LLM4Log: A Systematic Review of Large Language Model-based Log Analysis

ZEYANG MA, Software PErformance, Analysis, and Reliability (SPEAR) lab, Concordia University, Canada
JINQIU YANG, Department of Computer Science and Software Engineering, Concordia University, Canada
TSE-HSUN (PETER) CHEN, Software PErformance, Analysis, and Reliability (SPEAR) lab, Concordia University, Canada

Software systems generate massive, evolving, semi-structured logs that are central to reliability engineering and AIOps, yet difficult to analyze at scale under drift and limited labels. Recent advances in pretrained Transformer models and instruction-tuned large language models (LLMs) have reshaped log analysis by enabling semantic generalization and cross-source evidence integration, but also introducing deployment risks such as context limits, latency/cost, privacy constraints, and hallucinations. This paper presents **LLM4Log**, a systematic review of LLM-based log analysis across the end-to-end pipeline, from upstream logging-statement generation and maintenance to log parsing/structuring and downstream tasks including anomaly detection, failure prediction, root cause analysis, and log summarization. Following a structured search and manual screening protocol, we completed literature collection in November 2025 and identified 145 unique papers across seven logging tasks. We synthesize the research area through a unified, task-driven taxonomy, summarize common design patterns (prompting/ICL, retrieval grounding, fine-tuning, tool/agent augmentation, and verification), and analyze evaluation practices, datasets, metrics, and reproducibility. Based on these cross-paper analyses, we distill key lessons and open challenges for reliable real-world adoption. We emphasize robustness under drift and long-tail events, grounding and faithfulness for operator-facing outputs, and deployment-oriented designs with verifiable behavior.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Log analysis, Software logs, Large language models

1 Introduction

Software systems, from cloud services and microservices to networked and cyber-physical infrastructures, emit massive volumes of execution logs that record runtime states, events, and failures. Logs are a primary source of evidence for reliability engineering and operations: they support debugging, monitoring, incident triage, and postmortem analysis. However, modern log streams are notoriously difficult to analyze at scale. They are *semi-structured* (mixing free-form text with identifiers and code-like tokens), *high-volume/high-velocity*, and *continuously evolving* due to software updates, configuration changes, and deployment diversity. Consequently, manual inspection and keyword search remain common yet brittle, while automated pipelines often suffer from format drift, long-tail behaviors, and limited labeled data [23, 48, 49].

Authors' Contact Information: Zeyang Ma, Software PErformance, Analysis, and Reliability (SPEAR) lab, Concordia University, Montreal, Canada, m_zeyang@encs.concordia.ca; Jinqiu Yang, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, jinqiu.yang@concordia.ca; Tse-Hsun (Peter) Chen, Software PErformance, Analysis, and Reliability (SPEAR) lab, Concordia University, Montreal, Canada, peterc@encs.concordia.ca.

ACM XXXX-XXXX/2026/2-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To address these challenges, the research community has developed a rich body of work on automated log analysis, typically organized as a pipeline spanning logging practice, parsing/structuring, representation learning, anomaly/failure detection, failure prediction, diagnosis, and summarization [49]. In production settings, however, log analysis is constrained by operational realities: incidents unfold under time pressure; log streams are interleaved across services, hosts, and tenants; failures manifest through long causal chains; and the most diagnostic events are often rare, noisy, or buried in high-volume background messages. Moreover, engineers rarely rely on logs alone—effective triage requires correlating logs with metrics/traces, historical incidents, runbooks, and code/configuration context. These characteristics make log analysis not only a *pattern recognition* problem, but also an *evidence-integration* and *human decision-support* problem.

The classic log-analysis pipeline was largely shaped before instruction-tuned LLMs became widely available, when methods were either rule-based or relied on fixed representations and task-specific supervision. In recent years, there has been a rapid shift: pretrained Transformer language models, including modern LLMs, are increasingly used as the backbone for log analysis. Unlike traditional approaches that heavily depend on templates, handcrafted features, or rigid assumptions about message formats, LLMs offer semantic generalization and can condition on heterogeneous evidence (e.g., logs together with tickets, traces/metrics, runbooks, configurations, and code and configuration context), making them a natural fit for modern AIOps workflows [22, 44, 71].

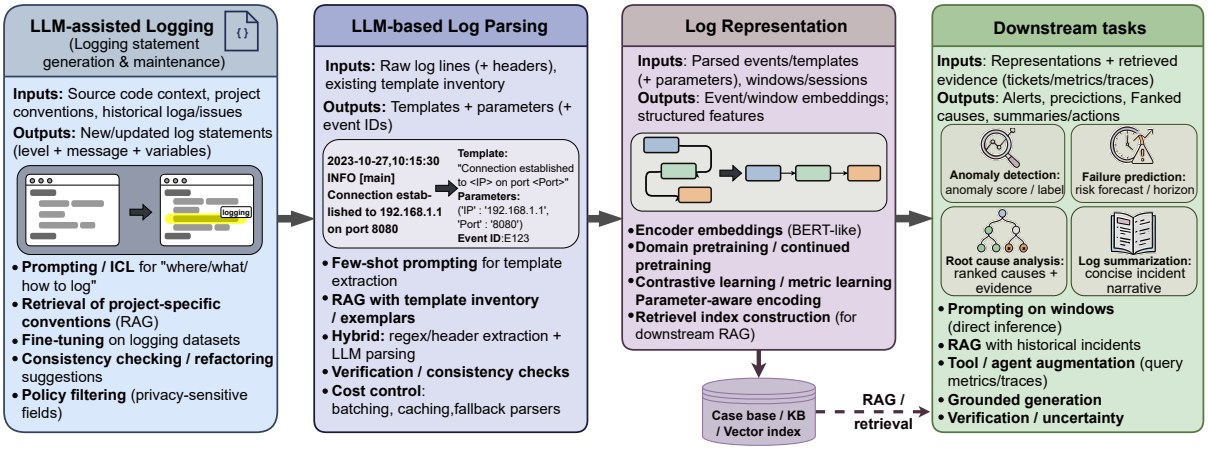

Fig. 1. LLM-based Log Analysis Across the Pipeline: From Logging to Downstream Tasks

This shift is already visible across the *entire* log-analysis pipeline. Figure 1 provides an overview of how LLMs are applied across the log-analysis pipeline. Upstream, LLMs are used to **generate and maintain logging statements**, helping developers decide where to log and what to record under project conventions. At the core of many pipelines, LLMs are applied to **log parsing**, converting raw messages into structured templates and parameters with improved robustness under drift, unseen formats, and long-tail events [6]. Between parsing and downstream inference, LLMs are also increasingly used for **log representation learning**, producing semantic embeddings or contextualized event representations for logs that improve robustness and transferability under drift. Downstream, LLMs are used for **anomaly detection** and **failure prediction**, but also increasingly for operator-facing tasks such as **root cause analysis** (RCA) and **log summarization**, where explanation quality and evidence integration are as important as prediction accuracy [22, 44]. At the same time, LLM-based log analysis introduces new trade-offs and risks that are less prominent in classic pipelines: context window limits, latency/cost constraints in online settings, privacy and data-governance concerns, sensitivity to prompting and

retrieval design, and hallucinations that can mislead incident response if outputs are not properly grounded and verified [22, 71].

Given the pace and breadth of this emerging area, practitioners and researchers need a consolidated, task-driven view of (i) *where* LLMs are applied in the log-analysis pipeline, (ii) *how* they are integrated (e.g., prompting/ICL, retrieval grounding, fine-tuning, tool/agent augmentation, and verification), (iii) *how* they are evaluated (datasets, metrics, baselines, and reproducibility), and (iv) *what* remains open for real-world deployment. While prior surveys have reviewed automated log analysis broadly [49] and recent work has summarized specific subareas such as LLM-based log parsing [6], a systematic, end-to-end synthesis that covers the full LLM4Log landscape is timely.

In this paper, we present **LLM4Log**, a systematic review of *large-language-model-based log analysis*. We conducted a structured literature collection and manual screening process, with the corpus collection completed in November 2025. Using a task-driven search and screening protocol, we identified 162 task-paper records spanning seven log-analysis tasks. Since some studies address multiple tasks, we deduplicated the task-paper records by title, resulting in 145 unique papers published between 2020 and 2025. Building on this corpus, we synthesize the field through a unified taxonomy and cross-paper analyses.

Our goal is to offer a consolidated, end-to-end reference for how LLMs are applied to log analysis, what works in practice, and what remains open. To that end, we make the following contributions:

- **A systematic, task-driven mapping of LLM4Log research (2020–2025).** We provide a curated and categorized corpus, summarize the temporal and task distribution, and characterize where the community concentrates effort and where gaps remain.
- **A pipeline-level synthesis across log analysis stages.** We review and compare LLM-based approaches for (i) logging statement generation and maintenance, (ii) log parsing, and (iii) downstream tasks (anomaly detection, failure prediction, root cause analysis, and log summarization), highlighting reusable design patterns and recurring trade-offs.
- **Cross-cutting insights and open challenges.** We distill lessons about evaluation practices (datasets, metrics, baselines), robustness under drift and long-tail events, grounding and faithfulness for operator-facing outputs, privacy/security constraints, and reproducibility, and we outline promising research directions for reliable LLM-assisted operations.

Paper organization. The rest of this paper is organized as follows. Section 2 describes our systematic survey methodology. Section 3 provides LLM background and a taxonomy of adaptation/enhancement paradigms for log analysis. Section 4 reviews LLM-based logging statement generation and maintenance. Section 5 surveys LLM-based log parsing. Section 6 covers downstream log analysis tasks, including anomaly detection, failure prediction, root cause analysis, and log summarization. Finally, Section 7 concludes the survey and discusses open problems.

2 Survey Methodology

To build the literature corpus for this survey, we conducted a structured search for publications on *LLM-based log analysis*. Our collection process followed a top-venue-first strategy: we started with keyword searches in top software engineering venues, expanded the set via backward and forward snowballing, and finally cross-checked coverage through widely used digital libraries and academic search engines. The literature collection was completed in November 2025. The complete paper list and corpus metadata are available in our online repository: https://github.com/zeyang919/LLM4Log.

2.1 Literature collection protocol

Stage 1: top-venue-first keyword search. We began with keyword-based searches in top-tier software engineering venues where log analysis and LLM-for-SE work frequently appear (e.g., ICSE, FSE, ASE, ISSTA, ICST, ICSME, ISSRE, ESEM, ICPC, TSE, TOSEM, EMSE, JSS, IST). We used combinations of *log-related* and *LLM-related* keywords. The log-related keywords include terms such as *"log", "system log", "runtime log", "log analysis", "log parsing", "log anomaly detection", "failure prediction", "root cause analysis", "incident diagnosis", "log summarization"*; the LLM-related keywords include terms such as *"large language model", "LLM", "Transformer", "BERT", "GPT", "ChatGPT", "in-context learning", "prompting", "retrieval-augmented generation", and "agent"*. We iteratively refined the keyword set as we encountered task-specific terminology and community-preferred phrasing (e.g., operational diagnosis and troubleshooting terms).

Stage 2: snowballing to extend coverage. From the seed set obtained in Stage 1, we performed backward snowballing by inspecting references and forward snowballing by checking citations to discover additional relevant papers that may not be easily retrieved via keyword queries (e.g., papers using non-standard task names, or papers emphasizing systems/operations rather than "log" in titles/abstracts). We also tracked community resources that aggregate log-analysis and LLM-for-operations literature (e.g., benchmarks, dataset/toolkit papers, and curated lists) as auxiliary entry points.

Stage 3: digital-library cross-check. To reduce omission risk and broaden coverage beyond core SE venues, we cross-checked our corpus using major digital libraries and preprint servers (i.e., IEEE Xplore, ACM Digital Library, SpringerLink, Elsevier/ScienceDirect, Wiley Online Library, arXiv, and OpenReview), and used Google Scholar for additional coverage checks and citation tracing. This step expanded the search surface and improved recall, aiming to include as many relevant papers as possible.

Scope control and manual screening. Keyword searches inevitably return false positives due to term ambiguity and task overlap. For example, some papers include both *"root cause analysis"* and *"LLM"* keywords, but their methods are not log-centric (e.g., they rely primarily on traces, metrics, code, tickets, or other modalities, with logs being absent or peripheral). Similarly, the term "log" can refer to non-runtime logs (e.g., query logs, change logs, process mining event logs) or even mathematical "log" usage. Therefore, after retrieving candidates, we manually screened each paper by reading the title, abstract, and the method/evaluation sections, and excluded papers where *software/system runtime logs* were not a primary data source for the studied method or evaluation.

Inclusion and exclusion criteria. A paper is included if it satisfies all of the following: (1) it targets one or more *log analysis* tasks (e.g., parsing/representation, anomaly/failure detection, failure prediction, diagnosis/RCA, summarization, or related operational assistance); (2) it uses a Transformer-based pretrained language model (encoder-only, encoder–decoder, or decoder-only) via prompting/ICL, retrieval grounding, fine-tuning/post-training, tool augmentation/agents, or representation learning; and (3) it reports an empirical evaluation, ablation/analysis, or a concrete case study on log data (public or proprietary). We exclude work where logs are not a primary modality (e.g., logs only as a minor illustration), work that focuses on non-runtime log types, and work outside log-centric operational analytics scope.

Data extraction protocol. For each included paper, we extracted structured information to support later taxonomy construction and cross-paper comparison. Importantly, the extracted fields depend on the *task type* of the paper. Across tasks, we recorded: (i) target task(s), (ii) dataset type and whether auxiliary artifacts are used (e.g., metrics/traces/tickets/runbooks/code/config), and (iii) evaluation setup. In addition, for each task, we explicitly extracted task-relevant technical details, such as: *baseline LLM(s), LLM enhancement/adaptation paradigm(s)* (e.g., ICL, retrieval grounding/RAG, fine-tuning, tool/agent augmentation, verification), and *evaluation metric(s)*. When a paper addressed multiple tasks, we categorized it by its primary contribution while retaining cross-references to its secondary tasks and extracted fields accordingly.

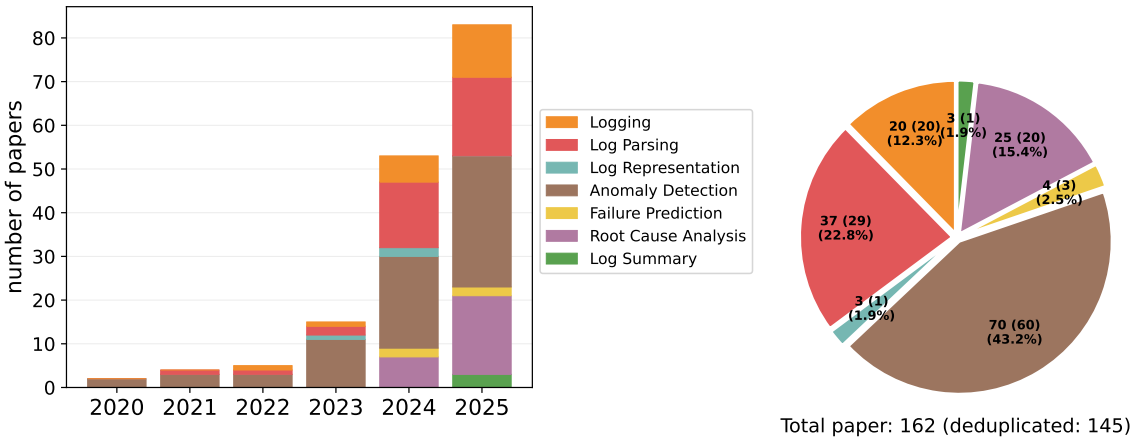

Fig. 2. Temporal and overall distribution of LLM4Log analysis papers across tasks. Note: Pie chart labels are shown as n (u), where n counts task papers (not deduplicated) and u counts unique papers (deduplicated).

2.2 Corpus summary and categorization.

Based on our task-driven screening, we identified 162 relevant studies spanning seven log-analysis tasks, published between 2020 and 2025. The literature search was completed in November 2025. Because some papers cover multiple tasks (e.g., a pipeline that both parses logs and detects anomalies), after deduplication by title, the corpus contains 145 unique papers. Notably, the pre-LLM log-analysis survey by He et al. [49] reported 158 log-analysis papers published from 1997 to 2020. In contrast, within only 2020–2025, we already observe 145 *LLM-based* log-analysis papers, ***indicating a rapid LLM-driven surge and a clear paradigm shift that motivates a dedicated LLM4Log survey***. Figure 2 summarizes the corpus from two complementary views: the *temporal trend* over years (left) and the *task composition* (right).

The task distribution (Fig. 2, right) is highly skewed. Most papers concentrate on Anomaly Detection and Log Parsing, together accounting for roughly *two-thirds* of all task-paper records, which reflects where LLMs are most immediately useful in practice: identifying suspicious behavior under noisy, drifting logs and structuring raw logs into analyzable forms. At the same time, the corpus also shows a clear expansion toward more downstream, operator-facing tasks. In particular, Root Cause Analysis forms a sizable slice and grows rapidly in the most recent years, indicating a shift from "detecting something is wrong" toward "explaining why and what to do," often requiring richer reasoning and evidence integration.

Temporally, the corpus spans 2020–2025 and exhibits a pronounced surge in recent years (Fig. 2, left). Only a small number of papers appeared before 2023, followed by a sharp jump in 2024 and an even larger volume in 2025. Beyond the overall growth, the per-task evolution is also informative: anomaly detection appears earliest and grows steadily, while several downstream tasks emerge later and accelerate quickly (e.g., RCA and log summarization are largely concentrated in the most recent period). This pattern suggests that the community first adopted LLMs where they can be plugged into existing pipelines (detection/parsing) and then progressively pushed them toward higher-level operational assistance as models and prompting/retrieval techniques matured.

Finally, the venue landscape indicates both broad dissemination and strong SE anchoring. As summarized in Table 1, a noticeable fraction of the corpus appears as preprints (reflecting the fast-moving nature of LLM research), while many peer-reviewed papers are mainly published in software engineering venues. Interestingly, the venue concentration differs by task: papers on anomaly detection are much more widely distributed across research communities and venues (e.g., systems, security, networking, data mining, and AI), whereas topics closer to software engineering practice (e.g., logging and some parsing work) are more likely to appear in SE-focused outlets. This cross-venue spread reinforces that LLM4Log is inherently interdisciplinary: it inherits operational problems from SE while drawing methods and evaluation norms from multiple adjacent fields.

Table 1. Top venues by number of papers in our LLM4Log corpus.

Venue	#Papers
arXiv	29
ICSE	11
ASE	7
ISSRE	7
FSE	6
CIKM	5
ICLR	5
EMSE	4
TOSEM	4
ICPC	3

Note: We report the top 10 venues in the paper. The full list is available in our online repository.

