLLM is unable to perform fine-grained ICL reordering within individual requests, leading to suboptimal KV cache reuse. Additionally, the adjustment of request order also introduces waiting delays. In contrast, *InferLog* adjusts the ICL examples of each incoming request, enabling optimal prefix cache utilization, and further optimizes the configuration to enhance performance.

Overall, the experimental results demonstrate that InferLog is more effective than existing inference optimization methods and can be applied to various online log parsing requests.

5.1.2 Parsing Accuracy. As mentioned in § 2.2.2, existing LLM-based parsers use ascending order to arrange ICL examples[23, 60], which has been proven to perform better than the descending and random orders. However, *InferLog* adopts the PAIR by modifying and reordering the examples to enhance the prefix KV cache hit rate, thereby accelerating the inference process. Tab.2 provides a comparative analysis of *w/o Inferlog* and *w/ Inferlog* across 16 datasets in terms of PA, PTA, RTA, and GA metrics. we marked the best results in each metric in bold font. Compared to original ICL examples with ascending permutation(*w/o InferLog*), *w/ InferLog* achieves comparable log parsing accuracy. Specifically, the changes in the four metrics of *InferLog* compared to *w/o InferLog* are +0.7, -0.1, +0 and +0.3. ***This indicates that InferLog achieves faster LLM inference without sacrificing log parsing accuracy***.

5.2 Generalizability Analysis (RQ2)

5.2.1 Performance Under Dynamic Workloads. In real production, the workload always changes over time. We select the concurrency of 50, 75, 100, 125, 150 and 175 to evaluate it's adaptability on different workloads, the results are shown in Fig.12. It is evident that *InferLog* performs well across all concurrency settings, compared to other methods, *InferLog* reduces the p95 latency by 42.43% and increases 1.58× throughput across all concurrencies on average. ***The results demonstrate that InferLog can adapt well to varying workloads and achieve stable performance improvements.***

5.2.2 Performance Under Different LLMs. To validate LLM generalization, we evaluate two representative LLMs with divergent architectures: (1) `LLaMA3-8B`[2], and (2) the sparse Mixture-of-Experts (MoE)-based `Mixtral-7B`[21]. As shown in Fig.13, *InferLog* achieves consistent performance enhancements across evaluated models. Specifically, it surpasses the baselines by 40.53% reduction on p95 latency and 1.45× throughput for `LLaMA3-8B`.

Due to the computational and memory access overhead introduced by the MOE mechanism, the inference speed of `Mixtral` model is slower than that of `LLaMA3`, *InferLog* achieves 20.52% reduction on p95 latency and 1.26× throughput for `Mixtral-7B`. ***This provides compelling evidence that InferLog is capable of effectively accelerating a variety of LLMs with diverse architectures.***

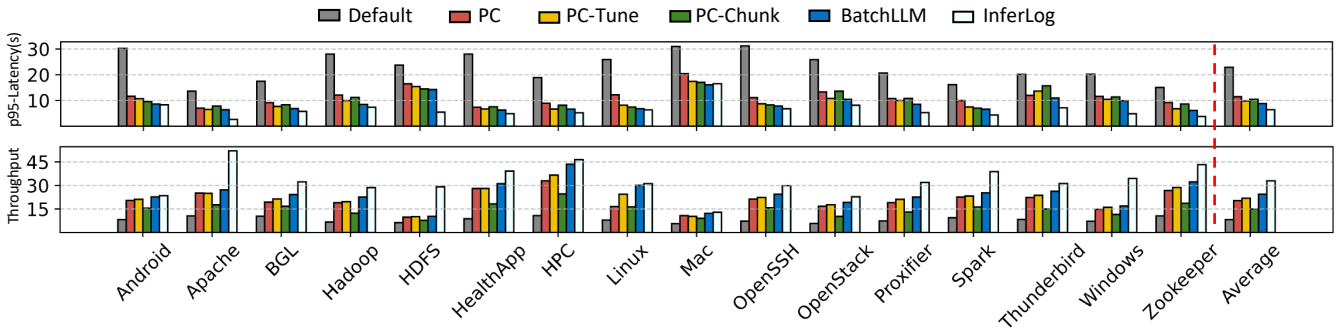


Figure 11. Performance of *InferLog* compared to other inference optimization methods on public datasets (Lower is better for p95-latency, higher is better for throughput).

