From Chaos to Clarity: Log-based Kernel Panic Root Cause Analysis for Large-Scale Cloud Services

Tianyu Cui[1], Yang Zhang[2], Shenglin Zhang[1,*], Xin Wu[2], Yicheng Sui[1], Liangyan Peng[2], Yuhe Ji[1], Feng Wang[2], Changchang Liu[1], Zeyu Che[1], Xiaozhou Liu[2], Yongqian Sun[1], Yu Zhang[2]

[1]Nankai University, Tianjin 300071, China
[2]ByteDance, Beijing 100191, China
*Corresponding author: zhangsl@nankai.edu.cn

Abstract Operating system (OS) kernel panics, which are triggered by unrecoverable fatal errors, pose serious threats to the stability and reliability of ByteDance's large-scale cloud services. Diagnosing such failures through log analysis is essential for identifying root causes and preventing recurrence. However, root cause analysis (RCA) for kernel panics faces two key challenges. First, only a small portion of logs explicitly indicate the kernel panic, making relevant signals difficult to extract. Second, there exist complex and long-range dependencies across log entries, making it difficult to pinpoint root causes effectively. To address these challenges, we propose LogSage, a novel log-based framework for kernel panic RCA in large-scale cloud environments. LogSage combines unsupervised clustering techniques with large language models (LLMs) to extract fault-indicating log snippets, and further employs a graph-based RCA module that integrates Graph Neural Networks (GNNs) for structured log representation and active learning for efficient label utilization. We evaluate LogSage on three real-world datasets, including 20,000 failure cases from ByteDance's production environment and two publicly available industrial datasets. Experimental results show that LogSage achieves high performance in root cause identification, with F1-scores of 92.2%, 95.3%, and 96.3%, respectively. These results outperform the strongest baseline methods by 15.5%, 20.3%, and 20.1%. In addition, LogSage has been deployed in ByteDance's cloud infrastructure for over six months. It has successfully assisted engineers in real-world RCA tasks, as demonstrated through multiple case studies. These results confirm both the technical effectiveness and practical applicability of LogSage in handling kernel panic analysis in complex production settings.

Keywords Kernel panic, root cause analysis, log analysis, large language model

1 Introduction

Large-scale cloud services are increasingly prevalent, with providers managing vast infrastructures to support critical Web applications and billions of users worldwide [1]. Ensuring the reliability and scalability of these services is paramount to meeting user demands. For example, ByteDance operates extensive cloud infrastructures, where operating systems (OS) manage and process massive volumes of user data across hundreds of thousands of machines. As shown in Figure 1, These systems underpin tens of Web services, including TikTok, CapCut, Lemon8, and FaceU. However, approximately 18% of these machines experience software or hardware failures annually [2–4], with OS kernel panics being a significant contributor.

A kernel panic occurs when the OS encounters a critical error from which it cannot recover [5, 6]. During such events, the system generates detailed logs that capture low-level system activities, including hardware interactions, memory management, process scheduling, and error messages specific to the panic [7, 8]. As shown in Table 1, a kernel panic log contains diverse types of information, such as error messages, memory addresses, register values, and system states. These logs serve as a valuable resource for analyzing the root causes of kernel panics, as they provide insights into the internal state of the system at the time of the error. For instance, the log entry *"BUG: unable to handle page fault at [REDACTED]"* signals a fatal memory access violation, which is further clarified by the subsequent message *"PF: supervisor write access in kernel mode"*, indicating an illegal write attempt in privileged mode. These messages help developers trace the root cause, such as invalid memory dereferencing or access to unmapped regions, which ultimately lead to a kernel panic when the system is unable to safely recover. Identifying recurring patterns in these kernel panic logs is essential for RCA (Root Cause Analysis) and implementing effective resolutions. RCA is the process of analyzing and identifying the root cause of system failures, which facilitates accurate localization, effective repair, and system recovery [8–10]. In the context of kernel panics, RCA enables engineers to pinpoint the exact cause of a crash, facilitating targeted remediation and preventing future recurrences. Effective RCA is essential for maintaining service availability, reducing downtime, and accelerating recovery in large-scale cloud environments.

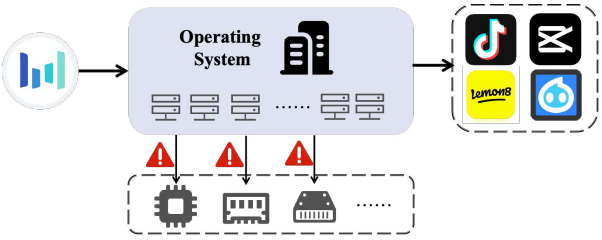

Fig. 1: System architecture of ByteDance's cloud infrastructure.

Despite their importance, performing RCA on kernel panic logs for large-scale cloud services faces two key challenges:

Challenge 1: Only a small subset of logs directly indicate kernel panics. A kernel panic often generates massive amounts of log data, yet only a small subset of these logs directly indicate the root cause of the panic. As shown in Table 1, a kernel panic case from ByteDance's production environment produced 4,445 log entries, the majority of which recorded routine system activities, such as network updates, hardware events, and process accounting. Only a small portion (lines 4,379 to 4,445, highlighted in red) contained critical panic indicators. This sparsity of relevant logs makes it challenging to identify panic-indicating entries without losing essential panic signals.

E-mail: zhangsl@nankai.edu.cn

Table 1: Kernel panic logs from ByteDance's production environment

Line	Timestamp	Content
...	...	...
524	[12.430891]	*random: dbus-daemon: uninitialized urandom read*
525	[13.314049]	*microcode: updated to revision [REDACTED]*
526	[13.315273]	*NIC Link is Up, [REDACTED] Gbps Full Duplex*
527	[13.638674]	*Process accounting resumed*
528	[13.724860]	*random: crng init done*
...	...	...
4379	[28.472077]	*BUG: unable to handle page fault at [REDACTED]*
4380	[28.472121]	*PF: supervisor write access in kernel mode*
4381	[28.472211]	*Oops: [REDACTED] SMP NOPTI*
4382	[28.472277]	*Hardware: [REDACTED], BIOS [REDACTED]*
...	...	...
4445	[28.612784]	*CR2: [REDACTED]*

Traditional methods, such as manual rules, keyword-based heuristics, or static filtering techniques, struggle to process such noisy and large-scale logs effectively. Creating manual rules is not only labor-intensive but also prone to missing key panic indicators or generating false positives when encountering new system behaviors. Furthermore, these approaches lack generalizability, as they rely heavily on predefined panic signatures, which are unable to adapt to unknown or evolving panic patterns.

Challenge 2: Long-range interdependencies in kernel panic logs. Even after extracting kernel panic-indicating logs, performing RCA remains highly challenging. Kernel panic logs often exhibit long-range interdependencies, making it difficult to identify and model dependencies between log entries. For example, as shown in Table 2, the log entry in line 38, *i.e., "memblock allocation failed"*, and the entry in line 323, *i.e., "Kernel panic not syncing: Out of memory"*, both refer to the same root cause: a fault in memory allocation that eventually leads to an out-of-memory condition, culminating in a kernel panic, although they are very distant in the extracted kernel panic-indicating logs. However, models relying solely on sequential or local semantic patterns often fail to capture these long-range interdependencies, making it difficult to pinpoint the root cause of the kernel panic. Existing log analysis methods, such as RNN-based approaches (*e.g.,* DeepLog [11] and LogAnomaly [12]) and Transformer-based approaches (*e.g.,* LogBERT [13] and NeuralLog [14]), have shown strong performance on common system logs. These logs, such as HDFS logs [15] from Amazon EC2 nodes, OpenStack logs from CloudLab [16], supercomputer logs from IBM Blue Gene [17], and application logs from Thunderbird [18], typically follow semi-structured formats with relatively simple semantic or sequential relationships. However, their reliance on sequential or semantic patterns makes them ill-suited for capturing the long-range interdependencies inherent in kernel panic logs, thereby limiting their effectiveness for RCA in this context.

Table 2: Examples of kernel panic-indicating logs.

Line	Timestamp	Content
...	...	...
38	[102.564738]	*memblock allocation failed*
...	...	...
62	[345.678901]	*eth0: transmit queue 0 timed out*
63	[346.123456]	*CPU0: soft lockup - CPU#0 stuck for [REDACTED]*
...	...	...
323	[5678.234987]	*Kernel panic not syncing: Out of memory*
324	[5679.987654]	*page fault at [REDACTED] caused by NULL pointer dereference*

To address these challenges, we present **LogSage**, a novel kernel panic RCA framework for large-scale cloud services. LogSage addresses the challenges through a two-stage process: **(1) LLM-Enhanced Fault-indicating Log Extraction (FILE)**: This stage combines clustering algorithms with LLM-based prompting techniques to extract fault-indicating logs and generate clear, interpretable fault descriptions.
(2) GraphSage-based Fault RCA (GARCA): This stage combines GraphSAGE-based framework with active learning to improve fault feature discovery and models long-range inter-log dependencies. **Our contributions are summarized as follows:**

- We design an LLM-enhanced fault-indicating log extraction mechanism to address Challenge 1 signals (**FILE**). By combining clustering-based log segmentation, LLM-based summarization, and prompt-guided highlighting, we extract critical fault-indicating logs and generate interpretable panic descriptions, effectively reducing noise and enhancing human interpretability.
- We propose an Active Learning-augmented graph-based RCA mechanism to address Challenge 2 signals (**GARCA**). By combining log-structured graph modeling, pretrained similarity estimation, GraphSAGE-based encoding, and uncertainty-driven active learning, we model long-range inter-log dependencies and enhance fault feature discovery, effectively improving RCA.
- We conduct comprehensive experiments on three real-world datasets. LogSage achieves F1-scores of 92.2%, 95.3%, and 96.3% on these datasets, respectively, outperforming the strongest baseline methods by 15.5%, 20.3%, and 20.1%. In addition, LogSage has been deployed in ByteDance's cloud infrastructure for over six months, where it has successfully assisted engineers in real-world RCA tasks.