2.3 Relation to existing surveys.

In recent years, several surveys and reviews have summarized progress in automated log analysis and related LLM-enabled operational analytics. He et al. [49] offered a broad survey of automated log analysis for reliability engineering, summarizing established pipelines spanning logging practice, compression, parsing, anomaly detection, failure prediction, and diagnosis, and largely reflecting the pre-instruction-tuned-LLM era. As LLMs entered log analytics, more focused surveys emerged: Beck et al. [6] concentrated on *system log parsing with large language models*, emphasizing how LLMs convert raw logs into structured templates/fields and how such parsers are evaluated. In parallel, Akhtar et al. [1] reviewed *LLM-based event log analysis* across diverse log types and domains, but uses a broader notion of "event logs" and does not foreground the operational log-analysis workflow patterns that increasingly dominate practice (e.g., retrieval grounding, tool augmentation, and verification under drift and long contexts). From the operational analytics side, another recent review discussed AIOps in the LLM era [199], covering LLM-enabled reliability engineering beyond logs as a single primary modality.

Our survey complements these efforts by focusing specifically on *LLM4Log*. We treat software runtime logs as the central evidence source and synthesize how modern LLMs are integrated into log-centric workflows, together with the deployment constraints (privacy, security, and efficiency) that shape real-world adoption.

3 LLM Background & Taxonomy for Log Analysis

LLMs have become a practical tool for log-centric software operations because logs are semi-structured, drift over time, and are typically interpreted alongside heterogeneous artifacts (tickets, dashboards, traces, configurations, and code). [42, 75, 105, 110] This section first explains why LLMs are a natural fit for log tasks, then summarizes the LLM landscape (how modern LLMs evolved, what capabilities matter for log analysis, and what kinds of models are commonly used in practice), and finally reviews common LLM adaptation and enhancement paradigms, as well as practical deployment concerns.

3.1 Why LLMs are a Good Fit for Log Tasks

Logs as semi-structured text that bridges natural language and code. A key reason LLMs fit log analysis is that log messages sit between natural language and code. A typical log line mixes free-form descriptions (symptoms, actions, causal hints) with code-like structure and identifiers (function and class names, stack frames, paths, IPs, config keys, request IDs, error codes) [65, 77]. This hybrid format aligns with two core LLM strengths:

natural-language understanding and code-aware pattern modeling [26, 110]. As a result, LLMs can capture intent and semantics that are often discarded when logs are reduced to templates/event IDs, and can still remain robust when token vocabularies evolve (e.g., new components, identifiers, or new error strings) [67, 105].

Latent semantics beyond templates and surface patterns. Although many classic pipelines reduce logs to templates/event IDs, real log messages encode operational meaning: what operation was attempted, what failed, which component is implicated, how severe it is, and what the system was doing right before/after [75, 77]. These signals are frequently expressed implicitly via linguistic cues (error idioms, temporal markers, causal connectors, negation) and code-like cues (exception types, API names, call sites, module prefixes) [42, 105]. LLMs can exploit these cues to support interpretation, explanation, and cross-source synthesis, especially when purely symbolic representations are brittle or incomplete [44, 71, 101].

Robustness under drift, partial observability, and long-tail behaviors. Logs evolve with versions, configuration changes, feature flags, and deployment environments. Meanwhile, rare events are common in operations and often carry the most diagnostic value [67, 105]. Compared with fixed-vocabulary or rule-based parser-dependent pipelines, LLM-based representations and prompting enable soft matching and semantic generalization, which helps when (i) rule-based parsers fail or drift, (ii) labeled anomalies/failures are scarce, or (iii) the same underlying issue is described with heterogeneous phrasing across services and versions [4, 105, 188].

Unifying heterogeneous evidence into a single reasoning space. In production triage, the relevant evidence rarely lives in logs alone [22, 44]. Engineers correlate logs with traces/metrics, tickets, runbooks, configurations, and sometimes code diffs [22, 44]. LLMs can condition on mixed inputs (log snippets, retrieved historical cases, metric summaries, runbook steps, code/config fragments) and produce operator-oriented outputs such as hypotheses, ranked causes, incident narratives, and action candidates [44, 71, 101]. This ability to "speak one language" across heterogeneous artifacts matches the reality of AIOps workflows, where the bottleneck is often evidence integration rather than classification alone [44, 71].

Human-facing outputs and decision support. Many operational tasks require a narrative (what happened, what changed, why likely) and not just a score/label [44, 71]. LLMs can highlight evidence, translate technical fragments into concise explanations, and generate structured artifacts for handoff (e.g., timelines, suspected components, extracted entities) [22, 101, 103]. However, free-form generation increases the risk of hallucinated rationales or overconfident claims [22, 71]. In practice, LLMs are most reliable when outputs are constrained and grounded (e.g., by quoting evidence spans, using structured extraction, or tying claims to retrieved/tool-produced facts) rather than relying on unconstrained storytelling [44, 101, 103].

3.2 Evolution of LLMs: Architectures and Training Stages

From sequence models to Transformers. Earlier neural language models and representations often relied on RNN/LSTM-style recurrence [53]. While effective for short contexts, recurrence is harder to parallelize and tends to degrade on long-range dependencies [8, 125]. Transformers [171] replace recurrence with self-attention, enabling efficient scaling and much stronger contextual modeling. This shift is foundational for log workloads, where useful cues can be far apart in a window (e.g., an early warning followed by a later error burst) and where multi-line context often changes the meaning of individual events [171]. The Transformer backbone underlies most modern pretrained language models (e.g., BERT-family encoders and GPT-family decoders), making it the architectural prerequisite for today's LLM-based log analytics [12, 26]. In practice, the architectural shift also enabled orders-of-magnitude scaling in parameter count relative to earlier recurrent models, a key driver of the emergent capabilities discussed later [12, 69, 180].

Encoder-only models: masked language modeling and contextual embeddings. Encoder-only pretraining with masked language modeling (MLM), popularized by BERT [26] and its variants (e.g., RoBERTa [98],

ELECTRA [19]), learns bidirectional contextual representations. Compared with decoder-only chat-style LLMs, encoder-only models typically have fewer parameters and lower inference cost, and some log-analysis papers therefore refer to them as *pretrained language models (PLMs)* rather than LLMs; however, since they are still substantially larger than traditional log models and have a large body of log-centric research, we treat them as part of the broader LLM family in this survey [133]. In log analytics, encoder-only LMs often play a dual role [26, 42]. First, they can serve as *foundational models* via their pretraining objectives: deviations can be detected via masked-token reconstruction errors (MLM-style) or replaced-token discrimination signals (ELECTRA-style), enabling semi- or unsupervised log analysis without relying solely on a supervised classifier [19, 26, 42]. Second, they are strong *representation backbones*: embedding log events/windows for retrieval, clustering, and discriminative detection, supporting (i) similarity-based historical matching, (ii) robustness under wording variation, and (iii) lightweight downstream heads for anomaly/failure classification when some supervision exists [26, 72, 138]. They are also commonly used through sentence-level variants such as Sentence-BERT [138] when the primary goal is semantic similarity rather than token-level prediction [138].

Encoder–decoder models: conditional generation and structured transformation. Encoder–decoder Transformers generalize from representation learning to conditional generation: an encoder ingests the input sequence and a decoder produces an output sequence [161, 171]. Representative families include T5 [137] and BART [81], which are typically trained with denoising or text-to-text objectives [81, 137, 172]. In terms of scale, encoder–decoder models are often trained at moderate-to-large sizes: commonly larger than lightweight encoders used purely for embeddings, but frequently smaller than the largest decoder-only frontier models used for open-ended interactive assistance [12, 137]. This paradigm is well-suited to tasks that transform one textual form into another, such as summarization, normalization, and information extraction [81, 143]. For logs, encoder–decoder models can be used to produce incident summaries, rewrite noisy messages into canonical descriptions, or extract structured fields conditioned on surrounding context, while keeping generation anchored to an explicit input [81, 137].

Decoder-only models: autoregressive pretraining and general-purpose assistance. Decoder-only autoregressive LMs are trained to predict the next token at scale and dominate interactive settings due to their flexibility under prompting [12]. Representative examples include the GPT family [12] and open-weight decoder models such as LLaMA-style backbones [168]. These models are typically scaled to the largest parameter counts among the three families, which supports strong generalization and instruction-following behavior but also increases inference cost and latency [69, 150]. This makes them attractive when log workflows require open-ended reasoning (hypothesis generation, stepwise triage, remediation suggestions) or when labeled data is scarce [180]. Post-training (instruction tuning and alignment, often via preference optimization) further improves usability by stabilizing instruction following, enabling more consistent formats (lists, JSON-like structures), and supporting multi-turn operator interaction [18, 122, 136].

Key capabilities that matter for log tasks. Across architectures, several capabilities are particularly relevant to log analysis [11, 82, 118, 122, 181, 193]: **(1) semantic normalization and abstraction**—mapping heterogeneous phrasings to consistent meanings and grouping variants of the same issue; **(2) contextual disambiguation**—interpreting the same token differently depending on surrounding components, timestamps, and execution context (crucial under drift); **(3) evidence synthesis over long contexts**—integrating signals across many lines and multiple sources (logs, traces, metrics, tickets), typically mediated by windowing or retrieval in practice; **(4) controllable outputs**—producing operator-facing artifacts (ranked lists, extracted entities, timelines) instead of only free-form text; **(5) reasoning and decomposition**—breaking complex tasks into steps (form hypotheses, retrieve context, verify, refine), which underpins agentic log analysis pipelines; **(6) software and code awareness**—understanding stack traces, API names, configuration keys, and common failure modes, enabling more actionable interpretation than purely surface-level text matching [14].

3.3 LLM Capability and Specialization Taxonomy for Log Analytics

LLMs used in log analysis can be grouped by specialization level, which often correlates with their strength in the log pipeline [11].

General-purpose LLMs are trained broadly on large-scale, diverse text corpora and excel at natural-language interpretation, summarization, and interactive assistance [12, 118]. They are commonly used for prompt-based log understanding, RCA narratives, and incident reporting [118, 122]. However, they may struggle with domain jargon, proprietary components, and organization-specific conventions unless supplemented with context (e.g., retrieved runbooks or historical incidents) [45, 82].

Code-oriented LLMs are trained or fine-tuned heavily on code and technical corpora, improving performance on stack traces, error messages, configuration files, and program-context reasoning [14, 35, 178]. They are often a better fit when the pipeline requires linking logs to code/config, generating remediation commands, or interpreting call stacks and error propagation paths [14].

Domain-tailored LLMs adapt a backbone to a specific operational domain (e.g., AIOps for cloud platforms, network diagnostics, ICS security) via continued pretraining, instruction tuning, or explicit knowledge integration [45, 122]. The goal is to internalize domain terminology and conventions, reduce spurious generalization by narrowing scope, and improve reliability on real-world log distributions [45]. These models are particularly valuable when logs contain specialized vocabularies and when organizations rely on internal documentation and recurring operational procedures [45].

3.4 Adaptation and Enhancement Paradigms

Off-the-shelf LLMs are often not trained *specifically* for log analysis: they rarely see an organization's proprietary log vocabulary, evolving component taxonomy, or operational ground-truth data (incidents, runbooks, and on-call decisions) [11, 45]. Therefore, LLM-based log analysis approaches typically improve the reliability and utility of LLMs by controlling *what* the model sees (context, retrieved evidence) and *how* it decides (prompting, reasoning scaffolds, tuning, tools, and verification) [82, 122, 181, 193]. Below, we summarize the most common paradigms and the design choices that matter in log settings [11].

Prompting and in-context learning (ICL), including retrieval grounding. ICL treats the LLM as a training-free learner: prompts provide task instructions, output constraints, and a small set of demonstrations [12, 94, 115]. In log settings, effective ICL is rarely a single "classify this log" instruction; it typically includes: (i) **input normalization** (variable masking, deduplication, component/time grouping), (ii) **bounded evidence** (top-k suspicious lines/windows rather than full streams), (iii) **schema-constrained outputs** (labels, ranked candidates, JSON fields), and (iv) **system-specific exemplars** (a few representative normal/failure patterns from the same environment) [94, 115].

A common extension is **retrieval-augmented generation (RAG)**, which retrieves relevant context (e.g., similar historical windows/incidents, runbook passages, known signatures, or indexed summaries) and injects it into the prompt as additional demonstrations or evidence [82]. In many log-analysis pipelines, RAG can be viewed as an *ICL enabler* rather than a separate paradigm: retrieval supplies high-quality in-context examples and system knowledge at inference time, without changing model weights [82]. This "retrieve → prompt → decide" pattern improves both *grounding* (anchoring outputs to concrete evidence) and *adaptation* (bringing system-specific knowledge under drift) [45, 82].

ICL remains attractive for rapid deployment and heterogeneous tasks, but it is sensitive to prompt quality, context selection, and distribution shift; consequently, many practical approaches combine RAG-grounded ICL with lightweight filtering (to reduce prompt noise) and downstream verification (to prevent overconfident hallucinations) [94, 115].

Reasoning scaffolds: CoT, ToT, and structured decomposition. Many log tasks (especially diagnosis and remediation) benefit from explicit decomposition rather than one-shot generation [181, 215]. **Chain-of-Thought (CoT)** prompting encourages stepwise reasoning (e.g., identify key symptoms → map to components → hypothesize causes → justify with evidence) [181]. In log settings, CoT is most useful when intermediate artifacts are *checkable* and *grounded* (e.g., extracted entities, cited log lines, matched templates), not when it produces purely speculative narratives [176, 181]. **Tree-of-Thought (ToT)** generalizes CoT by exploring multiple candidate hypotheses/paths, scoring or pruning them, and selecting the best [176, 194]. This matches operational diagnosis where multiple plausible causes exist; ToT-style search can be implemented via self-consistency sampling, explicit branching (e.g., per service/component), and re-ranking using retrieved evidence or tool outputs [176, 194]. The practical trade-off is cost: ToT multiplies calls/tokens, so it is often reserved for high-severity complex incidents, and paired with retrieval and/or a smaller model to narrow the search space [194]. More generally, many approaches adopt **structured decomposition** without explicitly naming CoT/ToT by splitting the workflow into stages such as symptom extraction, candidate generation, evidence retrieval, and final decision [193, 215].

Fine-tuning and post-training for log tasks. When data is available, training often yields better stability than pure prompting [45]. It is useful to distinguish: (i) **continued pretraining** (domain-adaptive pretraining on large unlabeled log corpora) to improve coverage of log vocabulary and style; (ii) **supervised fine-tuning (SFT)** on labeled log tasks; (iii) **parameter-efficient tuning (PEFT)** (e.g., LoRA/adapters) to reduce cost and enable more frequent updates under drift; and (iv) **post-training/alignment** (instruction tuning and preference optimization) to improve instruction following, format adherence, and safety/harmlessness behaviors [122, 136]. In log analysis, post-training is especially relevant when the model must (a) reliably follow output schemas, (b) avoid leaking sensitive data in summaries, and (c) produce calibrated, non-overconfident explanations (including the ability to abstain) [122]. A common pattern is **hybrid training**: fine-tune a smaller encoder/LM for scoring or classification while using a larger aligned model for explanation and interactive assistance [11, 118]. Another pattern is **task-to-instruction transformation**, where multiple log tasks are cast into a unified instruction format so one model can support classification, extraction, and generation consistently [122].

Tool augmentation and agentic workflows. Agentic log-analysis approaches extend in-context retrieval by letting an LLM iteratively call external tools to gather, filter, and validate evidence [113, 141, 193]. Typical tools include log search/index queries, metric/trace dashboards, ticket/KB/runbook lookup, config/code retrieval, and lightweight parsers or summarizers that pre-compress high-volume inputs [113]. This paradigm is a natural fit for real-world log analysis because a single prompt window is rarely sufficient: relevant signals may be scattered across long time spans, multiple components, and heterogeneous repositories [113, 193].

Agentic workflows commonly follow a loop of: (i) **planning** (what evidence to obtain next and which hypotheses to test), (ii) **acting** (tool calls for retrieval, slicing, aggregation, or correlation), (iii) **observing** (interpreting tool outputs and updating the working context), and (iv) **deciding** (producing a final prediction/explanation/summary with evidence pointers) [193]. The benefits are stronger grounding and broader coverage; the costs are latency, more failure modes (tool errors, cascading mistakes), and higher security risks (e.g., prompt injection via retrieved documents) [113, 141]. Accordingly, practical deployments often add guardrails such as allow-listed tools, bounded iterations, schema validation, and conservative fallback behaviors [113, 149].

Verification, calibration, and controllability. Because log analysis is high-stakes and noisy, many approaches add explicit reliability layers: **self-consistency** (sample multiple reasoning paths and aggregate), **cross-checking** (verify claims against retrieved lines/windows or tool outputs), **schema validation** (force typed/JSON outputs and reject invalid responses), **evidence citation** (require each claim to point to a log span or retrieved document), and **selective generation** (generate only after candidate filtering by a smaller model) [66, 68, 107, 176].

3.5 Practical Concerns: Privacy, Security, Compliance, and Efficiency

Logs may contain sensitive identifiers (user IDs, IPs, hostnames), infrastructure topology, security indicators, and occasionally personal data [33, 73]. As a result, deploying LLMs for log analysis is often governed as much by **privacy/security constraints** as by model accuracy, while **latency and cost** remain first-class operational requirements [33, 36].

Commercial LLMs vs. self-hosted LLMs. Commercial LLMs are typically consumed via *remote APIs*, meaning prompts (and any retrieved context inserted into them) must be transmitted to an external model provider [33, 36]. This increases **data exposure risk** and reduces end-to-end controllability: even with contractual terms and vendor assurances, organizations may have limited visibility into provider-side handling (e.g., retention, secondary processing, or cross-region routing) and must treat the prompt as a potentially sensitive payload [13, 33]. In contrast, self-hosted (open-source) models keep inference *within the organization's trust boundary*, making data handling more controllable: access can be restricted via internal IAM, prompts/outputs can be logged or disabled under local policy, and compliance requirements (e.g., residency) can be enforced operationally [36]. However, self-hosting shifts responsibility to the operator: the main risks become the local security posture, including securing model-serving endpoints, hardening the retrieval stack, and protecting stored prompts, retrieved documents, and generated outputs [36].

Tool-augmented and retrieval-based workflows amplify these issues regardless of deployment. Untrusted retrieved text can introduce prompt-injection style instructions, and generated summaries/explanations can inadvertently echo secrets or sensitive identifiers if not constrained [13, 40].