Table 2. Log parsing accuracy comparison.

	w/o *InferLog*				w/ *InferLog*			
	PA	PTA	RTA	GA	PA	PTA	RTA	GA
Android	**97.4**	86.3	88.0	**98.9**	**97.4**	**87.0**	**90.0**	97.4
Apache	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
BGL	**97.8**	**94.2**	**95.8**	**99.3**	**97.8**	92.7	95.0	98.3
Hadoop	92.9	**91.2**	**91.2**	**100.0**	**99.3**	88.6	88.6	99.8
HDFS	**99.8**	**75.0**	**85.7**	**82.9**	99.6	75.0	**85.7**	**82.9**
HealthApp	97.5	87.3	92.0	87.8	**98.1**	**93.4**	**94.7**	**99.2**
HPC	98.4	**75.0**	**84.8**	**94.9**	**99.3**	71.7	82.6	93.4
Linux	**99.8**	**95.6**	**95.6**	**100.0**	99.6	93.0	92.2	99.9
Mac	72.3	**64.2**	72.1	**79.2**	**74.2**	63.0	**73.0**	76.9
OpenSSH	93.2	**88.9**	**92.3**	**93.3**	**93.3**	**88.9**	**92.3**	**93.3**
OpenStack	**99.8**	**90.7**	**95.1**	**97.9**	**99.8**	86.4	92.7	96.8
Proxifier	**99.9**	**69.2**	**75.0**	**67.8**	99.7	64.3	**75.0**	54.4
Spark	**99.9**	**94.4**	**94.4**	**100.0**	**99.9**	**94.4**	**94.4**	**100.0**
Thunderbird	**88.9**	83.0	85.2	98.3	**89.9**	**86.2**	**87.9**	**99.0**
Windows	99.0	74.0	80.0	99.4	**99.3**	**82.3**	**84.0**	**99.9**
Zookeeper	**99.9**	90.4	**94.0**	84.9	**99.9**	**92.2**	**94.0**	**99.5**
Average	96.0	**85.0**	**88.9**	92.8	**96.7**	84.9	**88.9**	**93.1**

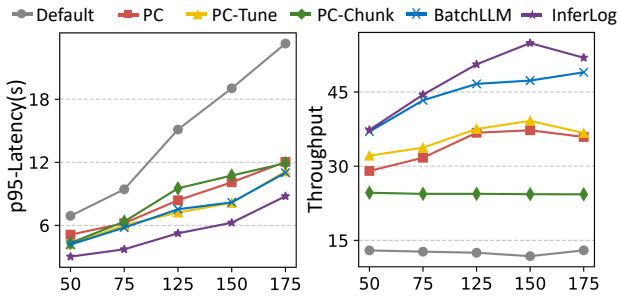


Figure 12. P95-latency and throughput on different concurrency.

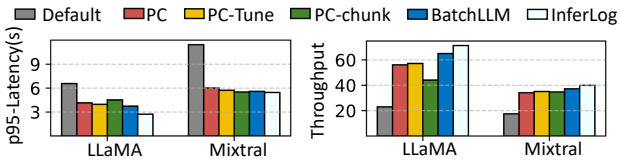


Figure 13. P95-latency and throughput on different LLMs.