2 Related Work

In our scenario, we frame kernel panic RCA as a classification task, where each fault type directly corresponds to a root cause category. This differs from traditional RCA settings that aim to rank multiple potential causes or extract root-cause-indicative log spans. Accordingly, we focus on methods that support categorical fault classification, we divide existing work into two categories: **Fault-Indicating Log Extraction** and **Log-based RCA**. Table 3 provides a comparative summary of representative methods.

Fault-Indicating Log Extraction. Early techniques for extracting fault-relevant log lines are predominantly heuristic. TF-IDF [19] and TextRank [20] rank logs based on frequency or centrality, but lack semantic understanding and do not support root cause classification, making them unsuitable as baselines. Onion [21] improves log extraction by combining multi-level clustering and context-aware features, but it requires normal logs for contrastive analysis, which are unavailable in our setting. SwissLog [22] localizes anomalies in interleaved logs by combining ID correlation, BERT [34]-based semantic embeddings, temporal modeling, and attention-based Bi-LSTM architectures. While effective at highlighting anomalous regions, it neither classifies fault types nor performs full RCA, and is therefore excluded. LogConfigLocalizer [23] uses an LLM-based two-stage strategy to identify misconfigured parameters through log pattern abstraction and reasoning. However, it focuses on configuration error localization rather than categorical fault diagnosis, and thus does not match our task. LoFI [24] adopts a two-stage pipeline that filters semantically relevant logs and applies prompt-based span prediction to extract failure descriptions and parameters. While

Table 3: Representative Works on Log Analysis and RCA

Method	Objective	Baseline	Reason for (Not) Being a Baseline
Fault-Indicating Log Extraction			
TF-IDF [19]/TextRank [20]	Rank log lines based on frequency or centrality	✗	Lack semantic understanding; not designed for classification or RCA
Onion [21]	Extract fault-indicating logs via clustering and contrastive analysis	✗	Requires normal logs; unavailable in our setting
SwissLog [22]	Anomaly localization using BERT-based temporal modeling	✗	Effective at region localization, but lacks classification capability
LogConfigLocalizer [23]	Identify misconfigured parameters via LLM reasoning	✗	Targets configuration issues; not fault-type classification
LoFI [24]	Extract fault descriptions via span prediction	✗	Assists diagnosis; does not predict root cause
Log-based RCA			
LogCluster [25]	Cluster log sequences and match historical cases	✓	Objective aligns with fault-type classification; adopted as a traditional baseline
Log3C [26]	RCA via KPI-correlated clustering and regression	✗	Depends on telemetry data; unavailable in our context
LADRA [27]	RCA for Spark using fixed root causes and statistical thresholds	✗	Limited to four categories; lacks generalization
LogRule [28]	Interpretable RCA via association rule mining	✓	Performs fault classification with rule-level explainability; aligned with our goals
LogKG [29]	Knowledge-graph-based symbolic RCA over structured logs	✓	Supports categorical classification via multi-field reasoning; objective aligns with ours
LogRCA [30]	PU-learning to score fault-relevant logs	✗	Produces ranked logs; does not classify root cause
LOGAN [31]	Distributed online log parsing via template extraction	✗	Acts as a preprocessor; requires normal logs; not applicable to classification
LogPrompt [32]	Zero/few-shot RCA via prompt engineering	✓	Adaptable to classification by designing RCA-specific prompts
LogGPT [33]	Prompt-based inference with GPT-3.5	✗	Covered by LogPrompt; lacks diverse strategies

helpful in surfacing fault-indicating content, it does not conduct root cause classification, and is thus not adopted as a baseline.

Log-based RCA. Supervised and unsupervised RCA methods have been extensively studied. LogCluster [25] clusters log sequences using TF-IDF-weighted vector representations, selects representative sequences, and compares them with historical cases to identify fault categories. Since LogCluster aims to classify faults into predefined types, its objective is directly aligned with ours, making it a representative traditional baseline. Log3C [26] addresses performance-impacting failures by applying cascaded clustering and correlating log clusters with system KPIs using multivariate regression. However, it relies on external telemetry data (e.g., KPI metrics), which are unavailable in our kernel-panic scenario, making it inapplicable. LADRA [27] detects and classifies anomalous Spark tasks by extracting task-level, GC-level, and CPU-related features, and assigning fault probabilities over four predefined categories (CPU, memory, disk, network) using statistical thresholds. Despite its interpretability, LADRA's narrow scope and fixed label space limit its generalization to broader RCA contexts. LogKG [29] constructs a multi-field knowledge graph (KG) by aligning semantically similar entities extracted from both structured and unstructured logs. It further introduces FOLR, a fault-oriented log representation that incorporates KG embeddings and TF-IFF patterns. As LogKG supports symbolic reasoning and categorical classification of root causes, it aligns well with our task and is adopted as a strong baseline. LogRule [28] performs interpretable RCA by mining association rules from structured logs, using item-based aggregation, disjunctive support, and semantic expansion. Its ability to produce explainable rule-based predictions aligns with our emphasis on interpretability, qualifying it as a baseline. LogRCA [30] formulates RCA as identifying a minimal subset of fault-indicating logs using PU-learning and Transformer-based scoring. Although it achieves high recall in retrieving relevant log spans, it does not generate fault type predictions and is thus excluded. LOGAN [31] is a high-throughput log parser that extracts templates from distributed log streams via dynamic LCS-based matching. While effective in parsing and anomaly localization, it assumes access to both normal and abnormal logs, and does not support root cause categorization. LogPrompt [32] leverages prompt engineering for zero/few-shot RCA, incorporating strategies such as Self-Prompt, Chain-of-Thought (CoT) [35], and In-Context Learning (ICL) [36]. Although originally not intended for RCA, we adapt LogPrompt by designing fault-type-specific prompts that enable LLMs to reason over log contexts. Its flexible prompting strategies and reasoning capabilities make it suitable as a baseline. LogGPT [33], while also based on prompt-guided inference using GPT-3.5, lacks diverse prompting strategies and RCA-oriented adaptations. As LogPrompt subsumes its capabilities and has been selected, we exclude LogGPT to avoid redundancy.

n this analysis, we select **LogCluster** [25], **LogRule** [28], **LogKG** [29], and **LogPrompt** [32] as representative baselines for evaluation.

3 Design Of LogSage

In this section, we detail the methodology of LogSage. As shown in Figure 2, the framework consists of three steps: (1) **Fault-indicating Log Extraction (FILE)** using unsupervised clustering and LLM summarization, (2) **GraphSage-based Fault RCA (GARCA)** via GraphSAGE and active learning, and (3) **Fault RCA with Trained Model**.

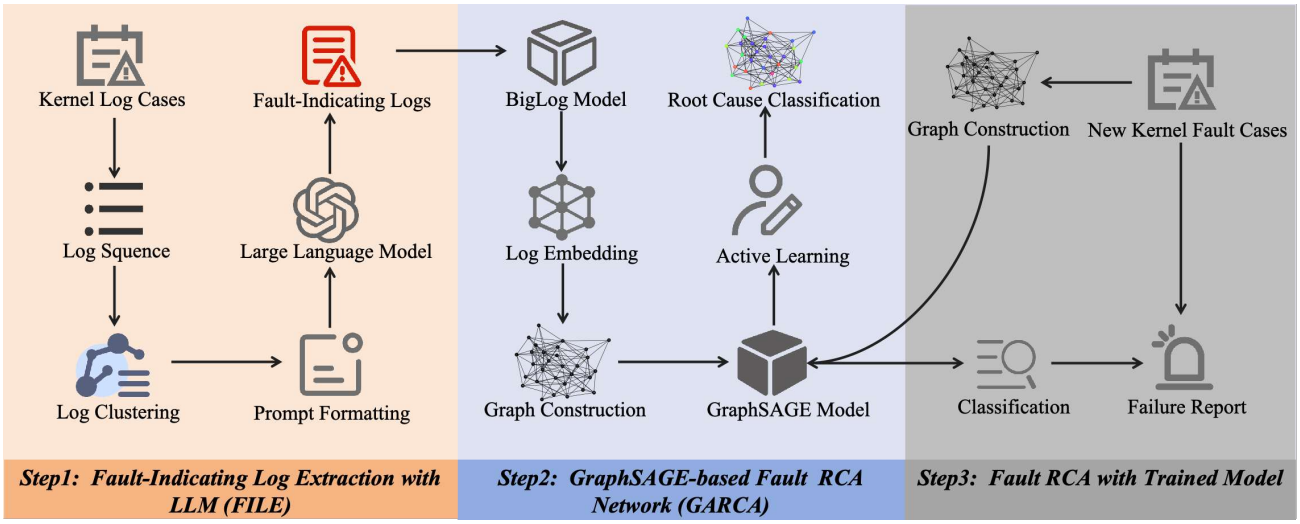

Fig. 2: The framework of LogSage.