Common mitigations in log analysis pipelines. Typical mitigations include: (i) **minimization** (send only the smallest relevant window; avoid full dumps), (ii) **redaction/masking** (identifiers, secrets, tokens), (iii) **bounded retrieval disclosure** (retrieve on-prem, but pass only small snippets/fields), (iv) **RBAC and auditing** (who can query what; safe logging of prompts/outputs), and (v) **policy-constrained outputs** (no secret echoing; structured extraction; citation requirements) [33, 36, 73]. These controls interact with utility: aggressive redaction may remove key signals, so many approaches combine masking with structured extraction and retrieval to preserve diagnostic content while limiting exposure [33, 36].

Cost drivers and practical optimizations. The main cost drivers are context length (tokens), number of model calls (multi-pass/agentic workflows), and retrieval/embedding computation [25, 129]. Common optimizations include windowing and deduplication, template compaction, hierarchical summarization, caching embeddings and intermediate summaries, batching, and **selective escalation** (invoke a large model only after a cheaper filter flags candidates) [113]. For self-hosted models, quantization and efficient serving reduce inference cost; for API models, prompt compression and retrieval-first designs reduce token budgets [25, 129].

Hybrid designs as the dominant deployment pattern. A frequent design is **small model for scoring/filtering** (ranking suspicious windows, candidate component selection, coarse labeling) plus **large model for explanation and synthesis** [11, 118]. This keeps expensive reasoning focused on a small subset of high-value context, reduces latency, and yields operator-facing artifacts (summaries, rationales, action candidates) without incurring LLM costs on every log line [113, 129]. In practice, such hybrids also simplify governance: only the most capable (and often most sensitive/costly) components are exposed to the most sensitive or costly data [33, 36].

4 Logging Statement Generation and Maintenance

Logging statement generation studies how to automatically insert and update log statements in source code so that systems emit informative runtime records for debugging and monitoring. In practice, insufficient logging can hinder incident triage and reduce observability, whereas excessive or inconsistent logging introduces noise, performance overhead, and scalability concerns. Hence, one active line of research is automated logging statement

generation, which aims to recommend or synthesize high-quality log statements so that developers can obtain effective logs with less manual effort.

4.1 Task Definition

Logging statement generation is the task of inserting and updating log statements into source code so that, during execution, the system emits textual records (logs) describing internal states and runtime events. Given a code context (e.g., a method, a basic block, or a file), automated logging must decide **where** to place logs, **what** to record, and **how** to phrase messages in a readable and maintainable way. A log statement is a hybrid artifact that combines a code-level logging API call (e.g., `logger.info(. . . )`) with a concise natural-language message and selected program variables. Effective generation, therefore, requires aligning program behavior with project-specific conventions, such as naming, verbosity, and severity-level usage, typically implemented through standard logging frameworks.

Listing 1 illustrates a simplified Java method instrumented with three log statements, and Listing 2 shows the corresponding runtime logs. The method logs an `INFO` message before accessing the repository, emits a `WARN` when the requested user is missing, and records an `ERROR` upon a database exception. Each statement pairs a concise event description with a small set of diagnostic variables (e.g., the user identifier and exception message), enabling developers to reconstruct key control-flow decisions and failure context from the log stream.

```java
public User getUser(String userId) {
    logger.info("Fetching user with id={}", userId);
    try {
        User user = userRepository.find(userId);
        if (user == null) {
            logger.warn("User not found: id={}", userId);
        }
        return user;
    } catch (DatabaseException ex) {
        logger.error("Failed to fetch user: id={}, error={}", userId, ex.getMessage());
        throw ex;
    }
}
```

Listing 1. A simplified example of single-line logging in Java

```
INFO  2025-05-12 10:15:03 UserService - Fetching user with id=U1024
WARN  2025-05-12 10:15:03 UserService - User not found: id=U1024
ERROR 2025-05-12 10:15:04 UserService - Failed to fetch user: id=U1024, error=connection timeout
```

Listing 2. Example single-line runtime logs emitted by Listing 1

In practice, logs are commonly placed at execution milestones, boundary interactions with external components, and exception-handling paths, where recorded context is most useful for diagnosis [38, 217]. Most systems rely on standard logging frameworks that provide unified APIs and configurable formatting; in Java, widely used options include `java.util.logging` [119], Log4j 2 [3], SLF4J [135], and Logback [134], while Python typically uses the built-in `logging` module [131], sometimes complemented by `structlog` [142] or `loguru` [24]. Accordingly, automated logging techniques generally assume these APIs and aim to generate statements that can be directly inserted and compiled in real projects.

4.2 Challenges

Although logging is essential for observability, producing high-quality logs in practice is non-trivial. Prior automated approaches, typically based on heuristics, static analysis, and traditional machine-learning models, have therefore faced persistent limitations. Existing studies consistently suggest that logging usefulness is largely determined by three intertwined decisions: where to log, what to log, and how to keep logs effective over time [38, 87, 213, 217].

4.2.1 Where-to-Log: Placement and Coverage. A core difficulty of logging is deciding **where** a statement should be inserted. Logs placed too sparsely may miss critical runtime evidence, reducing diagnosability, whereas overly dense logging introduces noise, log flooding, and runtime overhead [9, 38, 87, 217]. In real systems, diagnostically important events often occur along complex, low-frequency control-flow paths (e.g., rare branches, retries, and exception chains), making it difficult to identify insertion points that reliably capture true execution states [38, 86, 217].

Traditional automated techniques often suggest logging-statement placement using local syntactic cues (e.g., method entry/exit, catch blocks, API calls) or handcrafted rules over AST patterns [110, 185, 187, 217]. Such signals are inexpensive and scalable, but they are weak proxies for diagnostic value: they may over-instrument common patterns while missing semantically important events that span multiple statements or methods [85, 163]. Static-analysis-based methods can improve precision by tracking control flow and exceptional paths, yet they still struggle to decide which program points are worth logging from a human-debugging perspective, and they often produce redundant locations when applied at scale [27, 38, 86]. Moreover, placement decisions made in isolation are difficult to reconcile globally: a log that appears useful within one method may be inconsistent or redundant when viewed across the system, especially in large-scale or distributed applications [38, 86, 214, 217].

4.2.2 What-to-Log: Level, Message, and Variables. Even if a location is chosen, determining **what** to record remains challenging. First, log levels encode severity and expected frequency, but these levels are often ambiguous in real projects; the same event may be viewed as `INFO` in one module and `WARN` in another, leading to inconsistency and unreliable filtering during incident response [51, 89, 217]. Second, messages must convey clear event semantics, yet developers frequently struggle to express intent concisely, avoid vague wording, and align with project terminology [90, 217]. Third, selecting which variables to include is subtle: missing key variables reduces diagnostic value, whereas logging too many values increases verbosity [27, 86, 217]. These issues are exacerbated by the need to maintain a consistent style across contributors, components, and versions.

Prior approaches typically decompose this decision into separate subtasks, such as level prediction and variable selection, using local features (identifiers, types, nearby API calls, shallow data-flow signals) and then fill messages with templates [85, 89, 217]. This decomposition is convenient for modeling, but it often yields incoherent outcomes, for example, a predicted level that does not match the message phrasing or variables that are only weakly related to the event being described. Template- and pattern-based message generation further struggles to express nuanced intent (e.g., distinguishing a transient network hiccup from a persistent configuration error), producing rigid or generic strings that provide limited diagnostic cues. Overall, the fundamental difficulty is that level, variables, and message are semantically coupled decisions, whereas many traditional techniques treat them as largely independent.

4.2.3 How-to-Log: Quality, Maintainability, and Evolution. Logging quality is not only about writing a single statement correctly, but also about keeping logs useful throughout software evolution. As code and requirements change, logs can become stale, misleading, or incorrect (e.g., referring to outdated conditions, variables, or execution phases), yet they are rarely updated with the same rigor as functional code [62, 175, 213]. In addition, practical logging must respect non-functional constraints, such as performance overhead, storage budget, and privacy/security policies, which further complicate the amount and type of information that can be safely recorded [49, 214].

Traditional tooling provides limited support for maintaining logs over time. Simple consistency checks (e.g., unused variables, formatting rules) can catch surface-level issues, but they do not reliably detect semantic drift, in which the code behavior changes while the log message remains plausible [62, 213]. Likewise, mining- or rule-based detectors often rely on project-specific patterns and are brittle in the face of refactoring, API evolution, or style changes [87, 213, 217]. As a result, projects accumulate redundant, inconsistent, or outdated logs, gradually reducing the overall utility of the logging infrastructure and increasing maintenance costs [87, 213].

Table 2. Summary of LLM-based automated logging studies organized by task category.

Paper	Task	Paradigm	Input	Output	Context/Technique	Base Model(s)	Comparisons	Metric(s)
Full Logging Generation (General Purpose: Location, Message, Variables)								
LANCE [110]	Full Gen	Fine-tuning	Java Method	Log Stmt	-	T5	-	Acc, BLEU
FastLog [185]	Full Gen	Fine-tuning	Method + AST	Insert Point	AST Locator	PLBART	-	Acc, BLEU
LEONID [111]	Multi-Log	Fine-tuning	Full Method	0-N Logs	-	T5	-	BLEU, METEOR
ELogger [39]	Block Gen	Fine-tuning	Code Block	Single Log	Block Features	Enc-Dec	-	-
UniLog [187]	Full Gen	ICL	Method	Loc+Level+Msg	Warm-up Demos	Codex, GPT-3	-	PA, LA, MA
SCLogger [86]	Full Gen	ICL + CoT	Method	Log Stmts	Call Graph (2-hop)	GPT-4, LLaMA2	GPT-3.5	PA, AOD, F1
PDLogger [27]	Full Gen	ICL + CoT	Method + Slice	Logs	Backward Slicing	o3-mini, DeepSeek	LLaMA3-70B	PA, L-ACC
AUCAD [198]	Align Gen	Fine-tuning	Method + Issue	Logs	Issue Alignment	LLaMA3.1	Magicoder	PA, LA, VF1
Zhong et al. [214]	Full Gen	Fine-tuning	Java Method	Logs	Compact Model FT	LLaMA-8B, Mistral	GPT-4o, Claude	PA, LA, F1
Specialized Logging Generation (Test Code, File-level, Configuration)								
File-Level [140]	File Gen	ICL	Python File	Multi Logs	Pipeline Stages	GPT-4o-mini	-	L-ACC, AOD
Test-Code [151]	Test Log	ICL	Test Method	Assert/Log	Test Intent	GPT-3.5, CodeLlama	GPT-4o, Llama3	Acc, BLEU
ConfLogger [147]	Config Log	ICL + Rules	Config Code	Logs	Config Analysis	GPT-4o	-	LA, AOD
Benchmarks, Empirical Studies, and Visions								
LogBench [85]	Study	ICL / Zero	Method	Level+Msg	-	GPT-3.5/4	InCoder, StarCoder	BLEU, ROUGE
AL-Bench [163]	Benchmark	-	Java Code	Logs	Benchmark Suite	-	-	ALD, DEA
Auto-Logging [9]	Vision	-	-	-	AI Instrumentation	-	-	-
Specific Log Tasks (Level Prediction, Defect Detection, Repair, Quality)								
Heng et al. [51]	Level Pred	FT / Zero	Snippet	Level	-	CodeLlama-13B	BERT, RoBERTa	Acc, AUC
OmniLLP [121]	Level Pred	RAG + ICL	Log + Neighbors	Level	Cluster Retrieval	CodeXEmbed	-	ARI, AUC
LogUpdater [213]	Log Repair	Agent	Defect Log	Fixed Log	Defect Taxonomy	CodeT5+, GPT-4o	Claude3.5	BLEU, ROUGE
Defects4Log [175]	Detection	ICL + CoT	Method + Log	Defect Type	Defect Patterns	DeepSeek-R1	GPT-4o	Precision, Recall
LOGIMPROVER [87]	Quality	-	Codebase	Improved Logs	Proactive Analysis	-	-	-

4.3 LLM-based Approaches for Logging Statement Generation

Table 2 shows that recent LLM-based logging techniques follow several methodological paradigms rather than a single recipe. Across these paradigms, the key idea is to exploit LLMs as joint models over code and natural language, so that the three core decisions, where to log, what level to use, and how to phrase the message and variables, are no longer made by hand-crafted rules or local classifiers; these logging decisions emerge from learned semantic patterns and in-context reasoning. Below, we organize prior work by how it uses LLMs and highlight the mechanisms that make each paradigm effective.

4.3.1 Fine-tuned End-to-End Generators. Early work treats logging as a sequence-to-sequence learning problem and fine-tunes encoder–decoder models to transform raw code into logged code. LANCE [110] fine-tunes a T5 model on pairs of Java methods with and without logging, so that the model learns to rewrite an input method into an output method containing an injected log statement. During decoding, the model jointly decides where to insert the statement, which level to use, which variables to log, and how to phrase the message. This joint modeling is precisely where LLMs outperform handcrafted AST patterns: instead of matching rigid templates,

the fine-tuned model learns the distribution of logging patterns from large corpora and can adapt to unusual constructs, such as nested exceptions or multi-call sequences, without additional rules.

As the first end-to-end LLM-based automated logging tool, LANCE delivers an innovative proof of concept. Yet, it suffers from three key limitations: (1) rewriting the entire method may inadvertently alter non-logging code, (2) full-method input and output incurs substantial latency compared to local insertion, and (3) it always injects exactly one log statement, thus failing to decide when logging is unnecessary or when multiple logs are required. Subsequent approaches explicitly refine the use of fine-tuning to mitigate these issues. FastLog [185] decouples location from content: a learned module first predicts fine-grained insertion points, and the LLM then generates only the log statement to be inserted. This changes the learning problem from "rewrite the whole method" to "predict a small edit" and thus preserves the original code while reducing decoding cost. LEONID [111] extends LANCE by teaching the model to predict not just the content of a single log, but also whether a method needs logging at all and how many logs to insert. This turns the fixed one-log-per-method assumption into a learned, method-level decision, closer to how developers actually log. ELogger [39] pushes the same idea to block level: it first classifies whether a block warrants logging and only then invokes the generator, reducing over-instrumentation by using the LLM where it matters most. AUCAD [198] further strengthens this fine-tuning line by automatically constructing aligned datasets from log-related issues; the model is trained not just on arbitrary method/log pairs, but on positive and negative examples mined from real bug reports, which sharpens its notion of "good" versus "bad" logging behavior.

Overall, these fine-tuned generators demonstrate that LLMs can implicitly encode common logging idioms and jointly model location, level, message, and variables. Their main trade-off is training and deployment cost: models are tied to specific architectures and datasets, motivating lighter-weight paradigms such as in-context learning.

4.3.2 Prompt-based In-Context Learning. With stronger LLMs, an alternative to task-specific fine-tuning is prompt-based in-context learning (ICL). Compared with fine-tuned models, ICL-based approaches greatly reduce training and infrastructure costs and make it easier to deploy or swap models in practice. UniLog [187] is the first general-purpose framework to systematically exploit ICL for logging. It feeds the target method, along with a small set of (code, log) examples from the same project, and asks a Codex-like model [14] to output the location, level, and message. Mechanistically, UniLog leverages an emergent ability of LLMs: given a handful of exemplars, the model can infer a project's logging style (e.g., how verbose to be, which variables to mention) and apply that style to new methods. A warm-up strategy stabilizes this behavior by first exposing the model to generic logging patterns before switching to project-specific examples, reducing variance across invocations.

ICL-based methods shift the burden from training to prompting: instead of adjusting millions of parameters, practitioners adjust the prompt. This makes automated logging much easier to deploy or migrate between models, and it directly leverages LLM strengths in style transfer and pattern imitation. File-level ML logging [140] prompts GPT-4o-mini with Python pipeline files and a few annotated examples, letting the model infer where to place logs at pipeline boundaries. Test-code logging [151] supplies test methods and exemplar test logs so that the model learns to describe test intent and failure conditions. ConfLogger [147] combines static analysis to find configuration-sensitive operations with ICL prompts that show how such operations are typically logged, enabling the LLM to specialize its suggestions for configuration diagnosability.

4.3.3 Execution-Aware Reasoning with CoT and Static Context. While plain ICL uses only the immediate method body, real logging decisions often depend on broader execution context and on a chain of reasoning (e.g., "this call propagates an error from a lower-level component, so it should be logged at `ERROR`"). To expose this context and reasoning to the LLM, several works enrich prompts with chain-of-thought (CoT) style structure and execution-aware static information.

SCLogger [86] augments the input with two-hop callers and callees on the call graph and organizes the prompt in a CoT fashion. Instead of giving the LLM only the target method, SCLogger explicitly injects the surrounding

static context: which functions call this method, what it calls, and how logging is used there. This design exploits a key LLM capability: when given a broader static context, the model can infer variable roles (e.g., which ID identifies a request versus a user), propagate logging conventions across the call chain, and better align with project-specific patterns without extra training. The CoT-style prompt further nudges the model to "think through" why a location is important before emitting a final log statement.

PDLogger [27] makes the execution link even tighter by performing backward slicing from predicted log locations. Data and control dependencies along the slice are turned into explicit prompt tokens, and variables flowing into the logging site are enumerated with their syntactic and semantic roles. In effect, PDLogger transforms implicit runtime behavior (which values influence this point, under which conditions) into an explicit context for the LLM. This allows the model to simulate execution more accurately and to generate multiple log statements per method that are consistent with the underlying control and data flow, rather than relying solely on local patterns.

These CoT- and execution-aware prompts thus serve as a bridge between classical program analysis and LLM reasoning: static analyses compute precise dependencies, and the LLM uses them to perform high-level semantic reasoning and message generation.

4.3.4 Retrieval-Augmented Logging and Subtasks. Some prior research focuses on specific subtasks (e.g., level prediction, message reconstruction) and combines LLMs with retrieval-augmented generation.

For log levels, Heng et al. [51] benchmark a range of models in zero-shot, few-shot, and fine-tuned settings, showing that local code context suffices in many cases but that borderline severity levels remain difficult. OmniLLP [121] takes a retrieval-augmented view: it clusters log statements by semantic similarity and developer ownership, then retrieves representative neighbors to include in the prompt when predicting the level for a new log. Here, retrieval plays to LLM strengths in analogical reasoning: instead of deciding "in the abstract" whether a statement is `INFO` or `WARN`, the model compares the target context against a handful of concrete, project-specific examples and chooses the level that best aligns with those patterns. This improves level consistency and disambiguation for borderline cases, effectively grounding the LLM in real project practices.

LogBench [85] investigates the message and level subtasks via a fill-in-the-blank formulation. It constructs paired datasets of original methods (LogBench-O) and transformed-but-functionally-equivalent methods (LogBench-T), removes existing log statements, and asks LLMs to reconstruct the level and message at fixed locations. This approach isolates "what-to-log" from "where-to-log" and assesses whether models rely on brittle syntax or genuine semantic understanding: if performance holds on LogBench-T, where surface syntax has changed, the model is likely relying on deeper semantic cues. By comparing a variety of commercial and open-source LLMs under this setup, LogBench clarifies which architectures and prompting strategies are better at reconstructing realistic log content.