5.3 Compatibility Evaluation (RQ3)

In this RQ, our primary focus is on assessing the compatibility of our method with current LLM-based log parsers. Specifically, recent studies have proposed methods to enhance the efficiency of LLM-based log parsing by analyzing the characteristics of log parsing tasks. For instance, LILAC[23] adopts an adaptive parsing cache to store the template to prevent duplicate queries of the LLM, while LogBatcher[57] employs a clustering method and generate log template for each partition. Therefore, we want to see if our method can work together with the current approaches and further improve their inference efficiency. We evaluate the performance of current LLM-based log parsers: DivLog[60], LILAC[23] and LogBatcher[57] with the acceleration by the *Inferlog*. To avoid the additional overhead of the log parser itself, for DivLog, we query all log parsing requests in the dataset. For LogBatcher, we first offline partition logs and perform only one query on the logs of each partition. For LILAC, we only query LLM for log parsing requests with different templates. The number of examples is set to 5.

The average results across 16 datasets are illustrated in Tab.3. It is obvious that with *InferLog*, the performance of current log parsers has been significantly enhanced. Specifically, for DivLog, the performance improvement is the most pronounced, the reason is that a large batch of queries have brought performance bottlenecks to the LLM inference system. Through PAIR and configuration optimization by *InferLog*, the latency is reduced from 22.89s to 6.43s, marking a 71.90% decrease, while the throughput improvement is 301.83%. For LILAC and LogBatcher, which eliminate redundant LLM queries and focus on parsing unseen logs, the average number of log parsing queries to the LLM is reduced by ~23×. Nonetheless, *InferLog* still achieves a performance improvement of 50.17% and 49.94% in latency reduction and 66.07% and 78.31% in throughput increase for LILAC and LogBatcher, respectively. The reason is that there are still reusable KV cache in the common instruction and ICL examples among different log parsing requests, so *InferLog* effectively boosts prefix cache hit rates and optimizes inference performance. ***Overall, InferLog can improve the efficiency of LLM-based log parsers and is compatible with existing techniques that aim to reduce LLM invocation times.***

Table 3. Results of accelerating LLM-based log parsers.

	p95-Latency	Throughput
DivLog	22.89	8.20
DivLog w/ *InferLog*	6.43 (↓ 71.90%)	32.95 (↑ 301.83%)
LILAC	12.30	5.04
LILAC w/ *InferLog*	6.13 (↓ 50.17%)	8.37 (↑ 66.07%)
LogBatcher	15.37	4.92
LogBatcher w/ *InferLog*	7.69 (↓ 49.95%)	8.77 (↑ 78.31%)

Table 4. Ablation study results.

	p95-Latency	Throughput
InferLog	6.43	32.95
InferLog w/o PAIR	9.45 (↑ 46.97%)	22.4 (↓ 32.02%)
InferLog w/o Config Tuning	9.88 (↑ 53.65%)	28.16 (↓ 14.54%)

5.4 Ablation Study (RQ4)

This RQ gives a comprehensive explanation of each module's contribution, i.e., the PAIR in and the inference configuration tuning. We create the following two variants of *InferLog* and compare them with the original approach. (1) *InferLog* w/o PAIR: remove the ICL refinement and only keep default PC. (2) *InferLog* w/o Config Tuning: remove configuration tuning to default setting. Tab.4 shows the average results cross 16 datasets. It is clear that removing any of the two modules will affect performance to some extent.

5.4.1 Contribution of PAIR. The dynamic nature of ICL in log parsing poses significant challenges for benefiting from PC techniques. PAIR is a crucial module that enables a significant part of ICL tokens to leverage the historical KV cache, thereby bypassing redundant computations cross online log parsing requests. As shown in Tab.4, when PAIR is removed, LLM can only reuse the KV cache of common instruction in user requests, due to its strict requirement for identical prefixes. Without it, the inference latency increases by 46.97% and processing throughput drops by 32.02%.

PC Hit Rate. To better understand the performance drop, Fig.14 illustrates the average PC hit rates across all datasets. Obviously, PAIR achieves the highest hit rate, with average 1.23×, 1.16× and 1.18× higher than naive PC[6], PC-Chunk[1] and BatchLLM[66], and reaching up to 2.1× on the HDFS dataset compared with PC, confirming the effectiveness of PAIR.