3.1 Step 1: FILE

LogSage employs the **FILE** module to extract fault-indicative logs through two core procedures: unsupervised log clustering and LLM-based explanation.

3.1.1 Log Clustering and Ranking

To isolate semantically coherent fault-related segments from large-scale kernel logs, we first apply unsupervised clustering. Specifically, we adopt two density-based methods: **DBSCAN** [37] and **OPTICS** [38], which are well-suited for noisy and irregularly distributed logs.

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [37] groups logs by local density, defined by two parameters: the neighborhood radius ε and the minimum number of points $MinPts$ to form a cluster. It is effective in identifying compact and localized fault regions. **OPTICS** (Ordering Points To Identify the Clustering Structure) [38] generalizes DBSCAN by avoiding a fixed ε, instead producing a reachability plot that reveals hierarchical clusters under varying density thresholds. This allows detection of broader or temporally inconsistent fault patterns.

Both clustering algorithms are evaluated in our pipeline to determine which yields better RCA accuracy. After clustering, we identify the cluster with the latest timestamp—i.e., the one whose logs occur closest to the kernel panic event—based on the observation that system-critical faults are typically logged immediately before a crash. We then chronologically sort the logs within this selected cluster and retain them as fault-indicating segments. These logs are subsequently passed to the LLM-based explanation module and used for downstream RCA. Details on parameter tuning and clustering performance are reported in Section 4.

3.1.2 Log Explanation with LLM and Prompt Design

Kernel logs—particularly those triggered by panic or crash events—are typically verbose, unstructured, and filled with low-value diagnostic noise, posing challenges for both automated parsing and human analysis. To extract semantically meaningful information, LogSage leverages LLMs to summarize fault-relevant log content in a structured and concise manner.

We experiment with three open-source LLMs: **DeepSeek-V3** [39], **LLaMA 3-8B** [40], and **Mistral-7B** [41], which are accessed via different deployment modes. All three models are fully open-source and accessible for offline deployment, with strong support for structured summarization tasks. To guide the models toward consistent and interpretable outputs, we adopt a **CoT** [35] prompting strategy that decomposes the summarization task into three explicit reasoning steps. As shown in Table 4, these include (1) identifying fault clues, (2) reasoning about root causes, and (3) composing a final summary.

Table 4: CoT-style Prompt for Kernel Panic Log Explanation

Chain-of-Thought Prompt Template for Kernel Panic Explanation
Task: Given raw kernel logs involving a kernel panic event, explain the cause in a human-readable format using step-wise reasoning.
Step 1: Identify Fault Clues Scan the log to locate lines that indicate kernel panic triggers, error codes, or faulting modules. Highlight only the most relevant lines.
Step 2: Reason About the Cause Interpret the extracted fault clues. Describe what might have caused the panic, referencing relevant context if available. Avoid speculation.
Step 3: Generate Final Summary Produce a concise and coherent summary of the kernel panic, its origin, and potential impact. Maintain fidelity to the log without introducing unverifiable assumptions.

3.2 Step 2: GARCA

After extracting and structuring the fault-indicative logs, LogSage performs GARCA by integrating **GraphSAGE** for structure-aware representation learning and **active learning** for efficient label acquisition. GARCA consists of two main stages: (1) construction of a case similarity graph based on log embeddings, and (2) RCA through GraphSAGE and label-efficient model training. We describe each component in detail below.

3.2.1 Graph Construction

To capture the semantic structure among historical fault cases, we construct a case similarity graph $G = (V, E)$, where each node $v_i \in V$ corresponds to a fault-indicating log case

c_i, and edges $e_{ij} \in E$ encode the pairwise semantic similarity between cases c_i and c_j. This graph serves as the foundational input to the GraphSAGE model in the next stage.

Instead of using general-purpose embeddings, we adopt **BigLog** [42], a domain-specific language model pretrained on large-scale industrial log corpora. BigLog is designed to encode both syntactic patterns and semantic relationships that are common in system logs. Given a preprocessed log case c_i, we compute its embedding $h_i \in \mathbb{R}^d$ via:

$$h_i = \text{BigLog}(c_i) \tag{1}$$

These embeddings are used as node features in the graph and are expected to reflect both the contextual meaning and log-structural regularities required for RCA tasks.

To construct graph edges, we calculate semantic similarity between each pair of log cases using cosine similarity:

$$\text{sim}(h_i, h_j) = \frac{h_i^\top h_j}{\|h_i\| \cdot \|h_j\|} \tag{2}$$

An undirected edge is added between nodes v_i and v_j if their similarity exceeds a predefined threshold τ. The edge weight w_{ij} is proportional to their similarity, defined as:

$$w_{ij} = \begin{cases} \alpha \cdot \text{sim}(h_i, h_j), & \text{if sim}(h_i, h_j) \geqslant \tau \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Here, $\text{sim}(h_i, h_j)$ denotes the cosine similarity between node embeddings h_i and h_j. The scaling factor $\alpha \in (0, 1]$ modulates the edge strength. In our implementation, we set $\alpha = 1.0$, which retains the original cosine similarity and performs well in practice.

The threshold $\tau \in [0, 1]$ controls the sparsity of the graph and plays a crucial role in filtering out low-confidence connections. We empirically determine its value via a sensitivity analysis (see Section 4). As shown in experiments, setting $\tau = 0.6$ achieves the best trade-off between noise reduction and semantic coverage.

The resulting graph G thus captures the latent structure of historical fault cases and provides the basis for learning node-level embeddings that facilitate downstream classification and active learning.

3.2.2 GraphSAGE-Based RCA with Active Learning

To learn context-aware representations of log cases in the similarity graph $G = (V, E)$, we employ the GraphSAGE framework [43]. GraphSAGE generates node embeddings inductively by recursively aggregating features from each node's neighborhood. The representation of each node is updated layer by layer as follows.

Let $h_v^{(k)} \in \mathbb{R}^d$ denote the embedding of node v at layer k, initialized with input features $h_v^{(0)}$ from BigLog [42]. The update process at layer $(k + 1)$ includes:

1. **Neighbor aggregation:**

$$m_v^{(k+1)} = \text{AGGREGATE}^{(k)}\left(\{h_u^{(k)} : u \in \mathcal{N}(v)\}\right) \tag{4}$$

where $\mathcal{N}(v)$ denotes the first-order neighbors of v, and $\text{AGGREGATE}^{(k)}$ is a permutation-invariant function (e.g., mean or LSTM).

2. **Feature transformation:**

$$\tilde{h}_v^{(k+1)} = W^{(k)} \cdot \text{CONCAT}(h_v^{(k)}, m_v^{(k+1)}) \tag{5}$$

where $W^{(k)} \in \mathbb{R}^{d' \times 2d}$ is a trainable weight matrix, and CONCAT denotes the concatenation of the current node and its aggregated neighbor features.

3. **Non-linear activation:**

$$h_v^{(k+1)} = \sigma\left(\tilde{h}_v^{(k+1)}\right) \tag{6}$$

with σ as a non-linear activation function such as ReLU. Optional normalization may be applied to stabilize training.

After $K = 2$ layers of aggregation, we obtain the final node embedding $h_v = h_v^{(K)}$, which captures both semantic content and structural context.

To minimize annotation cost, we adopt an active learning strategy based on **prediction uncertainty**, a widely used and effective approach in graph-based semi-supervised learning. Specifically, we train a lightweight classifier (e.g., logistic regression) over the current node embeddings and compute the entropy of prediction for each unlabeled node:

$$\mathcal{H}(v) = -\sum_{c=1}^{C} p_v^{(c)} \log p_v^{(c)} \tag{7}$$

where $p_v^{(c)}$ is the predicted probability of class c. We then select the top-M nodes with the highest entropy values as the query set $\mathcal{Q}$:

$$\mathcal{Q} = \text{Top-}M\,(\mathcal{H}(v)) \tag{8}$$

Following our experiment design (Section 4), we initialize the training with **1%** randomly labeled data, and perform **5 rounds** of active learning. In each round, we query **1% of the most uncertain nodes**, resulting in a final labeled set of **6%** of the training graph. We find that RCA performance plateaus after 4–5 rounds, indicating the high label efficiency of our method.

The expert-labeled pairs $\mathcal{L} = \{(h_v, y_v)\}_{v \in \mathcal{Q}}$ are then used to retrain the classifier, which is finally applied to predict the root cause labels of all remaining unlabeled cases. This entropy-driven sampling method ensures that labeling effort is focused where it is most impactful.