4.3.5 Agentic Pipelines for Log Maintenance. Finally, some work goes beyond generating new logs and using LLM agents in multi-step maintenance pipelines. LogUpdater [213] first uses a classifier (trained on synthesized defective logs) to detect and categorize logging defects, and then invokes an LLM such as CodeT5+ or GPT-4o to repair the flagged statements. The approach decomposes the task: a lightweight detector narrows down suspicious logs, and the LLM focuses on high-level semantic repair (e.g., aligning message text with code behavior and fixing mismatched variables or temporal descriptions). By separating "where is something wrong?" from "how should it be rewritten?", LogUpdater reduces the risk of unconstrained LLM edits and leverages the model's strength in natural-language revision rather than raw anomaly detection.

Defects4Log [175] complements this pipeline view by providing a benchmark and taxonomy for log defects and systematically evaluating LLMs' ability to classify and reason about them. Together, these efforts illustrate a broader pattern: instead of using LLMs as single-shot generators, logging research increasingly wraps them in

agentic workflows—detect, classify, explain, repair—so that each step can be checked, constrained, or retried, and logs can evolve along with the codebase.

In summary, LLM-based approaches for logging differ not only in which tasks they tackle, but also in how they expose code, context, and prior examples to the model. Fine-tuned generators encode logging patterns into model parameters; ICL methods exploit emergent style imitation; CoT and slicing turn program structure into explicit reasoning context; retrieval augments level and message decisions with concrete project examples; and agentic pipelines harness LLMs' semantic repair abilities while controlling their edits. These mechanisms collectively explain why LLMs can address long-standing challenges in where-to-log, what-to-log, and how-to-maintain-logs that were difficult to handle with hand-crafted rules or traditional ML alone.

Across studies, the key shift introduced by LLMs is not raw generation capability but the ability to treat logging intent as a semantic inference problem rather than a syntactic insertion task. The most effective approaches consistently constrain and inform LLMs using execution-aware context, project-specific exemplars, or multi-step control, instead of relying on unconstrained generation. At the same time, LLMs do not eliminate the ambiguity of logging decisions but relocate difficulty from rule design to context selection, validation, and oversight, pointing toward hybrid, iterative maintenance workflows as the dominant design pattern.

4.4 Performance Evaluation Metrics

For automated logging, evaluation has gradually shifted from "exact-match-centric" reporting to a more nuanced, component-wise view. Early end-to-end generators often borrowed machine translation-style metrics (e.g., n-gram overlap such as BLEU [124]) because the task was framed as producing a single reference-like statement for each context. However, as LLM-based methods moved to prompt-based generation and multi-log settings, two issues became hard to ignore: (1) logging is inherently under-specified (multiple insertion points or phrasings can be equally reasonable), and (2) surface-form metrics are brittle (semantically correct messages can score poorly due to paraphrasing). As a result, more recent evaluations increasingly combine strict metrics (for comparability) with softer, semantics-aware ones (for faithfulness), such as embedding-based similarity metrics like BERTScore [203].

Below, we summarize the most widely adopted metrics for each aspect and highlight what they capture (and what they miss).

For **logging-statement position**, the de facto metric is **Position Accuracy (PA)**: whether the predicted insertion point matches the reference point at an agreed granularity (often statement/block rather than exact line). PA is easy to interpret and facilitates comparability across papers, so it remains a useful headline number. However, PA becomes less informative once methods are allowed to output multiple logs or "no log" decisions, because the problem is no longer a single-label prediction. In such cases, it is more appropriate to treat placement as a set prediction problem and report **precision/recall/F1** over predicted logging-statement position. This directly exposes two practically relevant failure modes: over-instrumentation (low precision) and under-instrumentation (low recall).

For **log level**, **Level Accuracy (LA)** (or L-ACC) is the simplest and most commonly reported metric. Yet log levels are not purely categorical in practice: they have an ordinal meaning (e.g., predicting `WARN` when the reference is `ERROR` is typically less harmful than predicting `DEBUG`). Therefore, modern evaluations often complement LA with an ordinal-distance metric, such as **Average Ordinal Distance (AOD)**, to quantify the extent to which predictions deviate along a severity scale. In addition, because log levels can be imbalanced (e.g., `INFO` dominates), macro-F1 or per-class reporting is often more diagnostic than a single LA number when the goal is to understand behavior on rare but critical levels.

For **log message**, exact-match metrics such as **Message Accuracy (MA)** are increasingly treated as a "strict sanity check" rather than the main quality signal, since natural language admits paraphrases. To maintain backward compatibility, many studies still report overlap-based metrics such as **BLEU/ROUGE**, but these metrics

mainly reflect surface similarity and can penalize semantically faithful rephrasing [124]. For LLM-generated messages, it is now common (and more defensible) to additionally report a semantics-aware metric such as **BERTScore**, which compares contextual embedding similarity instead of exact token matches [203]. Finally, edit-distance-style measures (token/character Levenshtein distance) are useful when the paper's objective includes developer fixing effort: they approximate how much manual editing is needed to correct a generated message.

For **variables**, the field has largely converged on viewing variable logging as a set selection problem. Accordingly, **Variable Precision/Recall/F1** is often more informative than a single "all-or-nothing" accuracy. These metrics separate "missing important variables" (low recall) from "logging too many irrelevant variables" (low precision), and they also interact naturally with hallucination checks (e.g., penalizing variables that do not exist in scope). In addition, if a method jointly generates message and variables, it is worth reporting variable metrics separately from message metrics to avoid conflating semantic description quality with instrumentation completeness.

5 Log Parsing

Modern software systems generate massive volumes of execution logs to support debugging, monitoring, and reliability engineering. However, the scale and velocity of log streams make manual inspection impractical: operators must quickly locate relevant events, correlate symptoms across components, and identify anomalous behaviors under strict time constraints. This motivates automated log analysis, which treats logs as machine-processable data rather than free-form text. A prerequisite for most downstream log analytics is log parsing, the process of converting raw log lines into structured event records [6, 49]. By normalizing heterogeneous messages into reusable templates and extracting runtime parameters, log parsing enables efficient indexing, aggregation, statistical modeling, and cross-run comparison, and thus directly impacts the effectiveness of tasks such as anomaly detection, failure prediction, and root cause analysis.

We review LLM-based log parsing via reusable paradigms and the approaches that make them practical at scale, highlighting how they address the limitations of traditional parsers (format drift, long-tail events) and what new trade-offs they introduce (e.g., context budgets and orchestration overhead).

5.1 Task Definition

Log parsing is the task that transforms each raw log line into a structured representation that separates **invariant** event semantics from **dynamic** runtime values. Concretely, given a raw log line x, a parser typically outputs (i) a set of **header fields** (e.g., timestamp/date, severity level, process/thread, component), (ii) a **log template** t where dynamic spans are replaced by placeholders (e.g., <*>), and (iii) a **parameter list** v that records the extracted runtime values. At the stream level, especially in online settings where logs must be parsed on-the-fly as they arrive, parsers often maintain a **template inventory** (or event dictionary) and assign an **event ID** to each unique template so that the raw stream can be converted into an event sequence for downstream mining [48, 165, 177].

Figure 3 illustrates this objective with Hadoop logs. For example, raw messages such as "Jetty bound to port 62267" and "Jetty bound to port 62258" differ only in the concrete port number, yet they represent the same underlying event. A correct parser normalizes them to the same template "Jetty bound to port <*>" and extracts the port number as a parameter. Similarly, the message "Web app /mapreduce started at 62267" is parsed into the template "Web app <*> started at <*>", separating stable keywords (e.g., "Web app", "started at") from variable tokens (e.g., the path and port).

In practice, log parsing is rarely a single monolithic step; most pipelines implement it as a two-stage process. First, a **preprocessing** stage extracts standard fields and isolates the message body. This stage is typically implemented using regular expressions or log format specifications because these header fields are usually emitted in a framework-controlled and relatively stable format (e.g., the timestamp/level/thread/component

Log messages

2015-10-18 18:01:51 INFO main org.apache.hadoop.http.HttpServer2 Jetty bound to port **62267**
2015-10-18 18:01:51 INFO main org.apache.hadoop.http.HttpServer2 Jetty bound to port **62258**
2015-10-18 18:01:52 INFO main org.apache.hadoop.yarn.WebApps Web app **/mapreduce** started at **62267**
2015-10-18 18:01:53 INFO main org.apache.hadoop. app.RMContainerRequestor nodeBlacklistingEnabled:**true**
2015-10-18 18:01:53 INFO IPC Server org.apache.hadoop.ipc.Server IPC Server listener on **62270**: starting

Log Parsing

Parsed logs

Date	Time	Level	Process	Component	Log Template
2015-10-18	18:01:52	INFO	main	org.apache.hadoop.http.HttpServer2	Jetty bound to port <*>
2015-10-18	18:01:52	INFO	main	org.apache.hadoop.http.HttpServer2	Jetty bound to port <*>
2015-10-18	18:01:52	INFO	main	org.apache.hadoop.yarn.WebApps	Web app <*> started at <*>
2015-10-18	18:01:53	INFO	main	org.apache.hadoop.app.RMContainerRequestor	nodeBlacklistingEnabled:<*>
2015-10-18	18:01:53	INFO	IPC Server	org.apache.hadoop.ipc.Server	IPC Server listener on <*>: starting

Fig. 3. An example log parsing result from Hadoop.

layout is determined by logging libraries and configuration, and thus changes slowly and predictably). Second, a **message parsing** stage focuses on the remaining free-text-like body and performs template–parameter separation. Unlike headers, the message body is largely developer-written and varies widely across components and versions; it mixes natural-language-like phrases with project-specific tokens and runtime values, and it evolves as code changes. As a result, a single set of regular expressions is rarely sufficient to robustly normalize message bodies without either over-generalizing templates or exploding them into many variants. Across these implementations, the core goal remains the same: to produce stable templates that support consistent grouping while accurately extracting the parameters needed for diagnosis.

5.2 Challenges

Although log parsing has been studied extensively, traditional (non-LLM) parsers face persistent difficulties because logs are semi-structured artifacts produced by evolving software. These difficulties arise from both the *data* (highly variable message styles and continuous drift) and the *operational requirements* (low latency, high throughput, and stable event identities in online pipelines). Below, we summarize four recurring challenges that limit the robustness of traditional parsers.

5.2.1 Structural variability and format drift. Logs exhibit substantial structural variability across systems, components, and logging libraries, and even within the same system, different modules may adopt different writing styles, delimiters, and token conventions. Many traditional parsers therefore embed implicit assumptions about message structure (e.g., token positions, separators, fixed vocabularies, or stable prefix patterns) to make parsing tractable [23, 48]. Such assumptions can work well for the formats they were designed around, but they degrade when encountering unseen layouts, rare formatting choices, or cross-component inconsistencies.

Crucially, this variability is not static: log formats drift over time as code evolves, dependencies are upgraded, configuration changes alter logging layouts, and developers revise wording for readability or diagnostics [67, 106, 197]. Under drift, an initially accurate template inventory can become stale, causing template fragmentation (the same event being split into multiple templates) or erroneous merges (different events being grouped together). Rule-based parsers then require continuous maintenance, while data-driven parsers trained on historical logs can suffer when the distribution of templates or vocabulary shifts.

5.2.2 Ambiguous template–variable boundaries. Even with a stable structure, deciding which tokens are *invariant* versus *variable* is inherently ambiguous. Runtime values have various forms (IDs, UUIDs, paths, IP addresses, error codes), and some values may appear constant within a single deployment but should be treated as variables across runs (e.g., stable ports or hostnames in a fixed environment). Conversely, some numeric tokens or symbols are semantically meaningful constants (e.g., HTTP status classes or well-known error codes) that should remain in templates.

As a result, common heuristics such as "all digits are variables" or delimiter-based splitting can lead to either over-general templates (too many placeholders, losing discriminative semantics) or over-specific templates (template explosion). Both outcomes reduce downstream utility: overgeneralization blurs distinct events, whereas over-specificity fragments event types, harming aggregation and anomaly detection.

5.2.3 Long-tail patterns and coverage gaps. Real log streams often follow a heavy-tailed distribution: a small number of frequent templates dominate the volume. In contrast, rare templates may be crucial for diagnosis (e.g., exception paths and failure-handling branches). Traditional parsers that rely on frequency cues or stable clustering can underperform on the long tail. Rare patterns may be misclassified into frequent templates (degrading precision) or split into many singletons (degrading recall and inventory usefulness), resulting in poor coverage precisely where operators most need structured signals.

This long-tail effect also interacts with drift and boundary ambiguity: rare events provide fewer repetitive examples to stabilize grouping, making them more vulnerable to mis-parsing when wording changes or when variable boundaries are subtle.

5.2.4 Operational constraints and stability requirements. Operational log parsing is commonly performed at a large scale, where millions of log lines must be processed with low latency, high throughput, and predictable resource usage. Traditional parsers are attractive in production usages because they provide deterministic execution and low per-line cost, but achieving *both* high throughput and high accuracy becomes difficult as structural variability and drift increase.

Moreover, many downstream pipelines assume that event identities remain stable across time windows (e.g., for alerting, dashboards, and anomaly detectors trained on historical event sequences) [49, 67]. Template churn caused by drift, boundary ambiguity, or long-tail fragmentation can propagate instability into downstream analytics, degrading both monitoring reliability and cross-run comparability.

5.3 LLM-based Approaches for Log Parsing

Table 3 shows that LLM-based log parsing has expanded from early capability probes into several reusable methodological families. Across these families, the core idea is to trade brittle, log-specific heuristics (e.g., delimiter rules and fixed similarity thresholds) for semantic generalization: LLMs can better infer which spans belong to the stable event skeleton versus instance-specific parameters, even when surface forms vary across environments and versions. In practice, however, the dominant constraint is often not only correctness but also the cost and stability of invoking LLMs at scale, motivating log parsers that amortize or reduce LLM queries through reuse layers.

5.3.1 Benchmarks and empirical analyses as the starting point. Early work primarily asked whether general-purpose LLMs can parse logs at all and how sensitive they are to prompting choices. Le & Zhang [76] provides a representative capability analysis of GPT-3.5 (ChatGPT) under zero-/few-shot prompts, showing that seemingly small prompt-format differences can materially change parsing quality. As the field matured, benchmark-style efforts (e.g., LogEval [22]) began to standardize evaluation and make it clearer *which aspect* improves (e.g., grouping versus precise template extraction), thereby enabling more comparable method design.

Table 3. Summary of LLM-based log parsing approaches.

Paper	Setting	Superv.	LLM Technique	Reuse / Eff.	Base Model(s)	Metric(s)
Benchmarks and empirical analyses						
Le & Zhang [76]	Online	Both	Zero-/few-shot prompting study (capability analysis)	–	GPT-3.5	PA, GA, ED
Astekin et al. [4]	Online	Both	Few-shot prompting comparison across LLMs	–	CodeLlama-7B	ED, PA, LCS
LogEval [22]	Offline	Sup.	Benchmark suite (multiple log-analysis tasks incl. parsing)	–	GPT-4	PA, ED
LogBase [196]	Offline	–	Benchmark dataset for semantic log parsing	–	–	–
LLMDPP [174]	Online	Sup.	DPP-based diverse sample selection for few-shot parsing	–	Flan-T5-small	GA, PA
VISTA [100]	Online	Sup.	Variable-aware ICL (efficiency/cost-oriented study)	Cache	GPT-3.5-Turbo	PA
Supervised parsers and log-specific models (fine-tuning / sequence labeling / pretraining)						
GPT-2C [145]	Offline	Sup.	Fine-tuned GPT-style parser (Q&A formulation)	–	GPT-2	F1
LogStamp [165]	Online	Sup.	Sequence labeling for template/parameter tagging	–	BERT-base	RI
LogPPT [75]	Offline	Sup.	Prompt-based few-shot token labeling	–	RoBERTa	PA, GA, ED
LLMParser [105]	Offline	Sup.	Few-shot fine-tuning for template generation	–	LLaMA-7B	PA, GA
Mehrabi et al. [112]	Offline	Sup.	Compact fine-tuning (effectiveness study)	–	Mistral-7B	MLA, ED, F1
OWL [44]	Offline	Sup.	Log foundation model training (IT operations)	–	OWL	RI, F1
PreLog [77]	Online	Sup.	Log foundation model pretraining (log analytics)	–	PreLog-140M	GA, OA, ED, uPA
Le et al. [78]	Online	Sup.	Few-shot tuning of semantic LLMs + adaptive caching	Adap. cache	RoBERTa (125M)	GA, PA, FGA, FTA
LogLM [97]	Offline	Sup.	Instruction tuning for log tasks	–	LLaMA-2-7B	RI, F1
SuperLog [65]	Offline	Sup.	Continual pretraining + interpretable domain knowledge	–	LLaMA-2-7B	RI, F1
Prompting / in-context learning parsers (zero-shot, few-shot, CoT, adaptive examples)						
DivLog [188]	Online	Sup.	Adaptive few-shot ICL with prompt enhancement	–	GPT-3 (Curie)	PA, PTA, RTA
Liu et al. [95]	Online	Unsup.	Prompt strategies (fixed demos) for online parsing/-analysis	–	GPT-3.5-Turbo	F1
Xu et al. [211]	Online	Unsup.	Zero-shot prompting for template extraction	Temp. DB	GPT-3.5	PA, GA, ED
LogGenius [195]	Online	Unsup.	Zero-shot prompting + synthetic log generation (assistance)	–	GPT-3.5-Turbo	PA
Zhou et al. [216]	Online	Unsup.	Few-shot prompting (with a BERT-based component in pipeline)	–	GPT-3.5	GA, MLA, ED
LLM-TD [170]	Offline	Unsup.	ICL-based template detection for security event logs	Temp. cand.	OpenChat	FGA
Lemur [206]	Online	Unsup.	Entropy-guided clustering + CoT template merging	Cluster	GPT-3.5	Template accuracy
LogRules [60]	Offline	Sup.	ICL enhanced with rules (case-/rule-based knowledge injection)	Rules	GPT-4o-mini + LLaMA-3-8B-Instruct	PA, GA, ED, FGA, FTA
Practical and efficiency-oriented approaches (RAG / grouping / caching / scheduling / agent)						
LILAC [67]	Online	Sup.	ICL + hierarchical demo sampling + cache refinement	Adap. cache	GPT-3.5	PA, GA, FGA, FTA
LogBatcher [184]	Online	Unsup.	Demonstration-free parsing via retrieval (no labeled demos)	Cache	GPT-3.5-Turbo	GA, MLA, ED
LogParser-LLM [212]	Online	Unsup.	Grouping + multi-step prompting (RAG-style)	Grouping	GPT-3.5-Turbo	GA, PA, FGA, FTA, GGD, PGD
SelfLog [126]	Online	Both	Group-wise ICL with self-evolutionary merge tree	Tree	GPT-3.5	GA, PA, PTA, RTA
HELP [186]	Online	Both	Embedding clustering + LLM template generation	Cluster	text-embedding-3-small + Claude-3.5-Sonnet	GA, FGA, PA, FTA
LibreLog [106]	Offline	Unsup.	Retrieval-augmented parsing + self-reflection refinement	Temp. mem.	LLaMA-3-8B-Instruct	PA, GA
Parse-LLM [153]	Online	Unsup.	Agentic decomposition with callable tools (header separation, etc.)	Tools	GPT-4	GA, FGA, PA, FTA
AdaParser [182]	Online	Unsup.	Self-generated ICL + self-correction	Tree	GPT-3.5	GA, FGA, PA, FTA
Huang et al. [57]	Online	Unsup.	Unsupervised parser with retrieval (no labeled examples)	Temp. DB	GPT-3.5	GA, FGA, PA, FTA
EPAS [16]	Online	Unsup.	Asynchronous scheduling of LLM queries (tail-latency aware)	Async	Llama-3.1-70B-Instruct	GA, FGA, PA, FTA
SemanticLog [197]	Online	Unsup.	Variable-aware template generation for large-scale parsing	Cache	LLaMA-2-7B	GA, PA, FTA
Duan et al. [28]	Offline	Unsup.	LogBatcher-style grouping + LLM template correction	Cache	GPT-3.5-Turbo	GA, FGA, PA, FTA
InferLog [177]	Online	Unsup.	ICL-oriented prefix KV-cache reuse + auto-tuning	KV cache	Qwen2.5-14B-Instruct	PA, PTA, RTA, GA

A key takeaway from follow-up empirical analyses is that *example selection* and *variable handling* repeatedly emerge as high-leverage factors that affect both accuracy and cost (e.g., LLMDPP [174], VISTA [100], and comparative studies [4]). In other words, the "benchmark line" of work did not merely measure performance; it also identified recurring design knobs that later approaches considered.