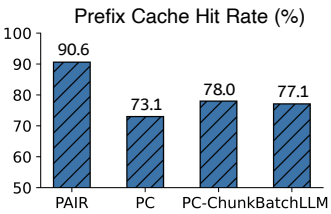


Figure 14. Prefix cache hit rate.

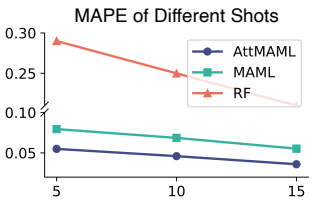


Figure 15. MAPE of few-shot learning.

Impact on Parsing Accuracy. To delve deeper into the parsing accuracy, we analysis the hidden representations when implementing PAIR by `Qwen2.5-14B`. First, we compared the representations of input tokens where examples were positioned in various orders, which reflects the impact of *Reordering*. To create different orders, we adopt the same setting from prior work[56]. As depicted in Fig.16, it presents the heatmaps of average representations similarities across different example positions of Apache dataset. It is noteworthy that the cosine similarity across representations at any position typically exceeded 0.80 and up to 0.86, that is considered to possess sufficient robustness to counterbalance the demonstration order sensitivity of decoder-only LLMs according to prior work[56]. We believe that with the enhancement of LLM capabilities, their context understanding and noise resistance will improve, rendering them more robust to variations in ICL order. Furthermore, carefully designed prompt engineering also reduces the sensitivity of LLM to input order, enables LLM to comprehend tasks at a semantic level, thereby diminishing sensitivity to sequence.

Second, we compared ICL after *Modifying* with the original ICL on the entire prompt representation. The average similarity of output representations from the last layer is 0.9977 and 0.9983 for HealthApp and Apache dataset, respectively (omitted in the figure). In general, the semantic impact of our PAIR strategy on the prompt is not obvious, and it still maintains the high level log parsing accuracy.

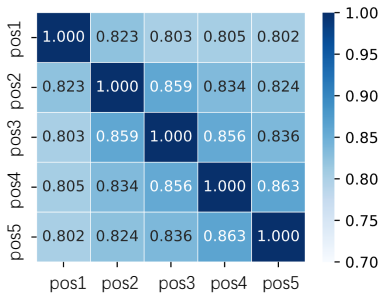


Figure 16. The heatmaps of similarities in representations of prompt from the last layer outputs across different ICL positions.

5.4.2 Contribution of Configuration Tuning. Inference configurations directly influence the request scheduling and execution performance of LLMs, *InferLog* aims to provide an efficient and highly transferable configuration tuning pipeline to further enhance the performance of inference systems. Without configuration tuning, the inference latency degrades by 53.65% and processing throughput drops by 14.5%. Notably, a good configuration achieves significant improvements in tail latency, which is crucial for meeting SLOs. Indeed, it limits the tokens processed per inference step, thereby preventing computational overload in prefill phase, which effectively alleviates the performance bottlenecks identified in § 2.2.1.

Prediction Accuracy of AttMAML. *InferLog* innovatively integrates the attention mechanism into meta-learning framework, enabling the MAML algorithm to automatically identify which meta-training tasks are more critical for the target tasks. This enhancement not only boosts the precision of the MAML algorithm but also enhances its ability to rapidly adapt to target tasks. Specifically, we conducted meta-learning on eight tasks and evaluated the few-shot learning performance on another eight tasks. The average mean absolute percentage error (MAPE) on target tasks are shown in Fig.15, we find that (i) compared to Random Forest, the representation of machine learning, AttMAML demonstrates a 81.76% reduction on MAPE, indicating it's ability to transfer knowledge effectively in facing new scenarios. (ii) AttMAML achieves a 32.74% MAPE reduction over the standard MAML algorithm on average, emphasizing the effectiveness of the attention mechanism in improving meta-training performance.