3.3 Step 3: Online RCA with Graph-Augmented Inference

In the final step, LogSage performs online root cause classification for newly observed kernel panic cases using the models trained offline—specifically, the GraphSAGE encoder and the lightweight classifier trained on expert-labeled historical data.

Given a new kernel panic log, we first apply the same preprocessing and summarization pipeline as in the offline phase, resulting in a structured and semantically condensed log case. This new case is then encoded into a semantic vector using the pretrained BigLog encoder. To enable structure-aware inference, we temporarily insert the new case into the historical case graph. Specifically, we compute cosine similarity between $h_{v_{\text{new}}}^{(0)}$ and all historical case embeddings $\{h_u^{(0)}\}_{u \in V}$, and connect the new node v_{new} to its top-k most similar neighbors:

$$\mathcal{N}(v_{\text{new}}) = \text{Top-}k\left(\text{sim}(h_{v_{\text{new}}}^{(0)}, h_u^{(0)})\right) \tag{9}$$

We set $k = 10$ in our implementation. This choice is based on practical experience and prior work [43], and provides a good balance between structural coverage and computational cost. As the new cases are unlabeled in production environments, k cannot be tuned through supervised validation. We treat it as a non-critical hyperparameter and observe that LogSage remains robust to small variations in its value.

Using the augmented graph $G^{\text{new}} = (V', E')$, where $V' = V \cup \{v_{\text{new}}\}$ and $E' = E \cup E_{\text{new}}$, we apply the pretrained GraphSAGE model to compute the updated embedding of the new node by aggregating information from its neighbors:

$$h_{v_{\text{new}}}^{(K)} = \text{GraphSAGE}(v_{\text{new}}, \mathcal{N}(v_{\text{new}}), \{h_u^{(0)}\}) \tag{10}$$

Finally, this embedding is fed into the trained classifier to predict the root cause label:

$$y_{v_{\text{new}}} = \text{softmax}(W^{\text{clf}} \cdot h_{v_{\text{new}}}^{(K)} + b) \tag{11}$$

This graph-augmented inference process enables efficient and accurate RCA in deployment. By leveraging both semantic similarity and structural context from historical faults, LogSage delivers consistent and explainable predictions for previously unseen kernel panic cases.

4 Experiments

In this section, we evaluate LogSage by addressing the following research questions (RQs):

- **RQ1:** How does LogSage perform in kernel panic RCA compared to the state-of-the-art methods?
- **RQ2:** How Do Different LLMs Affect LogSage 's Performance on the RCA Task?
- **RQ3:** How does each component of LogSage contribute to the overall RCA results?
- **RQ4:** How do key hyperparameters affect LogSage 's performance in RCA tasks?

4.1 Experimental Design

4.1.1 Datasets

We evaluate LogSage using three real-world datasets:

- **Dataset 1**: Sourced from the ByteDance production environment. It contains 20,000 kernel panic incidents across 32 fault cases, accompanied by over 43.26 million raw logs.
- **Dataset 2**[1)]: Collected from the Alibaba Cloud Tianchi platform, which records data from production-deployed servers. This dataset includes 2,671 fault cases accumulated over several weeks.
- **Dataset 3**[2)]: Sourced from an OpenStack-based system deployed by China Mobile, which operates on a large-scale 4G/5G core network infrastructure. The dataset contains 93 fault cases collected over a 24-day observation period.

4.1.2 Implementation Details and Environment

LogSage is implemented in Python 3.8. The graph-based reasoning module is built using PyTorch and DGL, employing a multi-layer GraphSAGE [43] architecture for fault-type classification. For log summarization, we leverage a variety of LLMs. Specifically, we deploy LLaMA 3 [40] and Mistral 7B [41] locally, and access DeepSeek-V3 [39] via its API. The local models are loaded from open-source weights on Hugging Face.[3)4)]

All experiments are conducted on a high-performance server equipped with 10 NVIDIA A6000 GPUs, each with 64 GB of RAM. The full training and evaluation pipeline—including LLM-based summarization, graph construction, GraphSAGE encoding, and RCA classification—is implemented and optimized for efficient end-to-end execution.

4.1.3 Baselines

We compare LogSage against four representative baseline methods selected based on their compatibility with our RCA classification formulation. To ensure fair and consistent evaluation, all baselines are adapted to use the same training/test splits and actively labeled samples as LogSage. Below, we describe how each baseline is integrated into our evaluation pipeline.

- **LogCluster** [25]: This method performs unsupervised clustering over TF-IDF-weighted representations of log cases. To adapt it for root cause classification, we first apply clustering to all training samples and then annotate each cluster centroid using the actively labeled samples. We ensure that each cluster contains at least one labeled case, and assign the cluster's fault type based on the most representative labeled sample. During inference, each test case is assigned to its nearest cluster, and the label of that cluster is used as the prediction. This approach enables LogCluster to participate in classification-based evaluation while remaining label-efficient.
- **LogRule** [28]: LogRule discovers interpretable association rules from structured log patterns. We use the actively labeled training samples as ground truth during rule mining, filtering, and evaluation. These labels are used to select and validate high-confidence, class-specific rules. For inference, each test sample is evaluated against the mined rule set, and the most strongly matched rule determines the predicted fault category. This adaptation preserves LogRule's interpretability while aligning its supervision level with that of LogSage.
- **LogKG** [29]: LogKG builds a heterogeneous knowledge graph from structured and unstructured logs, using semantic alignment and symbolic reasoning for RCA. In our adaptation, we follow the original design by clustering the FOLR, and then annotate the resulting cluster centers using the actively labeled training samples. Each test case is assigned to its nearest cluster based on vector similarity and inherits the associated fault label. This allows LogKG to operate under the same supervision setting as LogSage and supports categorical fault classification.
- **LogPrompt** [32]: This method reformulates RCA as a prompt-based classification task using LLMs. We adapt it by designing RCA-specific prompts that incorporate few-shot examples drawn from the actively labeled training set. Each test case is inserted into the prompt context and processed using the same LLMs as in LogSage. LogPrompt receives supervision solely via these prompt examples, enabling few-shot reasoning. We ensure consistency in prompt format, model usage, and inference settings to support fair comparison.

All baseline methods are re-implemented or faithfully adapted based on their original papers or official codebases. Hyperparameters are tuned exclusively on the training split, and test labels are never accessed during training, adaptation, or prompt design. This ensures all methods are evaluated under identical data splits and supervision budgets.

[1)] https://tianchi.aliyun.com/competition/entrance/531947/information

[2)] https://github.com/SycIsDD/LogKG

[3)] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct

[4)] https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.1

Table 5: RCA performance comparison across three datasets

Method	Dataset 1				Dataset 2				Dataset 3			
	P	F	R	T	P	F	R	T	P	F	R	T
LogCluster	42.6	38.3	36.9	0.03	40.1	35.9	34.5	**0.01**	58.7	54.4	53.1	**0.01**
LogRule	65.9	61.5	60.3	**0.02**	63.2	59.8	58.1	0.02	64.0	59.2	58.6	0.02
LogKG	81.3	76.7	75.4	2.40	79.9	75.0	74.3	2.45	80.2	76.2	75.0	2.42
LogPrompt	74.7	70.1	68.9	4.26	73.5	69.3	67.6	4.24	75.6	71.0	69.2	4.25
LogSage	**92.4**	**92.2**	**91.8**	3.21	**94.1**	**95.3**	**95.9**	3.08	**96.7**	**96.3**	**95.9**	3.10

4.1.4 Evaluation Metrics

We formulate RCA as a multi-class classification task, where each kernel panic case is assigned to one of the predefined root causes. To comprehensively evaluate model performance, we adopt the following widely-used metrics:

- **Precision**: Measures the proportion of correctly predicted cases among all cases assigned to a given root cause:

$$\text{Precision}_i = \frac{TP_i}{TP_i + FP_i} \tag{12}$$

  A higher precision indicates fewer false positives for class i.
- **Recall**: Measures the proportion of correctly predicted cases among all actual cases of a given root cause:

$$\text{Recall}_i = \frac{TP_i}{TP_i + FN_i} \tag{13}$$

  A higher recall indicates better coverage of the true instances for class i.
- **Macro F1-Score**: Balances precision and recall for each root cause and computes their unweighted average:

$$\text{F1}_i = 2 \times \frac{\text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i} \tag{14}$$

$$\text{Macro F1} = \frac{1}{N} \sum_{i=1}^{N} \text{F1}_i \tag{15}$$

  where N is the total number of root cause classes. Macro F1 treats each class equally and is particularly suitable for imbalanced classification.
- **Runtime**: Measures the average time to complete RCA for a single case, including all preprocessing, embedding, and classification steps. This reflects the efficiency and deployability of each method.

To assess both classification performance and label efficiency, all metrics are reported after each round of active learning, enabling us to track how model performance evolves with incremental supervision.