5.3.2 Supervised parsers and log-specific models: training for structure, but with fewer labels than before. One family treats log parsing as a supervised learning problem and leverages modern LLMs to learn invariant skeletons and variable boundaries from data, often with greater tolerance for noisy token patterns than purely heuristic rules. A useful way to view this family is *where supervision is placed*: some methods supervise *token-level decisions* (constant vs. variable), while others supervise *template generation* directly. LogPPT [75] is a label-efficient supervised parser. It fine-tunes a RoBERTa model on a small labeled dataset to perform few-shot token-level parameter labeling, reducing reliance on handcrafted parsing rules while remaining grounded in curated examples. Complementarily, LLMParser [105] represents lightweight supervised adaptation via few-shot fine-tuning, enabling a compact open-source LLM to directly generate templates with high accuracy when limited labels are available.

In parallel, log-specific pretraining and domain adaptation aim to internalize "log language" from large corpora, narrowing the gap between natural text and log tokens. PreLog [77] illustrates this direction by pretraining a log-focused model for log analytics tasks (including parsing), providing a stronger initialization than generic LLMs for downstream log understanding. Overall, supervised paradigms tend to provide more controllable output formats and predictable behavior (often with lower inference-time cost than per-line prompting), but they trade off flexibility and maintenance: they rely on curated labels or domain corpora, and evolving templates may still require re-training or re-alignment to sustain accuracy under drift.

5.3.3 Prompt-based parsing: treating template extraction as structured generation. A second family avoids parameter updates and instead casts log parsing as a *structured prompting* problem: given a raw message, the model is instructed to emit a normalized template (and optionally the extracted parameters) under a constrained output format. This direction is attractive for rapid deployment because it can work in a zero-shot or a few-shot manner without collecting training data. For example, Xu et al. [211] show that even a zero-shot LLM can extract templates online when paired with a lightweight template repository to reuse previously derived results. Beyond one-shot prompting, fixed-demo prompt strategies have been explored to stabilize parsing behavior in streaming settings (e.g., Liu et al. [95]), and some pipelines further leverage synthetic or auxiliary LLMs to improve downstream parsing quality (e.g., LogGenius [195]).

More broadly, prompt-based parsing effectively converts "rules in code" into "rules in the prompt", which improves iteration speed and portability across models, but shifts the main technical challenge to *controllability and normalization*: the approach must guard against over-generalization, inconsistent placeholders, and format violations, which motivates explicit output constraints, prompt templates tailored to log syntax, and restricted decision spaces such as candidate/template detection prompts (e.g., LLM-TD [170]) or rule-augmented prompting (e.g., LogRules [60]).

5.3.4 In-context parsing with adaptive example selection: from static demos to on-the-fly abstraction. Within prompting, a key refinement is to make demonstrations adaptive rather than fixed. DivLog [188] is representative: it dynamically selects relevant labeled examples for each incoming log, turning parsing into a contextual compare-and-contrast process that helps the model separate invariants from variables for the specific message at hand.

Empirical follow-ups suggest that *diversity* in the selected examples and explicit treatment of variables are recurring levers for robustness and cost control (e.g., LLMDPP [174] and VISTA [100]). The synthesis is that adaptive ICL shifts the core engineering problem from "design a single good prompt" to "design a good *retriever/sampler* for prompts", which improves coverage but introduces a new dependency on selection quality and context-budget constraints.

5.3.5 Practical online parsing: ICL combined with reuse (cache/memory/template repositories). A third family focuses on making LLM parsing feasible in online pipelines by explicitly *reusing* past results to amortize LLM queries and stabilize outputs. LILAC [67] exemplifies this engineering pattern: it combines accuracy-oriented ICL (via carefully selected demonstrations) with an adaptive parsing cache that stores and refines templates over time, reducing repeated calls for recurring patterns while improving consistency under streaming workloads. In this view, reuse is not merely an optimization; it becomes part of the parser's state, enabling near-real-time throughput when logs contain many near-duplicates.

Many other approaches instantiate the same reuse principle with different "cached artifacts." Some attach a template repository to zero-shot prompting so previously derived templates can be looked up and reused [211], while unsupervised parsers can rely on a template database to eliminate labeled demonstrations but still exploit reuse across the stream (e.g., Huang et al. [57]). Other approaches reduce open-ended generation by narrowing the search space, e.g., by prompting the model to identify multiple templates from a batch or from candidate structures (LLM-TD [170]), or by coupling semantic modeling with adaptive caching for robustness under drift [78]. The key trade-off is *error amplification*: once an incorrect template is cached or promoted, reuse layers can propagate it widely. Hence, practical approaches typically require cache update policies, validation/refinement hooks, and sometimes variable-aware reuse strategies that explicitly track parameter slots to avoid template corruption at scale (e.g., SemanticLog [197]).

5.3.6 Unsupervised retrieval and refinement: reducing labels via self-generated context and iterative correction. A fourth family targets the no-label regime by replacing curated demonstrations with retrieval and refinement loops. Instead of supplying ground-truth templates as exemplars, these methods retrieve representative logs (or groups of logs) as contextual anchors and let the LLM infer templates from the retrieved context. LogBatcher [184] is representative: it performs demonstration-free parsing by retrieving or grouping similar logs and querying the LLM only on representative instances, thereby reducing both labeling and query volume. LibreLog [106] pushes this direction further with open-source models: it combines retrieval-augmented parsing with iterative self-reflection and a template memory, using refinement to compensate for missing labels while improving efficiency and privacy. Related work employs the same approach via self-generated ICL and self-correction mechanisms (e.g., AdaParser [182]) or via unsupervised template repositories that accumulate reusable templates over time [57].

A recurring observation is that "unsupervised" here is rarely "single-shot". Effectiveness hinges on the quality of upstream grouping/retrieval: if retrieved logs mix multiple templates, the model may output an averaged, over-general template; if retrieval is too narrow, variables may be frozen as constants. Consequently, many approaches couple retrieval with explicit *grouping and correction* strategies, such as embedding-driven clustering before template generation (e.g., HELP [186]), entropy- or structure-guided grouping followed by template merging/refinement (e.g., Lemur [206] and entropy-based correction pipelines [28]), and memory-based reuse to converge toward stable inventories over long streams [57, 106]. The synthesis is that these parsers trade manual labels for approach-level iteration: retrieval/grouping proposes candidate structure, while refinement/memory stabilizes templates across time and workload shifts.

5.3.7 Efficiency-oriented approach designs: grouping, scheduling, inference caching, and agentic decomposition. To better meet production constraints, recent LLM-based parsers increasingly treat efficiency as a primary objective, aiming to minimize both the number and cost of LLM calls. Grouping-based approaches parse only a small set of representative logs and propagate templates to the rest. LogParser-LLM [212] is representative of this family, using grouping and multi-step prompting to balance template fidelity against query volume.

Beyond grouping, execution-layer optimizations target tail latency and inference overhead. EPAS [16] frames online parsing as an asynchronous scheduling problem, while other lines optimize prefix/KV-cache reuse or agentic decomposition (e.g., InferLog [177] and Parse-LLM [153]). A useful synthesis is that these approaches

treat LLM reasoning as a high-quality but expensive primitive, and invest in orchestration layers (grouping, scheduling, caching, and verification) to maximize structured output per LLM token.

Early log parsing work focused on feasibility and prompting sensitivity, while more recent work increasingly centers on reusing approach-level parsing results and efficiency. Importantly, these families of approaches coexist rather than strictly replace one another: supervised adaptation remains attractive when labeled logs are available, prompting dominates rapid deployment, and reuse-centric approaches are essential for online or mass scale. Across them, a unifying design principle is to combine LLM inference with lightweight parsing components, sampling, retrieval, grouping, caching, and validation, that amortize LLM cost while preserving template stability under drift.

5.4 Performance Evaluation Metrics

Evaluation of log parsing has gradually shifted from *message-level* headline scores toward *template-level* metrics, because production logs are heavy-tailed and drift over time: frequent templates can dominate averages and hide failures on rare but diagnosis-critical events. As a result, recent LLM-based parsers increasingly report classic metrics together with template-level ones to better reflect robustness (e.g., LILAC [67], Le et al. [78], LogParser-LLM [212]).

Classic and still common: GA and PA (but frequency-biased). The most widely used metrics remain **Grouping Accuracy (GA)** and **Parsing Accuracy (PA)** because they are simple and comparable across studies [4, 22, 76]. GA evaluates whether each line is assigned to the correct event group, while PA evaluates whether the per-line template exactly matches the oracle [22, 76]. However, both are *message-level* averages: frequent templates dominate, so a parser can score high while failing on the long tail or under drift [67, 78]. They can also blur "grouping is correct" versus "placeholder boundaries are correct", motivating template-level complements [60, 67].

Increasingly adopted for robustness: FGA and FTA. To mitigate frequency bias, many recent parsers report **FGA** (*F1 of Grouping Accuracy*) and **FTA** (*F1 of Template Accuracy*) as *template-level* complements to GA/PA [67, 78, 186, 212]. FGA scores correctness *per template group* (then aggregates via precision/recall-style F1), making rare-template fragmentation/merging visible. FTA is stricter because it also requires the *template text* (static tokens and placeholders) to match, exposing boundary errors that GA/PA can hide [60, 67]. In practice, reporting {GA,PA}+{FGA,FTA} gives both backward comparability and clearer evidence of template-inventory recovery [67, 78].

Diagnostics and niche variants: ED/LCS, GGD/PGD, PTA/RTA, F1, and RI. Several studies add **Edit Distance (ED)** and/or **Longest Common Subsequence (LCS)** between predicted and oracle templates to quantify *how far* near-misses are (useful for diagnosing systematic boundary mistakes), but these should be treated as secondary signals rather than standalone correctness measures [4, 22, 76]. Some papers further introduce **GGD/PGD** to diagnose *error modes*: whether mistakes stem mainly from over-merging/over-splitting (grouping granularity) or from static↔variable boundary mismatches in templates [212]. A few prompting-based parsers report **PTA/RTA** as template-level precision/recall variants that emphasize strict template recovery (often stricter than PA) [126, 188]. When parsing is formulated as token labeling, **F1** evaluates whether parameter boundaries are identified correctly [145]; and for clustering-style evaluation, **RI** (Rand Index) measures pairwise agreement of the predicted vs. oracle partition (together vs. apart) [44, 165].

6 Downstream Log Analysis Tasks

After log parsing transforms raw messages into reusable event templates and parameters, downstream log analysis aims to turn high-volume log events into actionable signals. In this survey, we focus on four representative tasks: **anomaly detection** (flagging abnormal log sequences or windows), **failure prediction** (forecasting

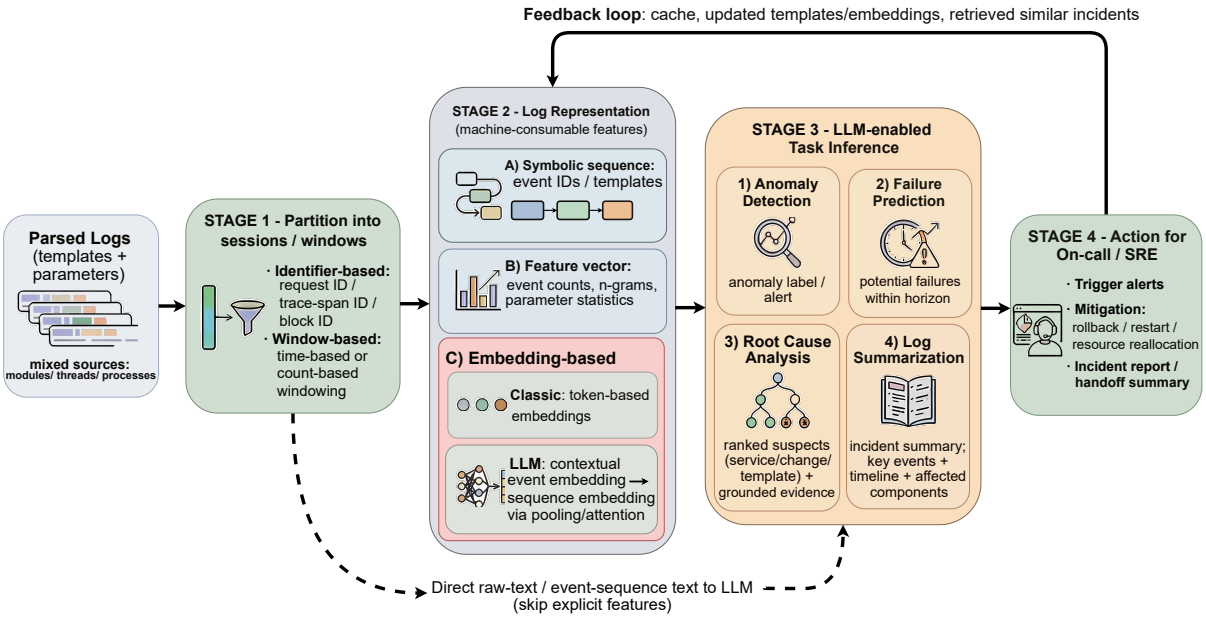

Fig. 4. Common workflow for LLM-enabled downstream log analysis.

failures within a future time horizon), **root cause analysis** (identifying the most likely cause or component of an incident and justifying it with evidence), and **log summarization** (compressing large log windows into concise, operator-friendly summaries).

Challenges motivating the shift toward LLMs. Historically, operators often relied on keyword search and manual inspection, which is fragile under message vocabulary variation and scale [49]. Rule-based analytics can be efficient but are brittle under format drift and hard to adapt across systems, especially when log formats and templates evolve over time [23, 48]. Learning-based detectors and predictors improve automation, yet they often depend on careful feature engineering, stable parsing/representation choices, and sufficient labeled data, and can still be sensitive to log evolution [120]. They may also struggle to provide human-friendly explanations, motivating more interpretable and interaction-oriented log analysis workflows [71, 95]. In the LLM era, researchers increasingly leverage models' semantic generalization to (i) better tolerate heterogeneous phrasing and evolution, and (ii) generate explanation-oriented outputs that help operators understand not only *what* happened but also *why* [22, 44].

A common workflow: partition → representation → task inference → action. In practice, downstream analysis is rarely performed on a monolithic log file. Instead, parsed logs are first *partitioned* into sessions or windows to separate mixed sources (modules/threads/processes) into coherent log sequences. Partitioning can be driven by identifiers (e.g., request ID, trace/span ID, block ID) or by time- or count-based windowing when identifiers are unavailable. After sequences are formed, the pipeline derives a *log representation* to capture semantic and temporal signals of each sequence, and then applies task-specific approaches to trigger alerts, rank suspected causes, predict upcoming failures, or summarize incidents for operators. Figure 4 summarizes this workflow and highlights where LLMs are commonly integrated to improve failure perception, diagnosis, and operator-facing actions.

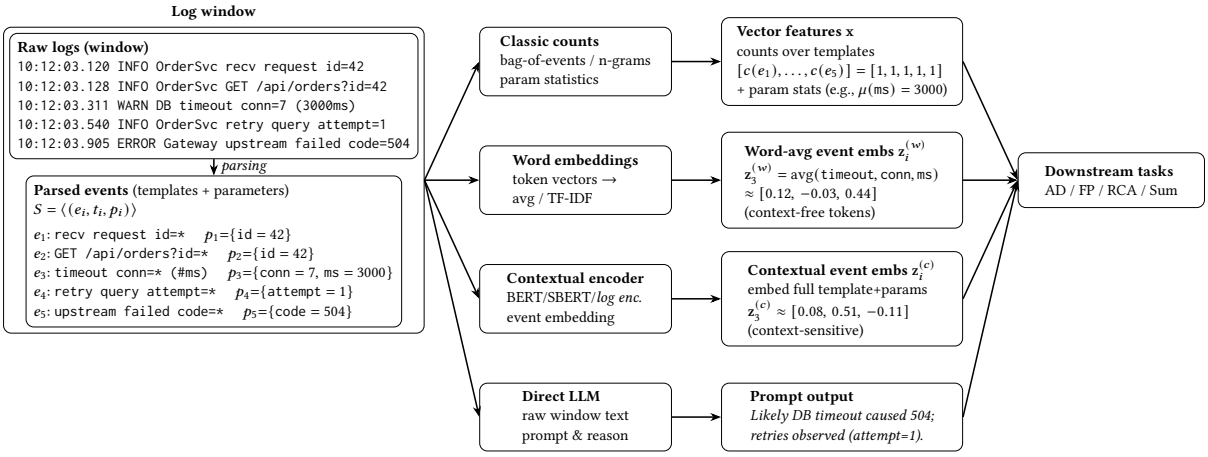

Fig. 5. An illustrative overview of log window representations: raw logs are parsed into templates and parameters, then mapped to (i) classic feature vectors, (ii) context-free word-averaged embeddings, (iii) contextual event embeddings, or (iv) direct prompt-based LLM outputs. Numeric vectors are schematic to highlight different representations.