Online Tuning Efficiency. Efficiency is a crucial consideration in recent configuration tuning methods[9, 45]. To evaluate it, we compare *InferLog* with several prominent algorithms: LHS[39], BO with Gaussian Processes (BO-GP)[43], DDPG[33], and the SOTA method for LLM configuration tuning SCOOT[7], the optimization was conducted within a sampling budget of 15 steps. We show tuning results of three datasets in the Fig.17. *InferLog* outperforms other

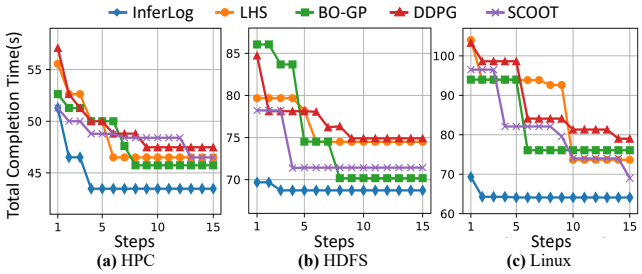


Figure 17. The current best performance along with the 15 online tuning steps.

methods by requiring 40.74%, 53.85% and 79.68% fewer iterations to determine the optimal configuration on HPC, HDFS and Linux, resulting in a performance improvement of 1.07×, 1.05× and 1.13×. The reason is that *InferLog* use a warm-start strategy to leverage historical optimal configurations from similar workloads to achieve rapid early convergence. Additionally, AttMAML-enhanced SMBO enables fine tune meta-model within few steps in new enviroments, efficiently guiding configurations tuning process.

6 DISCUSSION

***InferLog* is orthogonal with LLM-based log parsers**. Recent LLM-based log parsers[20, 23, 55, 57, 60] design log parsing algorithm such as in-context learning, template caching or clustering, aiming to improve parsing accuracy. Conversely, *InferLog* targets the underlying LLM inference system, it acts as an intermediate layer between log parsers and the LLM inference system, *InferLog* is agnostic to the upstream parsers and seamlessly integrates with them to improve inference efficiency via prefix KV cache reusing and configuration tuning without compromising accuracy.

***InferLog* efficiently accelerates log parsing task than other LLM inference workloads**. Although methods like KV cache reusing and configuration tuning in *InferLog* can accelerate LLM inference in other contexts, there are two main features in log parsing make *InferLog* specifically suitable for log parsing and may lead to suboptimal performance in other scenarios. Firstly, the order of in-context learning examples in log parsing has little impact on *InferLog*'s results. This is mainly because log parsing exhibits greater predictability and stability in its output structure, focusing on extracting log templates, so the model's output is less affected by the order of examples. In contrast, in long document question answering, the relative position of information within the document is more critical for accurate answers[25, 34]. Secondly, configuration tuning process requires a stable mapping between configuration and metrics. Log parsing task outputs stable log templates with predictable token length, creating a foundation for configuration tuning. However, in general workloads, LLM output length is dynamic, disrupting tuning model convergence.

6.1 Threats to Validity

We have identified the following threats to validity:

Subject Systems. We conduct experiments on the vLLM inference system, due to its high popularity. We also compare the inference performance across various open-source LLMs. In the future, we plan to evaluate our method on more inference systems and models. *InferLog* only requires the target inference system and LLMs to support prefix caching, without the need for system modifications. Overall, *InferLog* can efficiently perform log parsing tasks on locally deployed open-source inference systems and LLMs without privacy concerns.

Randomness. Randomness may affect the performance through two aspects:(i) the randomness of LLM output and (ii) the randomness introduced during the performance testing of the system, particularly under online concurrent requests where task scheduling introduces variability in system performance. To mitigate the former threat, we set the `temperature` to 0, guaranteeing consistent outputs for identical input text. To mitigate the latter threat, each experiment was repeated three times for every experimental setting, and we report the the average results as the final outcome.