4.2 RQ1: How does LogSage perform compared to baseline methods on different datasets?

To evaluate the effectiveness and robustness of LogSage, we conduct a comprehensive comparison with four representative methods—**LogCluster**, **LogRule**, **LogKG**, and **LogPrompt**—across three real-world datasets. We adopt **Precision (P)**, **Macro F1-score (F)**, **Recall (R)**, and **Runtime (T)** as evaluation metrics. Table 5 summarizes the detailed results.

LogCluster exhibits the lowest performance across all datasets. On Dataset 1, it yields 42.6% precision, 38.3 F1-score, and 36.9% recall. On Dataset 2 and Dataset 3, it performs similarly poorly, with F1-scores of 35.9 and 54.4, respectively. Its advantage lies in the fastest inference time of 0.01–0.03 seconds due to its simple heuristic-based clustering. However, it fails to capture semantic patterns. **LogRule** improves upon LogCluster with higher precision and recall, leveraging historical pattern-matching rules. On Dataset 1, it achieves 65.9% precision, 61.5 F1-score, and 60.3% recall. It maintains similar performance on the other datasets, with a stable inference time of 0.02 seconds. Despite its efficiency, LogRule suffers from poor generalization to unseen failure patterns. **LogKG** enhances log understanding through domain-specific knowledge graphs. It achieves 81.3% precision, 76.7 F1-score, and 75.4% recall on Dataset 1, with comparable improvements on Dataset 2 and 3. However, its runtime increases to 2.4–2.45 seconds, which may limit scalability in low-latency environments. **LogPrompt** uses LLMs for few-shot classification. On Dataset 1, it achieves 74.7% precision, 70.1 F1-score, and 68.9% recall. On Dataset 2 and 3, F1-scores reach 69.3 and 71.0, respectively. Despite its strong semantic capacity, LogPrompt suffers from the highest latency (4.24–4.26 seconds), as it lacks structural guidance and requires repeated LLM inference. **LogSage** achieves the best overall performance across all datasets. On Dataset 1, it reaches 92.4% precision, 92.2 F1-score, and 91.8% recall. On Dataset 2, it achieves 94.1% precision, 95.3% F1-score, and 95.9% recall. On Dataset 3, it obtains 96.7% precision, 96.3% F1-score, and 95.9% recall. With an inference time of around 3.1 seconds, LogSage remains suitable for post-mortem RCA tasks where latency is not mission-critical.

In conclusion, LogSage consistently outperforms existing baselines across all evaluation metrics. Although LogSage introduces moderate inference latency, this is acceptable in typical RCA settings. Unlike real-time anomaly detection tasks that demand instant response, RCA is inherently retrospective and tolerates slightly delayed outputs. Therefore, the runtime overhead of LogSage is justified by its substantial gain in precision, recall, and interpretability, making it well-suited for practical deployment in complex and dynamic system environments.

4.3 RQ2: How Do Different LLMs Affect LogSage 's Performance on the RCA Task?

To assess the impact of different LLMs on the performance of LogSage, we compare three representative models—**DeepSeek V3** [39], **LLaMA 3** [40], and **Mistral 7B** [41]—across three real-world datasets. We evaluate each model using four metrics: Precision, Recall, F1 Score, and Inference Time. The results are visualized in Figure 3.

Overall Performance. DeepSeek V3 achieves the best performance across all datasets and metrics. On Dataset 1, it yields 92.4% precision, 91.8% recall, and 92.2 F1-score; on

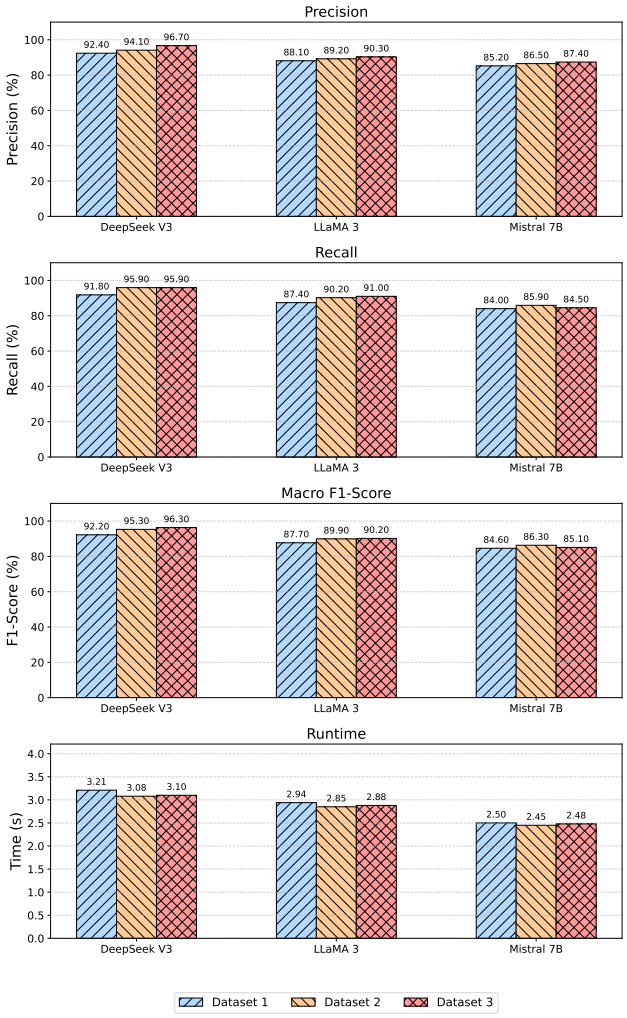

Fig. 3: Performance comparison of different LLMs.

Dataset 2, it reaches 94.1%, 95.9%, and 95.3 respectively; and on Dataset 3, it achieves the highest values—96.7% precision, 95.9% recall, and 96.3 F1. This consistent superiority reflects its strong generalization and robustness for diverse kernel panic logs. **Model Comparison.** LLaMA 3 shows stable but slightly lower results compared to DeepSeek V3. Its F1-score varies between 87.7–90.2 across datasets, and recall peaks at 91.0 on Dataset 3. In contrast, Mistral 7B achieves the lowest F1 and precision across all datasets. These results suggest that Mistral 7B is more sensitive to dataset-specific characteristics, while LLaMA 3 provides a more balanced baseline. **Inference Efficiency.** As shown in the last subplot of Figure 3, Mistral 7B achieves the fastest inference time (2.45–2.50s per case), followed by LLaMA 3 (2.85–2.94s), while DeepSeek V3 requires slightly more time (3.08–3.21s). Nonetheless, the performance improvements offered by DeepSeek V3 justify its modest overhead. Since RCA typically occurs post-failure and does not require real-time response, this level of latency remains acceptable in practical deployments.

In conclusion, DeepSeek V3 consistently achieves the best performance across all evaluation metrics and datasets, demonstrating its superior capability. Despite incurring slightly higher inference time compared to other models, its accuracy and robustness significantly outweigh this cost. Therefore, we adopt DeepSeek V3 as the default LLM backbone in all subsequent experiments, including ablation studies and component evaluations. This ensures a stable and high-performing foundation for analyzing the effectiveness of LogSage.

4.4 RQ3: How does each component of LogSage contribute to the overall RCA results?

To evaluate the contribution of each module in LogSage, we conduct ablation studies on its two core components: FILE and GARCA.

4.4.1 Ablation on FILE

We compare three variants: (1) directly using raw logs as input (w/o FILE), (2) using FILE without LLM summarization (FILE (No LLM)), and (3) the complete FILE module (FILE (Full)). Results across three datasets are shown in Table 6.

Among the three variants, w/o FILE consistently yields the lowest performance. Precision ranges from 40.1% to 58.7%, and recall is generally below 63.1%. This poor performance highlights the challenges of using raw kernel logs directly: the presence of noisy, redundant, or irrelevant lines significantly weakens the discriminative power of downstream classifiers. FILE (No LLM) shows moderate improvement by isolating more relevant log fragments via clustering. It improves F1-score by 10 points over the raw log, reaching up to 71.5% on Dataset 2. This demonstrates that density-based clustering helps extract fault-relevant regions, but the lack of abstraction still limits its semantic clarity. FILE (Full) achieves the best performance across all metrics and datasets. It reaches 92.4%/94.1%/96.7% precision and 92.3%/95.3%/96.3% f1 on Dataset 1/2/3 respectively. These results confirm the effectiveness of combining clustering with LLM summarization for extracting semantically rich and compact log representations. In terms of efficiency, FILE (No LLM) is fastest (0.03–0.05s per case), while FILE (Full) takes approximately 3.1s due to LLM inference. Despite this overhead, the substantial performance gains justify the latency in post-mortem RCA scenarios where accuracy is typically prioritized.