We organize the rest of this section following the workflow in Figure 4. Section 6.1 discusses **log representation**, which largely determines what any downstream model can learn or infer. Sections 6.2, 6.3, 6.4, and 6.5 then cover four representative LLM-enabled tasks: **anomaly detection**, **failure prediction**, **root cause analysis**, and **log summarization**.

6.1 Log Representation

Given a sessionized (or windowed) log sequence, typically a sequence of parsed events/templates with optional parameters and timestamps, **log representation** maps it into machine-consumable features that preserve task-relevant signals while suppressing nuisance variability. Depending on the downstream task, the representation can be (i) a discrete symbolic sequence (event IDs/templates), (ii) a fixed-length feature vector (counts, n-grams, parameter statistics), or (iii) a continuous embedding sequence or sequence embedding produced by neural encoders. Figure 5 illustrates a log window and how it can be mapped into different representation types.

Traditional representations: statistical, parameter-aware, and semantic sequence features. A common starting point is **statistics-based** representation, which converts each log sequence into a feature vector by counting event occurrences (bag-of-events) or short patterns such as n-grams. This view is compact and efficient, but it can lose semantic similarity between differently worded events and captures ordering only shallowly. A closely related line is **parameter-aware** representation, which augments event statistics with counts/patterns over extracted parameter values (IDs, paths, codes), useful when diagnosis depends on *which entities/values* appear, but potentially brittle under drift and high-cardinality parameters.

The most widely used **semantic** representations in traditional pipelines build embeddings from the message/template text. A typical workflow is to map each token in a log event to a word embedding (e.g., word2vec algorithm [114]), aggregate token vectors into an *event embedding* (often by weighted averaging), and then compose a *sequence embedding* by pooling or concatenating embeddings across events in the window. This family is attractive because it provides a continuous semantic space while remaining lightweight.

LLM-era representations: reuse classic features or switch to contextual encoders. Many recent LLM-based log analytics approaches still reuse classic representations (event-count vectors, parameter statistics, n-grams, or

word-embedding-based features) as inputs to downstream models because they are cheap, stable, and compatible with existing monitoring pipelines. In these hybrid designs, LLMs may serve only as an auxiliary component (e.g., to improve parsing quality or provide explanations), while the anomaly detector/predictor still relies on traditional features and machine-learning models.

At the same time, many recent pipelines adopt *contextual event embeddings*. Instead of averaging context-free word vectors, they embed each entire log event with a pretrained encoder. The main advantage is robustness: contextual encoders model word order and compositional meaning within an event, and they can assign different representations to the same token across different contexts (which is common in fast-evolving software logs) [32, 128]. As a result, semantically equivalent events with heterogeneous phrasing tend to stay closer in the embedding space, while subtle boundary or wording changes are less likely to break downstream models, a property that is particularly useful under drift, long-tail patterns, and limited supervision.

In practice, the workflow is lightweight. Each parsed event (often the template text, optionally concatenated with selected parameters) is converted to a short input string and encoded with a contextual model such as BERT [26] and Sentence-BERT [138], or with log-oriented encoders (e.g., PreLog [77] and OWL [44]) to obtain a fixed-length *event embedding*. For sequence-level tasks, event embeddings within a session/window are then aggregated (e.g., pooling/attention or a lightweight sequential model) into a *window representation*. Downstream modules use these representations either for (i) similarity-based retrieval/matching (nearest-neighbor search for related events/windows) or (ii) training detectors/predictors for anomaly detection and failure prediction, often with improved tolerance to wording variation compared to word-embedding pipelines.

Unified log representations via large-scale pretraining. Beyond "using a general-purpose encoder," BigLog [166] advocates *unsupervised large-scale pre-training* to learn a *unified* representation space for logs. The core idea is to pretrain on massive, heterogeneous log corpora so that a single encoder can produce transferable embeddings across systems and log styles, reducing the reliance on per-system feature engineering or task-specific representation tuning. This perspective is especially relevant in LLM-era pipelines where the bottleneck often shifts from model capacity to *representation compatibility*: when logs change frequently or when operators need cross-service retrieval and correlation, a unified embedding space enables (i) more reliable similarity search over historical windows, (ii) easier cross-system transfer/bootstrapping under limited labels, and (iii) a common interface that multiple downstream tasks (anomaly detection, failure diagnosis, summarization) can share. In practice, BigLog-style pretraining can be viewed as a middle ground between "classic features" and "prompt on raw text": it keeps inference efficient (fixed-length embeddings) while improving semantic robustness and portability at scale.

Skipping explicit features: direct raw-text or event-sequence text as input. Finally, some LLM-based approaches partially or fully bypass handcrafted feature construction by feeding an LLM the *raw log window* (or a text form of the *event sequence*) and prompting it to perform the downstream task directly. This design can preserve richer context and enable explanation-oriented outputs, but it introduces new constraints: long contexts require careful windowing, summarization, or retrieval to control noise and cost, and online deployment may demand additional compression or caching mechanisms. Overall, representation choices trade off efficiency, robustness, and fidelity. Thus, many recent approaches combine compact classic features with contextual embeddings or selective raw-text reasoning, depending on task requirements and operational budgets.

6.2 Anomaly Detection

6.2.1 Task definition. Log anomaly detection identifies *unexpected* or *undesired* system behavior patterns from logs. Such anomalies typically indicate potential faults, misconfigurations, performance regressions, or other conditions that violate the expected normal execution, and thus serve as early signals for reliability triage.

Table 4. Summary of LLM-based anomaly detection studies organized by task category. (continued in Table 5)

Paper	Supervised?	Train?	Dataset(s) evaluated	Parsing?	Metric(s)	Base Model(s)
Discriminative & Representation Learning (Encoder-based)						
HitAnomaly [58]	Supervised	Yes	HDFS, BGL, OpenStack	Yes	F1	BERT
SwissLog [83]	Supervised	Yes	HDFS, BGL	Yes	F1	BERT
Ott et al. [120]	Semi-Sup	Yes	OpenStack	Yes	F1	BERT
LogBert [42]	Semi-Sup	Yes	HDFS, BGL, Thunderbird	Yes	F1	BERT
NeuralLog [74]	Supervised	Yes	HDFS, BGL, TB, Spirit	No	F1	BERT
BERT-Log [15]	Supervised	Yes	HDFS, BGL	Yes	F1	BERT-110M
LogST [209]	Semi-Sup	Yes	HDFS	Yes	F1	Sentence-BERT
Prog-BERT-LSTM [148]	Semi-Sup	Yes	HDFS, BGL, Thunderbird	Yes	F1	BERT
HilBERT [59]	Supervised	Yes	HDFS, BGL	Yes	F1	BERT
LogBP-LORA [50]	Supervised	Yes	BGL, Thunderbird	Yes	F1	BERT-base
LogDAPT [210]	Supervised	Yes	HDFS, BGL, Thunderbird	Yes	F1	BERT
LogADSBERT [55]	Semi-Sup	Yes	HDFS, Hadoop, OpenStack	Yes	F1	Sentence-BERT
Karlsen et al. [70]	Unsup	No	ECML, CSIC, Access	Yes	F1	DistilRoBERTa
FastLogAD [93]	Semi-Sup	Yes	HDFS, BGL, Thunderbird	Yes	F1	ELECTRA
LAnoBERT [80]	Semi-Sup	Yes	HDFS, BGL, Thunderbird	No	F1, AUC	BERT
Corbelle et al. [20]	Supervised	Yes	HDFS	Yes	F1	BERT-base
Zhou et al. [216]	Supervised	Yes	HDFS, BGL	Yes	F1	BERT
LogELECTRA [191]	Semi-Sup	Yes	BGL, Thunderbird, Spirit	No	F1	ELECTRA
LogFiT [2]	Semi-Sup	Yes	HDFS, BGL, Thunderbird	No	F1	BERT
LogRoBERTa [160]	Supervised	Yes	HDFS, BGL	No	F1	RoBERTa
ADALog [130]	Unsup	Yes	BGL, Thunderbird, Spirit	No	F1	DistilBERT
Thali et al. [167]	Supervised	Yes	SWaT, Morris ICS	No	F1	BERT
APT-LLM [7]	Semi-Sup	Yes	Android, Linux, Win	Yes	AUC	BERT
Entezami et al. [31]	Supervised	Yes	KDD99, UNSW, CICIDS	Yes	F1	BERT
AnomalyExplainerBot [5]	Supervised	Yes	HDFS	Yes	F1	RoBERTa
Horváth et al. [54]	Supervised	Yes	BGL, Thunderbird	Yes	F1, Acc	LLaMA
Generative Forecasting & Next-Token Prediction (Decoder-based)						
LogGPT (BigData) [169]	Supervised	Yes	HDFS, BGL, Thunderbird	No	F1	GPT-2
LogSense [155]	Semi-Sup	Yes	-	Yes	-	GPT-2
PreLog [77]	Supervised	Yes	HDFS, BGL, Spirit	No	F1	FLAN-T5 (140M)
Bogdan et al. [10]	Supervised	Yes	ECU Comm. Logs	Yes	Top-k	Qwen2-0.4B
LogLM [97]	Supervised	Yes	BGL, Spirit	Yes	F1	LLaMA-2-7B
NLPLog [65]	Supervised	Yes	BGL, Spirit	Yes	F1	LLaMA-2-7B

Given a sessionized (or windowed) log sequence $S = \langle (e_i, t_i, p_i) \rangle_{i=1}^{n}$, where e_i is a parsed event/template (or the raw log message), t_i is the timestamp, and p_i are optional parameters, the task outputs either (i) an anomaly score $a(S)$ or a binary label $y \in \{0, 1\}$ indicating whether the sequence is abnormal, and optionally (ii) localized evidence such as suspicious positions $\mathcal{I} \subseteq \{1, \ldots, n\}$ (or key templates/snippets) to support debugging.

In practice, anomaly detection is usually *reactive*: it flags abnormal states or behavior patterns in observed logs (often in unsupervised or weakly supervised settings), without necessarily predicting whether/when a concrete failure will occur.

Table 5. Summary of LLM-based anomaly detection studies organized by task category. (continued from Table 4.)

Paper	Supervised?	Train?	Dataset(s) evaluated	Parsing?	Metric(s)	Base Model(s)
Prompt-Based & In-Context Learning (Zero/Few-shot)						
LogPrompt (IJCNN) [204]	Supervised	Yes	HDFS, BGL	No	F1, Acc	BERT-large
LogPrompt (ICPC) [95]	Unsup	No	HDFS, BGL	Yes	F1	ChatGPT
Setu et al. [108]	Supervised	Yes	Apache, BGL, HDFS	Yes	F1, AUC	GPT-3
LogGPT (HPCC) [132]	Supervised	No	BGL, Spirit	Yes	F1	GPT-3.5-turbo
Egersdoerfer et al. [29]	Supervised	No	Lustre	No	F1	GPT-3.5-turbo
Owl [44]	New Model	Yes	BGL, Spirit	Yes	F1	Owl (LLaMA)
Fariha et al. [34]	Unsup	Yes	HDFS	Yes	F1	GPT-3.5-turbo
VMFT-LAD [144]	Semi-Sup	Yes	VM/Server failures	Yes	AUC	GPT-3.5-turbo
FLEXLOG [46]	Supervised	Yes	ADFA, SynHDFS	Yes	F1	Mistral-22B
CloudAnoAgent [219]	Supervised	No	CloudAnoBench	No	Acc	Gemini-2.5
Retrieval-Augmented Generation (RAG)						
RAPID [117]	Semi-Sup	No	HDFS, BGL, Thunderbird	No	F1	BERT
LogRag [205]	Semi-Sup	Yes	BGL, Spirit	Yes	Prec, Rec, F1	GPT-3.5-turbo, Mistral-7B-Instruct
RAGLog [123]	Semi-Sup	No	BGL, Thunderbird	No	F1	RAG Model
XRAGLog [201]	Semi-Sup	No	HDFS, BGL, Thunderbird	Yes	F1	GPT-3.5
Hybrid & Collaborative (Small Model + Large Model)						
LLMeLog [47]	Supervised	Yes	HDFS, BGL, Thunderbird	Yes	F1	ChatGPT + BERT
LogFormer [43]	Supervised	Yes	HDFS, BGL, TB, GAIA	No	F1	S-BERT + ChatGPT
CLogLLM [139]	Unsup	Yes	HDFS, BGL, Thunderbird	No	F1	Qwen + GPT-3.5
LogLLM [41]	Supervised	Yes	HDFS, BGL, TB, Liberty	No	F1	BERT + LLaMA-3
AdaptiveLog [104]	Supervised	Yes	BGL, Thunderbird	No	F1	BERT + ChatGPT
LogRules [60]	Supervised	Yes	BGL, Spirit	No	F1	GPT-4o + LLaMA-3
LogADRef [92]	Supervised	Yes	BGL, HDFS, Spirit, TB	Yes	F1	RoBERTa + GPT2
SemiRALD [159]	Supervised	Yes	HDFS, BGL	Yes	F1	ChatGPT + BERT
LogSynergy [156]	Supervised	Yes	BGL, Thunderbird, Spirit	Yes	F1	ChatGPT-4o
CLSLog [183]	Supervised	Yes	BGL, Zookeeper	No	F1	BERT + Qwen
LEMAD [64]	Semi-Sup	Yes	State Grid Corp	Yes	F1	BERT + GPT-4o
CoLA [218]	Supervised	Yes	HDFS, BGL, Spirit	Yes	F1	LLaMA-3 + GPT-4o
LUK [101]	Supervised	Yes	BGL, Spirit, Thunderbird	Yes	F1	GPT-3.5 + Expert
SemiSMAC [158]	Supervised	Yes	BGL, HDFS, Spirit	Yes	F1	GPT-4-turbo
Agentic, Reasoning & Neuro-Symbolic Frameworks						
Audit-LLM [152]	Unsup	No	CERT r4.2/5.2	No	F1	GPT-3.5 (Agent)
WebNorm [91]	Semi-Sup	No	TrainTicket, NiceFish	Yes	F1	GPT-3.5 (Agent)
LogRESP-Agent [79]	Semi-Sup	Yes	EVTX-ATTACK	No	F1, TPR	Gemini-2.0-Flash
Song et al. [154]	Supervised	Yes	CERT v6.2	No	F1	ChatGLM-6B
SHIELD [157]	Supervised	Yes	DARPA-E3, ATLASv2	Yes	Prec	DeepSeek-R1
LogReasoner [103]	Supervised	Yes	BGL, Spirit	No	F1	Qwen2.5-1.5B
Data Generation, Benchmarks & Visions						
Karlsen et al. [71]	Supervised	Yes	CSIC, Thunderbird, BGL	Yes	F1	RoBERTa/GPT
LogEval [22]	Supervised	No	BGL, Thunderbird	No	F1	GPT-4
AD-LLM [192]	Semi-Sup	Yes	News, Reviews, Spam	No	AUC	LLaMA-3/GPT-4o
AnomalyGen [84]	Supervised	Yes	Hadoop, HDFS	Yes	F1	GPT-4o

6.2.2 LLM-based Approaches for Anomaly Detection. Table 4–5 summarizes LLM-based anomaly detection studies. Across these works, a common pipeline is: (i) partition logs into sessions/windows, (ii) encode each sequence into a representation (templates, embeddings, or raw text), (iii) compute an anomaly signal (score/label), and (iv) optionally return *evidence* (highlighted lines, retrieved neighbors, or explanations) to support triage. Existing approaches mainly differ along four design axes: **(1) representation interface** (parsed templates vs. raw text), **(2) supervision setting** (fully supervised with both normal/anomalous labels vs. semi-supervised/one-class settings that train mostly or only on *normal* data, vs. unsupervised settings), **(3) adaptation/training strategy** (fine-tuning/prompt-tuning vs. keeping the LLM frozen and using it in a training-free manner), and **(4) inference form** (discriminative classification, likelihood/perplexity scoring, prompting/ICL, retrieval, or agentic reasoning). Notably, prompting/in-context learning (ICL) is largely orthogonal to supervision, and "training" is not synonymous with "supervision": (i) many methods *train* models without using task labels (e.g., further pretraining or contrastive learning on related unlabeled logs/corpora), and (ii) some methods are *training-free* but still supervised at inference time by using labeled exemplars as demonstrations in ICL. Note that boundaries are not strict: many systems combine multiple mechanisms (e.g., encoder embeddings + prompting, or retrieval + generation).

Discriminative and representation-learning encoders. A large body of work uses pretrained encoders (typically BERT-like) to map log events or event sequences into embeddings, followed by a classifier or sequence model for anomaly prediction. Representative examples that operate on parsed templates and learn discriminative decision boundaries include HitAnomaly [58], SwissLog [83], and HilBERT [59]. Several studies adapt or optimize the encoder/sequence modeling pipeline, such as BERT-Log [15], Prog-BERT-LSTM [148], and the semantic hierarchical compaction + BERT-based detection by Corbelle et al. [20]. Other works emphasize robustness/transfer across systems and unstable log vocabularies; for instance, Ott et al. study robustness and transferability with pretrained language models [120], and APT-LLM applies embedding-based detection to advanced persistent threats in cyber logs [7]. Some papers explicitly investigate LLM-assisted *parsing + embedding* pipelines before detection (e.g., Zhou et al. [216]), while others evaluate broader log/security settings with encoder backbones (e.g., Thali et al. [167] and Entezami et al. [31]). Finally, as LLM-era anomaly detection expands beyond classic system logs, recent large-scale studies also include LLaMA-family backbones in supervised pipelines (e.g., Horváth et al. [54]) and explanation-oriented add-ons (e.g., AnomalyExplainerBot [5]).

Masked-LM and self-/semi-supervised normality modeling. A closely related family models *normal* log patterns using self-supervised objectives and flags deviations as anomalies, which is well-suited to rare-anomaly regimes. LogBert [42] and LAnoBERT [80] exemplify masked-language-model training on normal sequences, producing anomaly scores via reconstruction/prediction errors. Recent variants continue this direction with alternative backbones and training strategies, such as LogELECTRA [191], LogFiT [2], and ADALog [130]. FastLogAD [93] further augments semi-supervised learning by generating pseudo anomalies to strengthen discrimination. In practice, these methods often reduce label requirements but still rely on careful sessionization/windowing and calibration to control false positives across evolving software versions.