7 RELATED WORK

LLM-based Log Parsing. Log parsing is a critical preliminary step for various log analysis tasks. Therefore, numerous efforts have been made to achieve accurate log parsing[11, 18, 24, 40, 46, 47, 62]. With the rise LLMs, a series of LLM-based log parsers have been developed to achieve more effective log parsing[20, 23, 37, 38, 55, 57, 60, 67]. These LLM-based log parsers leverage fine-tuning[37], in-context learning[23, 55, 60]or log clustering analysis[55, 57] to specialize LLMs for log parsing tasks. Considering high invocation cost, some efforts have been made to achieve cost-effective LLM-based log parser. LILAC[23] enhances the efficiency of LLM-based log parsing by incorporating an adaptive parsing cache that stores parsing results, Logbatcher[57] and LUNAR[20] partition logs and only queries LLM once for logs belonging to the same cluster. *InferLog* identifies the performance bottleneck in LLM inference under concurrent requests and optimized by ICL-oriented prefix caching, orthogonal to existing techniques for reducing query times.

LLM Inference Optimization. Existing inference engines, such as vLLM[27] and SGLang[65], integrate numerous advanced techniques such as iteration-level scheduling[61], PagedAttention[27], chunked prefill[1] and speculative decoding[29]. Reusing KV cache across different requests have been applied to recent works. Prompt Cache[16] allows flexible reuse of the same tokens at different positions. Liu et al.[35] proposed column reordering and row sorting to improve the prefix KV cache hit rate in relational analytical workload. RAGCache[25] improved the cache efficiency for RAG system. *InferLog* optimizes prefix KV cache reusing for ICL paradigm and enhances performance based on configuration tuning.

8 CONCLUSION

In this work, we propose *InferLog*, a novel framework for optimizing LLM inference in online log parsing tasks. Based on the characteristics of LLM-based log parsing tasks and LLM inference performance, *InferLog* employs PAIR that consisted of prefix-based matching, modifying and reordering to refine ICL examples, thereby enhancing prefix KV cache hit rates. *InferLog* further combines attention-augmented meta-learning with SMBO for rapid, tailored configuration optimization. Experiments confirm its effectiveness in enhancing inference efficiency while preserving log parsing quality. The source code and data is available at https://github.com/wiluen/InferLog.

REFERENCES

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. 2023. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369* (2023).

[2] AI@Meta. 2024. Llama 3 Model Card. (2024). https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md

[3] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.

[4] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *The journal of machine learning research* 13, 1 (2012), 281–305.

[5] Christopher M Bishop and Nasser M Nasrabadi. 2006. *Pattern recognition and machine learning*. Vol. 4. Springer.

[6] Automatic Prefix Caching. 2024. https://docs.vllm.ai/en/latest/design/automatic_prefix_caching.html.

[7] Ke Cheng, Zhi Wang, Wen Hu, Tiannuo Yang, Jianguo Li, and Sheng Zhang. 2024. Towards SLO-Optimized LLM Serving via Automatic Inference Engine Tuning. *arXiv preprint arXiv:2408.04323* (2024).

[8] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[9] Hui Dou, Yilun Wang, Yiwen Zhang, and Pengfei Chen. 2022. DeepCAT: A cost-efficient online configuration auto-tuning approach for Big Data frameworks. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[10] Hui Dou, Lei Zhang, Yiwen Zhang, Pengfei Chen, and Zibin Zheng. 2023. TurBO: A cost-efficient configuration-based auto-tuning approach for cluster-based big data frameworks. *J. Parallel and Distrib. Comput.* 177 (2023), 89–105.

[11] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[12] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. To tune or not to tune? in search of optimal configurations for data analytics. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2494–2504.

[13] Matthias Feurer, Jost Springenberg, and Frank Hutter. 2015. Initializing bayesian hyperparameter optimization via meta-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 29.