Table 6: Ablation Study on FILE Module

Dataset	Variant	Precision	F1	Recall	Time
Dataset 1	w/o FILE	42.6	60.1	59.4	0.06
	FILE (No LLM)	69.3	70.8	72.1	**0.05**
	FILE (Full)	**92.4**	**92.2**	**91.8**	3.21
Dataset 2	w/o FILE	40.1	62.9	63.1	0.04
	FILE (No LLM)	70.1	71.5	73.0	**0.03**
	FILE (Full)	**94.1**	**95.3**	**95.9**	3.08
Dataset 3	w/o FILE	58.7	56.6	54.8	0.04
	FILE (No LLM)	66.8	67.4	68.2	**0.03**
	FILE (Full)	**96.7**	**96.3**	**95.9**	3.10

4.4.2 Ablation on GARCA

To investigate the contribution of the GARCA module, we compare three configurations: (1) removing graph reasoning and using average-pooled BigLog embeddings directly for classification (w/o GARCA), (2) retaining graph reasoning but replacing BigLog with a general-purpose BERT encoder (GARCA (BERT)), and (3) using the full GARCA design with structure-aware modeling and domain-specific embeddings (GARCA (Full)). The results are reported in Table 7.

Among the three variants, w/o GARCA performs the worst across all datasets, with F1-scores below 69%. For example, on Dataset 1, it yields 65.1% precision, 66.7 F1, and 64.9% recall.

This confirms that simply averaging log embeddings fails to capture inter-case dependencies or latent patterns. GARCA (BERT) introduces graph reasoning but lacks domain-aware encoding. It improves F1 by 7–9 points compared to w/o GARCA, reaching 75.2 on Dataset 2. However, the generic language representations still limit its ability to distinguish fine-grained root causes. GARCA (Full) consistently achieves the best performance. It reaches 92.4%, 94.1%, and 96.7% precision on the three datasets, and achieves 92.2%, 95.3%, and 96.3% F1-scores. This demonstrates the effectiveness of combining structure-aware GNN reasoning with log-specific semantic encoding from BigLog.In terms of efficiency, GARCA (BERT) is slightly faster due to lighter encoding, but the runtime difference is marginal (e.g., 3.21s vs 2.98s on Dataset 1). Overall, the significant improvement in precision and F1-score validates the design choice of GARCA (Full) for robust RCA performance.

Table 7: Ablation Study on GARCA Module

Dataset	Variant	Precision	F1	Recall	Time
Dataset 1	w/o GARCA	65.1	66.7	64.9	3.14
	GARCA (BERT)	73.2	74.4	72.8	**2.98**
	GARCA (Full)	**92.4**	**92.2**	**91.8**	3.21
Dataset 2	w/o GARCA	67.4	68.2	66.5	**2.83**
	GARCA (BERT)	74.5	75.2	73.6	2.95
	GARCA (Full)	**94.1**	**95.3**	**95.9**	3.08
Dataset 3	w/o GARCA	62.9	63.5	61.8	**2.72**
	GARCA (BERT)	71.1	72.0	70.2	3.06
	GARCA (Full)	**96.7**	**96.3**	**95.9**	3.10

The results demonstrate the necessity of both modules. FILE enhances the signal-to-noise ratio of logs through clustering and abstraction, enabling effective summarization of fault-related behavior. GARCA further models case relationships and propagation patterns via graph-based reasoning. Their joint design achieves state-of-the-art RCA performance with acceptable runtime overhead.

4.5 RQ4: How do key hyperparameters affect LogSage 's performance in RCA tasks?

To assess the robustness and adaptability of LogSage, we conduct sensitivity analysis on two critical components:The impact of different clustering methods and the influence of key hyperparameters in the GraphSAGE model.

4.5.1 Impact of Clustering Methods

To assess the impact of different clustering strategies, we compare **OPTICS** and **DBSCAN** under identical downstream settings. Figure 4 presents the RCA performance across three datasets in terms of precision, recall, F1-score, and runtime.

Performance Comparison. DBSCAN consistently outperforms OPTICS in all datasets and evaluation metrics. For example, on Dataset 1, DBSCAN achieves 92.4% precision, 91.8% recall, and F1-score of 92.2, compared to 88.5%, 87.7%, and 88.0 with OPTICS. A similar trend is observed in Dataset 2 and Dataset 3. These results indicate that DBSCAN is better suited in capturing compact and semantically aligned clusters. **Efficiency.** The runtime of both methods is nearly identical across datasets. For instance, on Dataset 1, OPTICS and DBSCAN take 3.23s and 3.21s respectively; in Dataset 3, the difference is 3.11s vs. 3.10s. This indicates that DBSCAN's performance gain comes with no significant cost in efficiency.

In conclusion, DBSCAN emerges as the more effective clustering algorithm in LogSage. It consistently yields higher precision and F1-score, making it a better default for RCA pipelines.

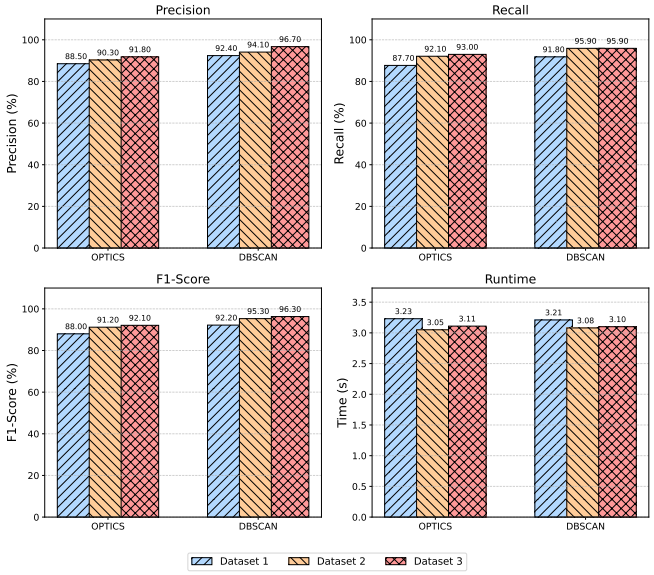

Fig. 4: Performance of different clustering methods on RCA

4.5.2 Impact of Similarity Threshold τ on Graph Construction

We study the impact of the similarity threshold τ on the construction of the case similarity graph. This threshold determines which edges are retained between log cases: higher values retain only highly similar neighbors, while lower values admit weaker (and potentially noisy) connections.

As shown in Figure 5, RCA performance improves steadily as τ increases from 0.3 to 0.6 across all three datasets. For example, in Dataset 1, the F1-score rises from 83.0% at $\tau = 0.3$ to 92.2% at $\tau = 0.6$. A similar trend is observed in Dataset 2 (from 83.5% to 95.3%) and Dataset 3 (from 81.5% to 96.3%). However, further increasing τ beyond 0.6 leads to performance degradation. This is because overly strict thresholds prune out helpful neighbors, reducing the effectiveness of graph-based reasoning.

In conclusion, $\tau = 0.6$ offers the best trade-off between noise filtering and semantic coverage, and is thus selected as the default threshold in LogSage.

4.5.3 Impact of Active Learning Rounds

We analyze how the number of active learning rounds affects RCA performance. Starting with an initial labeled set of 1%, we conduct 5 rounds of uncertainty-based sampling, where each round adds 1% of informative samples. The test set remains fixed across all rounds.

As shown in Table 8, RCA performance improves substantially during the first four rounds. On all datasets, the F1-score and recall grow steadily, while precision also rises before stabilizing. Notably, round 4 yields the best overall results, with F1-scores exceeding 95% on Dataset 2 and Dataset 3. The marginal gains in round 5 (less than 0.3%) indicate that the majority of informative cases are already captured, and additional annotation has diminishing returns.

These results suggest that LogSage achieves strong RCA performance after only 4 rounds of active learning, using just 5% of labeled data. This confirms the label efficiency of the framework, and supports our choice of 5 rounds as a good

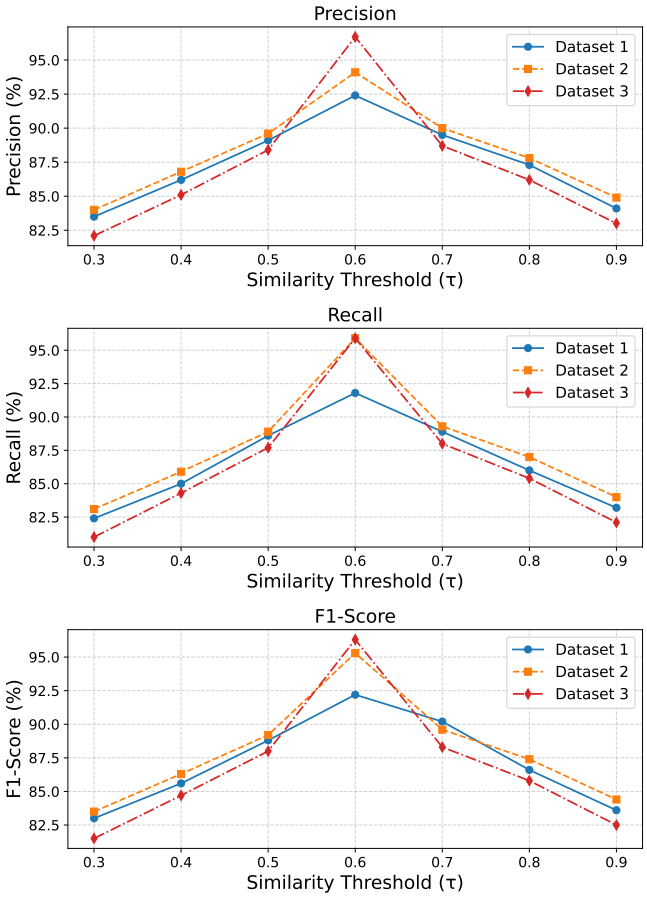

Fig. 5: Impact of similarity threshold (τ).