Reducing reliance on log parsing: from templates to raw text. Because parsing errors and format drift can directly harm downstream detectors, multiple works explicitly minimize or remove template parsing. NeuralLog [74] studies anomaly detection without conventional log parsing, and LogRoBERTa [160] further reduces parser reliance via hybrid language modeling. In addition, several prompt- or LLM-assisted pipelines still use lightweight normalization (e.g., regex-based variable masking) as a middle ground between full template parsing and raw text, especially when deploying at scale under frequent log evolution.

Decoder-style generative forecasting and likelihood-based scoring. Instead of classification, decoder LMs treat anomaly detection as a *language modeling* problem: a sequence is anomalous if it is hard to predict under a model trained (or adapted) on normal behavior. LogGPT [169] is a representative GPT-style next-token prediction

approach, while PreLog [77] explores a pretrained model tailored for log analytics tasks (including anomaly detection) using a compact generative backbone. Instruction-centric systems also evaluate anomaly detection within a broader log analysis interface (e.g., LogLM [97] and knowledge-augmented adaptation in NLPLog [65]). Beyond system logs, "Good Enough to Learn" investigates anomaly detection in ECU logs under unreliable labels with small LMs [10], highlighting a practical trend toward smaller, cheaper generative models for edge/embedded scenarios.

Prompting and in-context learning for training-free detection and interpretation. With instruction-tuned LLMs, several works bypass explicit model training and instead prompt an LLM to classify sequences or explain suspicious patterns. LogPrompt-style prompt engineering emphasizes interpretability and online usage [95, 96], while feasibility studies explore ChatGPT-style models for anomaly detection under different settings and log sources [29, 108, 132]. Some lines treat LLMs as an *operations foundation model* that supports multiple tasks, including anomaly detection, under a unified interface (e.g., OWL [44]). At the benchmark level, large-context prompting is also evaluated in broad suites (e.g., LogEval [22]), clarifying how far prompting alone can go without task-specific training.

Retrieval-augmented detection: comparing against a memory of normality. Retrieval-based methods treat anomaly detection as *nearest-neighbor comparison* in an embedding space of normal sequences and often provide evidence by returning similar historical cases. RAPID [117] embeds normal windows using a pretrained encoder and flags anomalies by distance/neighborhood structure, while RAGLog [123] frames detection through retrieval-augmented generation by retrieving similar normal cases and judging the target's consistency. XRAGLog [201] further targets resource efficiency and context awareness in RAG-style pipelines. This family is especially compatible with semi-supervised settings (normal-only data) and naturally supports operator-facing justifications through retrieved exemplars.

Hybrid and collaborative frameworks: small models for scoring, LLMs for enrichment, verification, or rules. Many practical designs combine a lightweight detector with an LLM component that improves robustness or interpretability. LLMeLog [47] enriches log events with LLM-derived semantics before training a smaller classifier; LogFormer [43] combines representation learning with LLM assistance; and LogLLM [41] explores joint use of encoders with larger LMs. AdaptiveLog [104] explicitly frames anomaly detection as collaboration between large and small models, while CoLA [218] studies model collaboration at scale with multiple LLMs. Knowledge and control signals are also injected via rules or domain constraints (e.g., LogRules [60]), and tuning procedures aim to stabilize performance (e.g., SemiSMAC [158] and SemiRALD [159]). Finally, cross-system generalization and "new system" adaptation are increasingly important, motivating transfer-focused designs such as LogSynergy [156].

Agentic and multi-step reasoning for complex security/incident contexts. A growing line of work frames anomaly detection as a multi-step investigation that iteratively retrieves context, applies tools, and produces both predictions and narratives. Examples include multi-agent collaboration for threat/anomaly scenarios (Audit-LLM [152]), consistency-based detection with evidence building (WebNorm [91]), and recursive/agentic pipelines for context-aware detection and response (LogRESP-Agent [79]). Related systems emphasize turning alerts into operator-facing intelligence (SHIELD [157]) or improving reasoning structure and controllability (LogReasoner [103]), while insider-threat anomaly detection appears as another domain where agentic reasoning is natural (Song et al. [154]). Agentic methods can improve interpretability and reduce analyst workload, but introduce additional latency/cost and require careful guardrails for reliable deployment.

Benchmarks and data generation. As the space expands, benchmarks and data-centric efforts have become important for fair comparison and stress-testing. Karlsen et al. benchmark LLMs for log analysis tasks including anomaly detection [71], and LogEval provides a broader benchmark suite for LLM-based log analysis [22]. Beyond

logs, AD-LLM benchmarks LLM-based anomaly detection across multiple text anomaly domains [192], helping disentangle what improvements come from stronger backbones versus prompting/retrieval. Complementarily, AnomalyGen [84] explores LLM-driven generation of anomalous log sequences, reflecting an emerging direction where synthetic anomalies and augmentation are used to improve coverage of rare patterns and unstable logs.

Across LLM-based anomaly detection studies, the key shift is not merely higher accuracy, but treating anomalies as *evidence-grounded inconsistencies* with expected behavior (e.g., deviations from retrieved "normal" cases, violations of explicit rules, or unusually high calibrated anomaly scores). The strongest systems therefore constrain LLMs with stable interfaces (templates/embeddings/normalized text) and use LLMs mainly to explain, verify, or enrich lightweight detectors. Meanwhile, LLMs do not remove the ambiguity of what counts as an anomaly; instead, they shift effort from feature design to normality curation, context selection (windowing/retrieval), and operational guardrails that control false alarms and cost—making hybrid, evidence-driven workflows the dominant deployment pattern.

6.3 Failure Prediction

6.3.1 Task definition. Log-based failure prediction is a proactive early-warning task: it forecasts whether the system (or a component) will experience a *future failure* within a horizon, so operators can take preventive actions (e.g., mitigation, rollback, or resource reallocation) before the system enters an unrecoverable or user-visible failure state. Given logs observed up to the current time t (often represented as a session/window $S_{t-L:t}$ or a prefix of an ongoing session), the task outputs a probability $\Pr(\text{failure in } [t, t+H])$ (or a binary label) for a future horizon H, and sometimes additional signals such as predicted time-to-failure or early-warning decisions under different lead-time budgets.

Compared with anomaly detection, failure prediction is explicitly *forward-looking* and tied to an operational failure definition (e.g., crash, outage, severe SLO violation, incident ticket). Depending on the environment, prediction may focus on (i) failures of similar components in homogeneous systems (sequence-centric modeling), or (ii) failures driven by interactions among heterogeneous components (leveraging cross-component correlations).

6.3.2 LLM-based Approaches for Failure Prediction. Compared with anomaly detection, failure prediction must explicitly optimize lead time and horizon awareness: a useful predictor should raise early warnings sufficiently ahead of a failure event while controlling false alarms. Consequently, papers often report ranking/probabilistic metrics (e.g., AUC) and sometimes assess whether the model can provide actionable context (time-to-failure, likely causes).

Generative prediction of crash events and causes. CrashEventLLM casts failure prediction as a text generation problem: given preceding logs, the model predicts forthcoming crash-related information (e.g., time and cause) using an instruction-tuned LLaMA backbone, and evaluates generation quality with ROUGE-style metrics. [116] This formulation is attractive when operators want human-consumable explanations, but it also raises a methodological question: lexical-overlap metrics (e.g., ROUGE) may not faithfully reflect correctness of timestamps/causal factors, so future work may benefit from structured evaluation (e.g., exact match on failure type/time bins, and evidence grounding).

Discriminative early warning from language-model representations. FALL proposes a language-model-based detector that outputs an early-warning score (reported with AUC-style evaluation) to detect imminent failures in large-scale systems. [63] In this design, the PLM (e.g., ELECTRA-style encoders) provides a robust representation of evolving log patterns, and the predictor focuses on separating pre-failure prefixes from normal operation. Conceptually, this parallels semi-supervised anomaly detection, but with labels aligned to future failure windows (pre-failure vs. non-pre-failure), making horizon definition and data leakage control (e.g., avoiding post-failure artifacts) especially important.

Leveraging operational context beyond application logs. Combining LLMs and shell logs for backup-failure prediction highlights another practical pattern: operational failures are often best predicted using heterogeneous telemetry, where shell commands, job traces, and system events provide context that is not present in application logs alone. [37] Here, an LLM can serve as a semantic integrator for noisy, free-form operational traces, potentially improving feature robustness and interpretability.

From anomaly detection to proactive fault tolerance. VMFT-LAD (Virtual Machine Proactive Fault Tolerance Using Log-Based Anomaly Detection) connects anomaly scoring with proactive mitigation for virtualized environments, treating detected abnormal patterns as early signals to trigger fault-tolerance actions (e.g., proactive migration or resource adjustment).[144] Although framed around anomaly detection, the operational goal aligns with failure prediction: catch the system early enough to intervene. This line suggests a promising integration point: LLM-based detectors could be paired with decision policies (rule-based or learned) that explicitly optimize lead time, cost, and risk.

The current failure prediction literature using LLMs is still sparse and heterogeneous. Most works either (i) reuse LLM representations for discriminative early warning, or (ii) exploit instruction-following to generate human-readable failure explanations. A key open challenge is establishing consistent evaluation that jointly measures predictive quality, timeliness, and actionability, under realistic constraints such as log evolution, partial observability, and cross-component dependency.

6.4 Root Cause Analysis / Failure Diagnosis

6.4.1 Task definition. Root cause analysis (RCA) / Failure diagnosis aims to identify the most likely underlying cause(s) of an observed incident and provide actionable, operator-facing evidence. The goal is not only to name a suspect (e.g., component/change/event type), but also to *justify* it with grounded signals so that time-to-mitigation can be reduced.

Given an incident context (e.g., an anomalous time window or a set of correlated sessions), inputs commonly include (i) logs from multiple components/services within a time range, and optionally (ii) service topology/dependency information, deployment/change metadata, and alerts/metrics describing the incident. The output is typically a ranked list of candidate causes (top-k suspects), together with supporting evidence such as representative templates/snippets, affected components, and (when available) an explanatory chain linking symptoms to causes.

6.4.2 LLM-based Approaches for Root Cause Analysis. Table 6 summarizes LLM-based root cause analysis (RCA) and closely related failure diagnosis studies. Overall, these works share a common goal: given heterogeneous operational evidence (logs and often metrics/traces, code, tickets, or knowledge bases), produce an actionable RCA output such as *(i)* a root-cause description or reason, *(ii)* localization to components/configurations, and sometimes *(iii)* remediation guidance and supporting evidence. Across studies, the main design differences are **(1) how the LLM is used** (prompting/ICL vs. agentic tool use vs. trained task model), **(2) what external context is integrated** (repositories, databases, SOPs, graphs, knowledge bases), and **(3) what the output is optimized for** (free-form explanation vs. structured labels/locations vs. resolutions).

Prompting and in-context learning for incident RCA. A pragmatic line treats RCA as a text understanding and summarization problem and relies on prompting/ICL to map incident artifacts to root-cause statements. Chen et al. [17] perform automatic RCA for cloud incidents by consuming multiple incident signals (e.g., error logs, stack traces, socket metrics) and outputting both explanations and categories, evaluated with F1. Similarly, Zhang et al. [208] study ICL-based root causing for cloud incidents using incident metadata (e.g., title/summary) and report a suite of generation-oriented metrics. In privacy- or token-limited settings, InsightAI [30] frames RCA as interactive diagnosis over large private logs, producing operator-facing artifacts (e.g., flame graphs

Table 6. Summary of LLM-based root cause analysis and failure diagnosis studies organized by category.

Paper	Train?	Dataset(s) type	Output	Metric(s)	Base Model(s)
Prompting & In-Context Learning (ICL) for RCA / Diagnosis					
Chen et al. [17]	Yes	log, traces, metrics	Explanation; category	Micro and macro F1	GPT-4
Shan et al. [146]	No	Log	Root-cause configuration location	Accuracy; FP	GPT-4
Zhang et al. [208]	No	Incident title, Incident summary (logs, traces, etc.)	Root cause	ROUGE-L, ROUGE-1, METEOR, GLEU, BERTScore, Nubia	GPT-3.5-turbo, GPT-4
InsightAI [30]	No	Log	Flame graph; user Q&A chatbot	F1	GPT-4o
ScalaLog [200]	No	Log	Root-cause description	F1	GPT-3.5-turbo
Agentic & Tool-Augmented RCA (Multi-step Reasoning)					
RCAgent [179]	No	Log, codebase	Root causes, solutions, evidence, responsibilities	METEOR, NUBIA, BLEURT, BARTScore, EmbScore	Vicuna-13B-V1.5-16K, GTE-LARGE (embedding)
OPENRCA [189]	No	Log, traces, metrics	Root cause reasons and components	Accuracy	Claude 3.5
TAMO [207]	Yes	Log, traces, metrics	Root cause localization; fault type classification	Acc@k; MiPr, MaPr, MiRe, MaRe, MiF1, MaF1	GPT-4
Flow-of-Action [127]	No	Log, traces, metrics	Root cause localization; fault type classification	Root cause location accuracy (LA); root cause type accuracy (TA)	GPT-4-Turbo
RCLAgent [202]	No	Metric, trace log	Root cause localization	Recall@k; MRR	Claude-3.5-sonnet
MicroRCA-Agent [164]	Yes	Log, traces, metrics	Root-cause	Top@K	deepseek-v3
Knowledge / Graph / Code-Augmented Diagnosis					
ForenSiX [61]	No	Log	Context extraction, data dependency graph, diagnosis	Accuracy	GPT-4o
COCA [88]	No	Log, codebase, trace	Root-cause	BLEU-4, ROUGE-1, METEOR, Semantics, Usefulness, Exact Match, Top-K	GPT-4o
AetherLog [21]	Yes	Log	Root-cause category prediction	F1	GPT-4o
General Log Diagnosis: Fault Type / Resolution / Understanding					
LogSage [190]	No	Log	Root causes; solutions	Precision, Recall, F1	GPT-4o, Deepseek V3, Claude-3.7-Sonnet
Herrmann et al. [52]	Yes	Issue description, log files, engineer communications	Root cause description	MSS	MIXTRAL-8X7B
Huang et al. [56]	Yes	Log	Fault-indicating description; fault-indicating parameter	Precision, Recall, F1	UniXcoder
LogLM [97]	Yes	Log	Log-problem-resolution	BLEU, ROUGE-1, ROUGE-2, ROUGE-L	LLaMA2-7B
LogReasoner [103]	Yes	Log	Failure type	Accuracy; Weighted-F1	Qwen2.5-1.5B
Ji et al. [65]	Yes	Log	0/1 failure detection; fault type	F1	LLaMA-2-7B
LogExpert [173]	Yes	Log, StackOverflow issues	Executable resolution steps, command filling	BLEU-4, ROUGE-L	GPT-3.5-turbo, GPT-4
KnowLog [102]	Yes	Log	Cause ranking	Accuracy; Weighted F1	BERT
AdaptiveLog [104]	Yes	Log	Failure type	Recall@k	bert, chatgpt
Taheri et al. [162]	Yes	Log	Fault classification	F1	RoBERTa, BigBird, Flan-T5
Benchmarks & Evaluation Suites					
LogEval [22]	Yes	Log	Fault types	Accuracy; F1	GPT-4

and Q&A-style assistance) and evaluating quality with F1. These ICL-style systems are attractive due to low engineering overhead (no task-specific training required), but their performance is often bounded by context selection, prompt robustness, and the fidelity of the provided evidence.

Agentic and tool-augmented RCA for complex systems. When RCA requires multi-step investigation (e.g., correlating signals, fetching relevant traces, querying databases, or checking code ownership), many works move from single-pass prompting to *agentic* pipelines. RCAgent [179] is representative: it uses autonomous agents and tool augmentation over logs and external artifacts (e.g., databases and code repositories) to produce root causes, solutions, evidence, and responsibilities, and evaluates with multiple semantic similarity metrics. OPENRCA [189] explicitly benchmarks the ability of agentic LLM systems to locate root causes from observability data (logs/metrics/traces), emphasizing accuracy on components and reasons. Other methods specialize in structured outputs and fine-grained localization: TAMO [207] targets cloud-native RCA with tool-assisted agents over multi-modality observations and evaluates localization via Acc@k alongside fault-type classification metrics, while Flow-of-Action [127] incorporates SOP guidance to improve location/type accuracy. RCLAgent [202] explore multi-agent recursion-of-thought for microservice localization, reporting ranking-style metrics (Recall@k, MRR). MicroRCA-Agent [164] further integrates an anomaly-identification stage (training an Isolation Forest) before agentic RCA, reflecting a common practical pattern: *small models for fast screening, agents for evidence-driven explanation and attribution.* Overall, agentic approaches tend to improve completeness and traceability of RCA, but introduce added latency/cost and require careful guardrails to avoid brittle tool use or overconfident narratives.

Knowledge-, graph-, and code-augmented RCA. Another prominent direction strengthens RCA by explicitly modeling dependencies and injecting structured knowledge. ForenSiX [61] combines context extraction with a data dependency graph for network diagnostics and reports accuracy, illustrating how graph structure can constrain reasoning and improve interpretability. COCA [88] augments RCA with code knowledge (logs + codebase + stack traces) to produce root-cause outputs and evaluates with both lexical and human-centric measures, reflecting the intuition that many failures are only disambiguated with code-level context. AetherLog [21] integrates knowledge graphs with LLMs for log-based RCA category prediction, evaluated with F1. These systems highlight that, beyond raw language ability, *external structure* (code, graphs, KBs) is a key ingredient for reliable RCA—especially when symptoms are ambiguous or multiple plausible causes exist.

RCA-adjacent diagnosis: fault types, resolutions, and log understanding. Several studies expand beyond pinpointing a cause and instead aim to complete the diagnosis loop: identifying fault types, extracting fault-indicating evidence, or generating resolutions. Huang et al. [56] extract fault-indicating descriptions and parameters from logs (precision/recall/F1), which can be viewed as a complementary capability that improves downstream RCA quality by distilling salient evidence. LogExpert [173] focuses on generating executable resolution steps grounded in external knowledge (StackOverflow), while LogLM [97] frames log analysis as instruction-following and outputs problem resolutions, evaluated with text generation metrics. For broader diagnostic labeling, LogReasoner [103] predicts failure types, and Ji et al. [65] adapt LLMs to log analysis with interpretable domain knowledge for failure detection and fault typing (F1). At the representation level, KnowLog [102] enhances log understanding and supports cause ranking, and AdaptiveLog [104] emphasizes collaboration between small and large models for failure-type identification (Recall@k). In telecom diagnostics, Taheri et al. [162] study domain-tailored models for log mask prediction used for fault classification (F1), reinforcing the importance of domain adaptation in operational contexts. Finally, in less standardized environments, Herrmann et al. [52] show a case study for robotics support tickets (including logs and human communications), indicating RCA often spans heterogeneous, partially structured evidence beyond pure log streams.