[14] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*. PMLR, 1126–1135.

[15] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*. PMLR, 1050–1059.

[16] In Gim, Guojun Chen, Seung-seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt cache: Modular attention reuse for low-latency inference. *Proceedings of Machine Learning and Systems* 6 (2024), 325–338.

[17] Qianyue Hao, Wenzhen Huang, Tao Feng, Jian Yuan, and Yong Li. 2023. GAT-MF: Graph Attention Mean Field for Very Large Scale Multi-Agent Reinforcement Learning. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 685–697.

[18] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*. IEEE, 33–40.

[19] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)* 54, 6 (2021), 1–37.

[20] Junjie Huang, Zhihan Jiang, Zhuangbin Chen, and Michael Lyu. 2025. No More Labelled Examples? An Unsupervised Log Parser with LLMs. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2406–2429.

[21] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[22] Xiang Jiang, Mohammad Havaei, Gabriel Chartrand, Hassan Chouaib, Thomas Vincent, Andrew Jesson, Nicolas Chapados, and Stan Matwin. 2018. Attentive task-agnostic meta-learning for few-shot text classification. (2018).

[23] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2024. Lilac: Log parsing using llms with adaptive parsing cache. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 137–160.

[24] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting execution logs to execution events for enterprise applications (short paper). In *2008 The Eighth International Conference on Quality Software*. IEEE, 181–186.

[25] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. 2024. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *arXiv preprint arXiv:2404.12457* (2024).

[26] Hongwei Jin, George Papadimitriou, Krishnan Raghavan, Pawel Zuk, Prasanna Balaprakash, Cong Wang, Anirban Mandal, and Ewa Deelman. 2024. Large Language Models for Anomaly Detection in Computational Workflows: From Supervised Fine-Tuning to In-Context Learning. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–17.

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[28] Van-Hoang Le and Hongyu Zhang. 2023. Log Parsing with Prompt-based Few-shot Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2438–2449. https://doi.org/10.1109/ICSE48619.2023.00204

[29] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[30] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[31] Yichen Li, Yintong Huo, Renyi Zhong, Zhihan Jiang, Jinyang Liu, Junjie Huang, Jiazhen Gu, Pinjia He, and Michael R Lyu. 2024. Go static: Contextualized logging statement generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 609–630.

[32] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, and Bin Cui. 2023. Towards General and Efficient Online Tuning for Spark. *Proc. VLDB Endow.* 16, 12 (2023), 3570–3583.

[33] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[34] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).

[35] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. 2024. Optimizing llm queries in relational workloads. *arXiv preprint arXiv:2403.05821* (2024).

[36] Yudong Liu, Xu Zhang, Shilin He, Hongyu Zhang, Liqun Li, Yu Kang, Yong Xu, Minghua Ma, Qingwei Lin, Yingnong Dang, et al. 2022. Uniparser: A unified log parser for heterogeneous log data. In *Proceedings of the ACM Web Conference 2022*. 1893–1901.

[37] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. Llmparser: An exploratory study on using large language models for log parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[38] Zeyang Ma, Dong Jae Kim, and Tse-Hsun Chen. 2024. LibreLog: Accurate and Efficient Unsupervised Log Parsing Using Open-Source Large Language Models. *arXiv preprint arXiv:2408.01585* (2024).

[39] Michael D McKay, Richard J Beckman, and William J Conover. 2000. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics* 42, 1 (2000), 55–61.

[40] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th Conference on Program Comprehension*. 167–177.

[41] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What makes In-context Learning Work?. In *EMNLP*.

[42] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/

[43] Carl Edward Rasmussen. 2003. Gaussian processes in machine learning. In *Summer school on machine learning*. Springer, 63–71.

[44] Rajkumar Sen, Jack Chen, and Nika Jimsheleishvilli. 2015. Query optimization time: The new bottleneck in real-time analytics. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. 1–6.