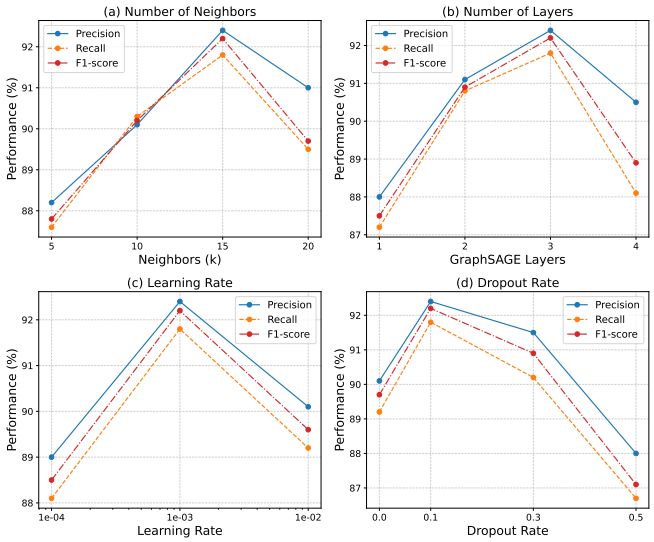

Fig. 6: Sensitivity of GraphSAGE's hyperparameters

balance between cost and performance.

Table 8: RCA performance across different active learning rounds

Dataset	Metric	Round					
		0	1	2	3	4	5
Dataset 1	P	75.4	83.7	88.5	91.1	**92.4**	92.3
	F	72.0	81.6	87.2	89.8	**92.2**	91.1
	R	70.6	80.5	86.4	89.3	**91.8**	91.0
Dataset 2	P	76.8	85.4	89.8	92.7	**94.1**	93.9
	F	74.1	84.2	88.9	91.5	**95.3**	94.9
	R	73.2	83.5	88.2	91.0	**95.9**	95.6
Dataset 3	P	74.5	84.1	89.2	92.1	**96.7**	94.5
	F	71.5	82.0	88.0	91.0	**96.3**	95.6
	R	70.3	81.3	87.3	90.6	**95.9**	95.2

4.5.4 GraphSAGE Hyperparameter Sensitivity

To evaluate the robustness of LogSage's graph-based reasoning module, we investigate the impact of four key hyperparameters in GraphSAGE: the number of neighbors, number of layers, learning rate, and dropout rate, each visualized in Figure 6.

Number of Neighbors. As shown in Figure 6(a), increasing the number of neighbors (k) from 5 to 15 leads to consistent improvements across all metrics. Performance peaks at $k = 15$, where F1-score reaches 92.2%. Beyond this point, a decline is observed due to noisy neighbor aggregation. **Number of Layers.** In Figure 6(b), the model performs best with 3 GraphSAGE layers, achieving a maximum F1-score of 92.2%. Adding a fourth layer results in performance degradation, likely due to over-smoothing effects that dilute discriminative information in deeper GNNs. **Learning Rate.** According to Figure 6(c), a learning rate of 10^{-3} provides the best stability and performance, achieving 92.4% precision and 92.2% F1-score. Both a smaller rate (10^{-4}) and a larger one (10^{-2}) result in suboptimal learning behavior. **Dropout Rate.** As illustrated in Figure 6(d), the best performance is achieved with a dropout rate of 0.1, balancing regularization and representation capacity. A high dropout of 0.5 leads to underfitting and significantly reduces performance across all metrics.

In conclusion, LogSage shows robust performance across a reasonable hyperparameter range. The optimal configuration used in our experiments is: $k = 15$, 3 layers, learning rate = 10^{-3}, and dropout rate = 0.1.

5 Deployment and Case Study

In ByteDance, the System Technologies and Engineering (STE) team is responsible for ensuring the reliability and stability of large-scale cloud services. To assist operations engineers in diagnosing kernel panics, we developed LogSage, which has been successfully integrated into the STE team's production RCA workflow, significantly improving diagnostic accuracy and operational efficiency.

5.1 Deployment Workflow

LogSage has been deployed in ByteDance's production environment to support the analysis of kernel panic incidents over the past six months. As illustrated in Figure 7, the deployment process consists of three key stages:

(1) **Kernel Panic Log Collection:** Logs are continuously ingested from production nodes via Kafka, RESTful APIs, and internal monitoring agents, covering a broad spectrum of kernel failure events.

(2) **RCA Processing:** The incoming logs are processed through the LogSage pipeline. The **FILE** module first condenses raw logs through filtering and summarization, followed by the **GARCA** module, which performs semantic reasoning and root cause classification.

(3) **Report Generation:** LogSage produces structured reports detailing the nature of the kernel panic, likely root causes, and potential mitigation strategies, which are consumed by operators through internal tools.

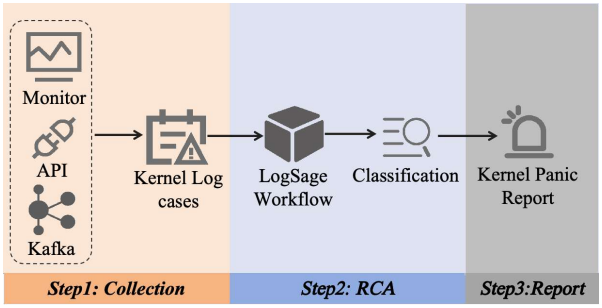

Fig. 7: Deployment workflow of LogSage in ByteDance

5.1.1 Production RCA Results

To evaluate LogSage 's real-world effectiveness, we analyzed a large batch of annotated kernel panic cases collected in the deployment period. These cases span five common failure categories identified by STE engineers: *memory leak*, *CPU overload*, *driver failure*, *hardware fault*, and *null pointer exception*.

As shown in Figure 8, LogSage achieves strong classification performance across all categories. Precision ranges from 90.2% to 94.5%, recall from 90.3% to 94.0%, and F1-score from 90.0% to 93.8%. Notably, the model performs best on *CPU overload* (F1 = 93.8%) and *driver failure* (Precision = 94.5%), suggesting its strength in identifying both resource-related and device-level anomalies.

In terms of efficiency, the end-to-end runtime remains under 3.2 seconds for all root causes, confirming LogSage 's feasibility for operational use. This balance of accuracy and latency ensures that RCA results can be delivered in a timely manner for most failure recovery scenarios.

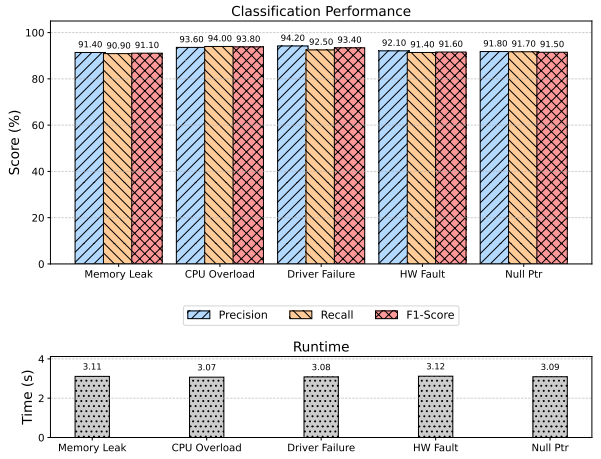

Fig. 8: RCA results on production kernel panics.

5.2 Case Study

To further demonstrate the utility of LogSage in complex scenarios, we present a representative kernel panic case collected from a production server. The failure was ultimately traced to a hardware fault, and the original log file consisted of 3162 lines.

As shown in Figure 9, we compare LogSage against the strongest baseline (*Cloud19*) in this setting. While Cloud19 processes the full raw log, it struggles to distinguish signal from noise due to the large volume of irrelevant entries. As a result, it misidentifies the cause as a driver failure.

In contrast, LogSage first applies log clustering and relevance ranking to extract the top 48 lines most temporally and semantically aligned with the panic trigger. Then, a large language model summarizes these condensed logs to highlight critical kernel error patterns. This process enables LogSage to correctly identify the hardware failure and provide a human-readable RCA summary.

This case highlights how LogSage 's two-stage pipeline—log filtering and LLM-guided reasoning—not only reduces noise but also enhances interpretability, helping engineers quickly understand complex failure contexts and take corrective actions.