[45] Yu Shen, Xinyuyang Ren, Yupeng Lu, Huaijun Jiang, Huanyong Xu, Di Peng, Yang Li, Wentao Zhang, and Bin Cui. 2023. Rover: An online Spark SQL tuning service via generalized transfer learning. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4800–4812.

[46] Keiichi Shima. 2016. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213* (2016).

[47] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 785–794.

[48] Sebastian Thrun and Lorien Pratt. 1998. Learning to learn: Introduction and overview. In *Learning to learn*. Springer, 3–17.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[50] Ricardo Vilalta and Youssef Drissi. 2002. A perspective view and survey of meta-learning. *Artificial intelligence review* 18 (2002), 77–95.

[51] vLLM Engine Arguments. 2025. https://docs.vllm.ai/en/latest/serving/engine_args.html.

[52] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. 2021. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of the ACM Symposium on Cloud Computing*. 639–653.

[53] Xuheng Wang, Xu Zhang, Liqun Li, Shilin He, Hongyu Zhang, Yudong Liu, Lingling Zheng, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2022. SPINE: a scalable log parser with feedback guidance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1198–1208.

[54] Yilun Wang, Pengfei Chen, Hui Dou, Yiwen Zhang, Guangba Yu, Zilong He, and Haiyu Huang. 2024. FaaSConf: QoS-aware Hybrid Resources Configuration for Serverless Workflows. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 957–969.

[55] Yifan Wu, Siyu Yu, and Ying Li. 2025. Log Parsing Using LLMs with Self-Generated In-Context Learning and Self-Correction. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. IEEE, 01–12.

[56] Yanzheng Xiang, Hanqi Yan, Lin Gui, and Yulan He. 2024. Addressing Order Sensitivity of In-Context Demonstration Examples in Causal Language Models. *arXiv preprint arXiv:2402.15637* (2024).

[57] Yi Xiao, Van-Hoang Le, and Hongyu Zhang. 2024. Free: Towards More Practical Log Parsing with Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 153–165.

[58] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. Locat: Low-overhead online configuration auto-tuning of spark sql applications. In *Proceedings of the 2022 International Conference on Management of Data*. 674–684.

[59] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2024. Unilog: Automatic logging via llm and in-context learning. In *Proceedings of the 46th ieee/acm international conference on software engineering*. 1–12.

[60] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2024. DivLog: Log Parsing with Prompt Enhanced In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[61] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[62] Siyu Yu, Pinjia He, Ningjiang Chen, and Yifan Wu. 2023. Brain: Log parsing with bidirectional parallel tree. *IEEE Transactions on Services Computing* 16, 5 (2023), 3224–3237.

[63] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 international conference on management of data*. 415–432.

[64] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In *Proceedings of the 2021 international conference on management of data*. 2102–2114.

[65] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently programming large language models using sglang. *arXiv e-prints* (2023), arXiv–2312.

[66] Zhen Zheng, Xin Ji, Taosong Fang, Fanghao Zhou, Chuanjie Liu, and Gang Peng. 2024. BatchLLM: Optimizing Large Batched LLM Inference with Global Prefix Sharing and Throughput-oriented Token Batching. *arXiv preprint arXiv:2412.03594* (2024).

[67] Aoxiao Zhong, Dengyao Mo, Guiyang Liu, Jinbu Liu, Qingda Lu, Qi Zhou, Jiesheng Wu, Quanzheng Li, and Qingsong Wen. 2024. Logparser-llm: Advancing efficient log parsing with large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4559–4570.

[68] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.

[69] Yiwen Zhu, Rathijit Sen, Brian Kroth, Sergiy Matusevych, Andreas Mueller, Tengfei Huang, Rahul Challapalli, Weihan Tang, Xin He, Mo Liu, et al. 2025. Rockhopper: A Robust Optimizer for Spark Configuration Tuning in Production Environment. In *Companion of the 2025 International Conference on Management of Data*. 743–756.