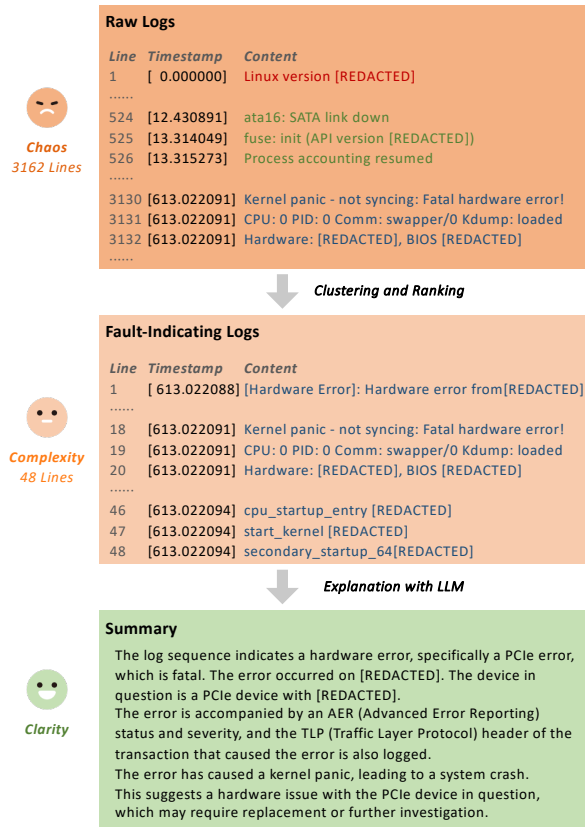

Fig. 9: Case study: RCA comparison on a hardware fault.

6 Conclusion and Future Work

In this paper, we introduced LogSage, a novel and practical framework for RCA of kernel panics in large-scale cloud services. LogSage seamlessly integrates log clustering, LLMs, and a GraphSAGE-based RCA network to enable effective fault log extraction and long-range dependency modeling. This combination allows LogSage to handle the inherent noise, heterogeneity, and complexity of kernel panic logs in real-world systems. Deployed in ByteDance's production environment, LogSage significantly reduces manual effort in RCA tasks and achieves high performance, with an accuracy of 0.91 and a macro F1-score of 0.86. Moreover, the system consistently delivers strong results across multiple fault types, including both frequent and rare kernel panic scenarios. These outcomes underscore its robustness, scalability, and practical value in industrial cloud infrastructure.

In future work, we plan to extend LogSage in several directions. First, we aim to evaluate its effectiveness in additional production environments across diverse hardware and software stacks to further validate its generalizability. Second, we will explore integrating online learning techniques to adapt to evolving system behaviors and newly emerging fault

patterns. Lastly, we envision combining LogSage with real-time alerting systems and feedback loops to support continuous RCA improvement and autonomous incident management. Overall, LogSage represents a promising step toward intelligent, automated RCA for critical system failures in modern cloud platforms.

References

1. Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An intelligent, End-To-End analytics service for safe deployment in Large-Scale cloud infrastructure. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 389–402, Santa Clara, CA, February 2020. USENIX Association.
2. Yulun Ma and Yue Hu. Business model innovation and experimentation in transforming economies: Bytedance and tiktok. *Management and Organization Review*, 17(2):382–388, 2021.
3. Xin He, Keyu Hua, Chen Ji, Haichuan Lin, Zhengqi Ren, and Wenyu Zhang. Overview on the growth and development of tiktok's globalization. In *2021 3rd International Conference on Economic Management and Cultural Industry (ICEMCI 2021)*, pages 666–673. Atlantis Press, 2021.
4. Ya Su, Youjian Zhao, Ming Sun, Shenglin Zhang, Xidao Wen, Yongsu Zhang, Xian Liu, Xiaozhou Liu, Junliang Tang, Wenfei Wu, et al. Detecting outlier machine instances through gaussian mixture variational autoencoder with one dimensional cnn. *IEEE Transactions on Computers*, 71(4):892–905, 2021.
5. Sanan Hasanov, Stefan Nagy, and Paul Gazzillo. A little goes a long way: Tuning configuration selection for continuous kernel fuzzing. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 521–533. IEEE Computer Society, 2024.
6. Zhaofeng Li, Vikram Narayanan, Xiangdong Chen, Jerry Zhang, and Anton Burtsev. Rust for linux: Understanding the security impact of rust in the linux kernel. In *2024 Annual Computer Security Applications Conference (ACSAC)*, pages 548–562, 2024.
7. Peiluan Li, Changjin Xu, Muhammad Farman, Ali Akgul, and Yicheng Pang. Qualitative and stability analysis with lyapunov function of emotion panic spreading model insight of fractional operator. *Fractals*, 32(02):2440011, 2024.
8. Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2021.
9. Hamzeh Zawawy, Kostas Kontogiannis, and John Mylopoulos. Log filtering and interpretation for root cause analysis. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5. IEEE, 2010.
10. Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 419–429. IEEE, 2021.
11. Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1285–1298, New York, NY, USA, 2017. Association for Computing Machinery.
12. Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. Loganomaly: unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, IJCAI'19, page 4739–4745. AAAI Press, 2019.
13. Haixuan Guo, Shuhan Yuan, and Xintao Wu. Logbert: Log anomaly detection via bert. In *2021 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2021.
14. Van-Hoang Le and Hongyu Zhang. Log-based anomaly detection without log parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 492–504. IEEE, 2021.
15. Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Online system problem detection by mining patterns of console logs. In *2009 ninth IEEE international conference on data mining*, pages 588–597. IEEE, 2009.
16. Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.
17. Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*, pages 575–584. IEEE, 2007.
18. Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Largescale system problem detection by mining console logs. In *Proceedings of SOSP*, volume 9, pages 1–17. Citeseer, 2009.
19. Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 29–48. Citeseer, 2003.
20. Rada Mihalcea and Paul Tarau. Textrank: Bringing order into text. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, pages 404–411, 2004.
21. Xu Zhang, Yong Xu, Si Qin, Shilin He, Bo Qiao, Ze Li, Hongyu Zhang, Xukun Li, Yingnong Dang, Qingwei Lin, Murali Chintalapati, Saravanakumar Rajmohan, and Dongmei Zhang. Onion: identifying incident-indicating logs for cloud systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1253–1263, New York, NY, USA, 2021. Association for Computing Machinery.
22. Xiaoyun Li, Pengfei Chen, Linxiao Jing, Zilong He, and Guangba Yu. Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 92–103, 2020.
23. Shiwen Shan, Yintong Huo, Yuxin Su, Yichen Li, Dan Li, and Zibin Zheng. Face it yourselves: An llm-based two-stage strategy to localize configuration errors via logs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software*

Testing and Analysis, ISSTA 2024, page 13–25, New York, NY, USA, 2024. Association for Computing Machinery.

24. Junjie Huang, Zhihan Jiang, Jinyang Liu, Yintong Huo, Jiazhen Gu, Zhuangbin Chen, Cong Feng, Hui Dong, Zengyin Yang, and Michael R. Lyu. Demystifying and extracting fault-indicating information from logs for failure diagnosis. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 511–522, 2024.
25. Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102–111, 2016.
26. Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 60–70, 2018.
27. Siyang Lu, BingBing Rao, Xiang Wei, Byungchul Tak, Long Wang, and Liqiang Wang. Log-based abnormal task detection and root cause analysis for spark. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 389–396, 2017.
28. Paolo Notaro, Soroush Haeri, Jorge Cardoso, and Michael Gerndt. Logrule: Efficient structured log mining for root cause analysis. *IEEE Transactions on Network and Service Management*, 20(4):4231–4243, 2023.
29. Yicheng Sui, Yuzhe Zhang, Jianjun Sun, Ting Xu, Shenglin Zhang, Zhengdan Li, Yongqian Sun, Fangrui Guo, Junyu Shen, Yuzhi Zhang, Dan Pei, Xiao Yang, and Li Yu. Logkg: Log failure diagnosis through knowledge graph. *IEEE Transactions on Services Computing*, 16(5):3493–3507, 2023.
30. Thorsten Wittkopp, Philipp Wiesner, and Odej Kao. Logrca: Log-based root cause analysis for distributed services. In *European Conference on Parallel Processing*, pages 362–376. Springer, 2024.
31. Byung Chul Tak, Shu Tao, Lin Yang, Chao Zhu, and Yaoping Ruan. Logan: Problem diagnosis in the cloud using log-based reference models. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 62–67, 2016.
32. Yilun Liu, Shimin Tao, Weibin Meng, Feiyu Yao, Xiaofeng Zhao, and Hao Yang. Logprompt: Prompt engineering towards zero-shot and interpretable log analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 364–365, New York, NY, USA, 2024. Association for Computing Machinery.
33. Wenxiang Jiao, Wenxuan Wang, Jen-Tse Huang, Xing Wang, and Zhaopeng Tu. Is chatgpt a good translator? a preliminary study. *ArXiv*, abs/2301.08745, 2023.
34. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.
35. Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
36. Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.
37. Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.
38. Mihael Ankerst, M Breunig, Hans-Peter Kriegel, R Ng, and J Sander. Ordering points to identify the clustering structure. In *Proc. Acm Sigmod*, volume 99, 2008.
39. Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.
40. Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
41. Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.
42. Shimin Tao, Yilun Liu, Weibin Meng, Zuomin Ren, Hao Yang, Xun Chen, Liang Zhang, Yuming Xie, Chang Su, Xiaosong Oiao, et al. Biglog: Unsupervised large-scale pre-training for a unified log representation. In *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*, pages 1–11. IEEE, 2023.
43. William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.