Benchmarks and evaluation: what "good RCA" means. As methods diversify, benchmark efforts help clarify evaluation targets. LogEval [22] provides a broader suite for LLM-based log analysis (including fault types), using accuracy/F1, and complements RCA-focused studies by stressing comparability and coverage. Across the

table, evaluation remains fragmented: incident RCA often uses semantic similarity metrics; localization prefers rank/Acc@k; categorization uses accuracy/F1; and resolution generation uses ROUGE/BLEU. This metric diversity reflects different operational definitions of RCA (explanation vs. localization vs. remediation), and suggests that future work may benefit from task-specific, operator-aligned evaluation protocols.

Across RCA studies, LLMs matter less as standalone "reasoners" and more as *orchestrators* that unify heterogeneous evidence (logs/metrics/traces/code/KBs) into a structured hypothesis with supporting traces. The most reliable designs constrain generation through retrieval, tools, SOPs, or graphs, shifting the core challenge from modeling to *evidence selection, grounding, and verifiable outputs*—and making hybrid, tool-augmented pipelines the dominant pattern.

6.5 Log Summarization

6.5.1 Task definition. **Log summarization** compresses high-volume logs into a concise, human-consumable artifact that helps engineers quickly understand *what happened* and *what matters*. It is commonly used for on-call handoff, incident reports, and rapid situational awareness, where reading raw logs is too costly.

Given a set/sequence of logs associated with a session/window (often an incident window), the task outputs a natural-language summary and/or a compact structured summary (e.g., a short timeline of key events, prominent error templates, involved components, and extracted entities such as IDs/paths). Two common instantiations are: *extractive* summarization (select representative lines/templates) and *abstractive* summarization (generate a narrative description), often combined with grouping-by-time/component to keep summaries faithful and easy to scan.

6.5.2 LLM-based Approaches for Log Summarization. Compared with anomaly detection and RCA, log summarization is one of the most *LLM-native* log analysis tasks: its primary deliverable is a human-readable artifact, and usefulness depends on semantic coherence, correct attribution, and actionable compression rather than only classification accuracy. Across the collected studies, LLM-based log summarization typically follows a "*structure-then-generate*" pattern: (i) pre-organize raw logs (by time, component, severity, or templates), (ii) select salient evidence (representative lines, errors, entities), and then (iii) generate an abstractive narrative or a structured synopsis (timeline, key events, involved modules), often with faithfulness constraints.

Benchmarks and evaluation: summarization as a first-class log task. LogEval [22] explicitly includes *log summarization* in a broader benchmark suite for LLM-based log analysis. It evaluates summary quality using both lexical overlap (e.g., ROUGE) and task-oriented correctness signals (e.g., accuracy/F1-style measures), and compares strong proprietary models against a wide range of open-source chat/instruction models. This benchmark perspective highlights two recurring issues in summarization: (i) *metric mismatch*—ROUGE-like scores only partially reflect incident usefulness; and (ii) *context limitation*—performance depends heavily on which log subset is provided to the model, motivating retrieval or guided selection before generation.

Guided enhancement assistants: making summaries faithful and operator-friendly. Practical summarization assistants often aim to reduce hallucinations and improve scanability by enforcing intermediate structure. logSage [99] (SAC'25) exemplifies this direction by treating summarization as an assistant workflow with guided enhancement: instead of asking an LLM to narrate raw logs end-to-end, the system emphasizes staged processing (e.g., evidence selection, grouping, and controlled rewriting) so that the final summary better reflects the underlying log evidence and aligns with on-call needs (handoff, incident notes, rapid situational awareness). Such designs typically trade some linguistic freedom for higher faithfulness and better operator trust.

Instruction-based log interpretation as summarization. Some works position summarization as part of a unified *instruction-following* interface for log analysis. LogLM [97] transforms multiple log tasks into instruction-based interactions; for summarization-like requests, it outputs natural-language interpretations and resolutions, and evaluates with standard generation metrics (BLEU/ROUGE variants). Similarly, Ji et al. [65] emphasize interpretable domain knowledge to adapt LLMs for log analysis; their "interpretation" outputs overlap with summarization in practice (explaining what the logs indicate), and are assessed via manual scoring, reflecting that human utility and correctness are hard to capture with automatic metrics alone. This line suggests that in real deployments, summarization is rarely an isolated task: it often co-occurs with Q&A, diagnosis hints, and "what should I do next" guidance.

Tooling for scalable log processing and its relationship to summarization. While not a summarization method per se, LogLead [109] illustrates an important enabling trend: scalable pipelines that load, enhance, and process logs efficiently can serve as the *front-end* to LLM summarization by curating what the model sees (e.g., enhancing logs, selecting windows, and organizing content). In other words, summarization quality is frequently bounded by upstream log preparation and selection rather than purely by the decoder's generation capability.

Across log summarization studies, the key contribution of LLMs is enabling *semantic compression*—turning noisy, heterogeneous log streams into coherent narratives and structured incident briefs. The most effective systems do not summarize "raw logs" directly; they first impose structure (grouping, salience selection, evidence extraction) and then generate a constrained summary, shifting the main challenge from generation to *faithful context curation and evaluation* under operator-centric criteria.

7 Conclusion

Recent advances in large language models have accelerated research on automated log analysis, driving rapid growth in LLM-based techniques over the last few years. This survey consolidates work on *LLM-based log analysis* across the operational pipeline and maps the literature into a task-driven view, summarizing how LLMs are applied to (i) improve upstream logging practices, (ii) parse semi-structured, drifting log streams into templates and parameters, and (iii) support downstream operational tasks such as anomaly detection, failure prediction, root cause analysis, and log summarization. Across these stages, a recurring pattern is that LLMs provide semantic generalization and enable evidence integration beyond logs alone, but also introduce practical constraints (context budgets, latency/cost, privacy/governance) and reliability risks (prompt sensitivity and ungrounded outputs). Future progress therefore depends not only on stronger models, but also on rigorous, reproducible evaluation and deployment-oriented designs that emphasize grounding, stability under drift and long-tail events, and verifiable, operator-aligned outputs.

References

[1] Siraaj Akhtar, Saad Khan, and Simon Parkinson. Llm-based event log analysis techniques: A survey. *arXiv preprint arXiv:2502.00677*, 2025.

[2] Crispin Almodovar, Fariza Sabrina, Sarvnaz Karimi, and Salahuddin Azad. Logfit: Log anomaly detection using fine-tuned language models. *IEEE Transactions on Network and Service Management*, 21(2):1715–1723, 2024.

[3] Apache Software Foundation. Apache log4j 2. Online documentation, 2024. URL https://logging.apache.org/log4j/2.x/.

[4] Merve Astekin, Max Hort, and Leon Moonen. A comparative study on large language models for log parsing. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2024)*, pages 36–47. ACM, 2024. doi: 10.1145/3674805.3686684. URL https://doi.org/10.1145/3674805.3686684.

[5] Prasasthy Balasubramanian, Dumindu Kankanamge, Ekaterina Gilman, and Mourad Oussalah. AnomalyExplainerBot: Explainable AI for LLM-based anomaly detection using BERTViz & Captum, 2025.

[6] Viktor Beck, Max Landauer, Markus Wurzenberger, Florian Skopik, and Andreas Rauber. System log parsing with large language models: A review. *arXiv preprint arXiv:2504.04877*, 2025.

[7] Sidahmed Benabderrahmane, Petko Valtchev, James Cheney, and Talal Rahwan. APT-LLM: Embedding-based anomaly detection of cyber advanced persistent threats using large language models. In *Proceedings of the 13th International Symposium on Digital Forensics and Security (ISDFS 2025)*, pages 1–6, 2025. doi: 10.1109/ISDFS65363.2025.11011912.

[8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. doi: 10.1109/72.279181.

[9] Jasmin Bogatinovski and Odej Kao. Auto-logging: Ai-centred logging instrumentation. In *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 95–100, 2023. doi: 10.1109/ICSE-NIER58687.2023.00023. URL https://doi.org/10.1109/ICSE-NIER58687.2023.00023.

[10] Bogdan Bogdan, Arina Cazacu, and Laura Vasilie. Good enough to learn: LLM-based anomaly detection in ECU logs without reliable labels, 2025. URL https://arxiv.org/abs/2507.01077. Accepted to IEEE Intelligent Vehicles Symposium (IV) 2025.

[11] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Jeff Dean, et al. On the opportunities and risks of foundation models, 2021. URL https://arxiv.org/abs/2108.07258.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

[13] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

[15] Song Chen and Hai Liao. Bert-log: Anomaly detection for system logs based on pre-trained language model. *Applied Artificial Intelligence*, 36(1):2145642, 2022. doi: 10.1080/08839514.2022.2145642. URL https://www.tandfonline.com/doi/full/10.1080/08839514.2022.2145642.

[16] Xiaolei Chen, Jie Shi, Jia Chen, Peng Wang, and Wei Wang. Epas: Efficient online log parsing via asynchronous scheduling of llm queries. In *Proceedings of the 41st IEEE International Conference on Data Engineering (ICDE 2025)*, pages 4025–4037. IEEE, 2025. doi: 10.1109/ICDE.2025.00318. URL https://ieeexplore.ieee.org/abstract/document/11113127.

[17] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 674–688, 2024.

[18] Paul F. Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://papers.neurips.cc/paper/7017-deep-reinforcement-learning-from-human-preferences.

[19] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[20] Clara Corbelle, Victor Carneiro, and Fidel Cacheda. Semantic hierarchical classification applied to anomaly detection using system logs with a bert model. *Applied Sciences*, 14(13), 2024. ISSN 2076-3417. doi: 10.3390/app14135388. URL https://www.mdpi.com/2076-3417/14/13/5388.

[21] Tianyu Cui, Ruowei Fu, Changchang Liu, Yuhe Ji, Wenwei Gu, Shenglin Zhang, Yongqian Sun, and Dan Pei. Aetherlog: Log-based root cause analysis by integrating large language models with knowledge graphs. In *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*, pages 49–60. IEEE, 2025.

[22] Tianyu Cui, Shiyu Ma, Ziang Chen, Tong Xiao, Chenyu Zhao, Shimin Tao, Yilun Liu, Shenglin Zhang, Duoming Lin, Changchang Liu, Yuzhe Cai, Weibin Meng, Yongqian Sun, and Dan Pei. Logeval: A comprehensive benchmark suite for llms in log analysis. *Empirical Softw. Engg.*, 30(6), October 2025. ISSN 1382-3256. doi: 10.1007/s10664-025-10701-6. URL https://doi.org/10.1007/s10664-025-10701-6.

[23] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. Logram: Efficient log parsing using *n* n-gram dictionaries. *IEEE Transactions on Software Engineering*, 48(3):879–892, 2020.

[24] Delgan and contributors. loguru: Python logging made simple. Online documentation, 2024. URL https://github.com/Delgan/loguru.

[25] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023. URL https://arxiv.org/abs/2305.14314.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *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, 2019.

[27] Shengcheng Duan, Yihua Xu, Sheng Zhang, Shen Wang, and Yue Duan. Pdlogger: Automated logging framework for practical software development, 2025. URL https://arxiv.org/abs/2507.19951.

[28] Yiqi Duan, Jianliang Xu, Changyu Fan, and Zixin Liu. Unsupervised log parsing based on large language models and entropy. In *2025 11th International Symposium on System Security, Safety, and Reliability (ISSSR)*, pages 1–10. IEEE, 2025.

[29] Chris Egersdoerfer, Di Zhang, and Dong Dai. Early exploration of using ChatGPT for log-based anomaly detection on parallel file systems logs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23)*, pages 315–316. Association for Computing Machinery, 2023. doi: 10.1145/3588195.3595943. URL https://doi.org/10.1145/3588195.3595943.

[30] Maryam Ekhlasi, Anurag Prakash, Maxime Lamothe, and Michel Dagenais. Insightai: Root cause analysis in large log files with private data using large language model. In *2025 IEEE/ACM 4th International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 31–41, 2025. doi: 10.1109/CAIN66642.2025.00012.

[31] Mahmoudreza Entezami, Shahabeddin Rahimi Harsini, David Houshangi, and Zahra Entezami. A novel framework for detecting anomalies in network security using LLM and deep learning. *Journal of Electrical Systems*, 21(1s):294–302, 2025. doi: 10.52783/jes.8791.

[32] Kawin Ethayarajh. How contextual are contextualized word representations? comparing the geometry of BERT, ELMo, and GPT-2 embeddings. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 55–65, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1006.

[33] European Parliament and Council of the European Union. Regulation (eu) 2016/679 of the european parliament and of the council (general data protection regulation). Official Journal of the European Union, L119, 2016. URL https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[34] Asma Fariha, Vida Gharavian, Masoud Makrehchi, Shahryar Rahnamayan, Sanaa Alwidian, and Akramul Azim. Log anomaly detection by leveraging llm-based parsing and embedding with attention mechanism. In *2024 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 859–863. IEEE, 2024.

[35] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139/.

[36] Joint Task Force. Security and privacy controls for information systems and organizations (nist special publication 800-53 revision 5), 2020. URL https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final.

[37] Amy Foster and Selva Kumar. *COMBINING LLMS AND SHELL LOGS TO PREDICT BACKUP FAILURES*. 07 2025.

[38] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33. ACM, 2014. doi: 10.1145/2591062.2591175.

[39] Ying Fu, Meng Yan, Pinjia He, Chao Liu, Xiaohong Zhang, and Dan Yang. End-to-end log statement generation at block-level. *Journal of Systems and Software*, 216:112146, 2024. doi: 10.1016/j.jss.2024.112146.

[40] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christopher Endres, Thorsten Holz, and Mario Fritz. Not what youve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection, 2023. URL https://arxiv.org/abs/2302.12173.

[41] Wei Guan, Jian Cao, Shiyou Qian, and Jianqi Gao. LogLLM: Log-based anomaly detection using large language models. *arXiv preprint*, 2024. URL https://arxiv.org/abs/2411.08561.

[42] H. Guo, S. Yuan, and X. Wu. Logbert: Log anomaly detection via bert. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[43] Hongcheng Guo, Jian Yang, Jiaheng Liu, Jiaqi Bai, Boyang Wang, Zhoujun Li, Tieqiao Zheng, Bo Zhang, Junran Peng, and Qi Tian. Logformer: A pre-train and tuning pipeline for log anomaly detection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 135–143, 2024.

[44] Hongcheng Guo, Jian Yang, Jiaheng Liu, Liqun Yang, Linzheng Chai, Jiaqi Bai, Junran Peng, Xiaorong Hu, Chao Chen, Dongfeng Zhang, Xu Shi, Tieqiao Zheng, Liangfan Zheng, Bo Zhang, Ke Xu, and Zhoujun Li. Owl: A large language model for it operations. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024. URL https://openreview.net/forum?id=SZOQ9RKYJu.

[45] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 8342–8360. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.acl-main.740. URL https://aclanthology.org/2020.acl-main.740/.

[46] Fatemeh Hadadi, Qinghua Xu, Domenico Bianculli, and Lionel Briand. Llm meets ml: Data-efficient anomaly detection on unstable logs. *ACM Transactions on Software Engineering and Methodology*, 2025. doi: 10.1145/3771283. URL https://doi.org/10.1145/3771283. arXiv:2406.07467.

[47] Minghua He, Tong Jia, Chiming Duan, Huaqian Cai, Ying Li, and Gang Huang. Llmelog: An approach for anomaly detection based on llm-enriched log events. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 132–143. IEEE, 2024.

[48] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017. doi: 10.1109/ICWS.2017.13.

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

[50] Shiming He, Ying Lei, Ying Zhang, Kun Xie, and Pradip Kumar Sharma. Parameter-efficient log anomaly detection based on pre-training model and lora. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–217. IEEE, 2023.

[51] Yi Wen Heng, Zeyang Ma, Zhenhao Li, Dong Jae Kim, and Tse-Hsun Chen. Benchmarking open-source large language models for log level suggestion. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 314–325, 2025. doi: 10.1109/ICST62969.2025.10988921.

[52] Jordis Emilia Herrmann, Aswath Mandakath Gopinath, Mikael Norrlof, and Mark Niklas Mueller. Diagnosing robotics systems issues with large language models–a case study. In *ICLR 2025 Workshop on Foundation Models in the Wild*.

[53] Sepp Hochreiter and J"urgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

[54] András Horváth, András Oláh, Attila Pintér, Bálint Siklósi, Gergely Lukács, István Z. Reguly, Kálmán Tornai, Tamás Zsedrovits, and Zoltán Máthé. Anomaly detection algorithms for real-time log data analysis at scale. *IEEE Access*, 13:136288–136311, 2025. doi: 10.1109/ACCESS.2025.3565575.

[55] Changze Hu, Yu Fang, Jinhua Wu, Haoyang Li, and Geng Wang. Research on log anomaly detection based on sentence-BERT. *Electronics*, 12(17):3580, 2023. doi: 10.3390/electronics12173580. URL https://www.mdpi.com/2079-9292/12/17/3580.

[56] 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. IEEE, 2024.

[57] Junjie Huang, Zhihan Jiang, Zhuangbin Chen, and Michael R. Lyu. No more labelled examples? an unsupervised log parser with llms. In *Proceedings of the ACM on Software Engineering (FSE 2025)*, number FSE. ACM, 2025. doi: 10.1145/3729377. URL https://dl.acm.org/doi/10.1145/3729377.

[58] Shaohan Huang, Yi Liu, Carol J. Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. Hitanomaly: Hierarchical transformers for anomaly detection in system log. *IEEE Transactions on Network and Service Management*, 17(2):2064–2076, 2020. doi: 10.1109/TNSM.2020.3034647.

[59] Shaohan Huang, Yi Liu, Carol J. Fung, He Wang, Hailong Yang, and Zhongzhi Luan. Improving log-based anomaly detection by pre-training hierarchical transformers. *IEEE Transactions on Computers*, 72(9):2656–2667, 2023. doi: 10.1109/TC.2023.3257518.

[60] Xin Huang, Ting Zhang, and Wen Zhao. LogRules: Enhancing log analysis capability of large language models through rules. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 452–470, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.28. URL https://aclanthology.org/2025.findings-naacl.28/.

[61] Xuanbo Huang, Kaiping Xue, Lutong Chen, Jiangping Han, Jian Li, and David S. L. Wei. Forensix: Automated network forensics and diagnostics for beyond-5g and 6g networks using large language models. *IEEE Network*, 39(5):74–80, 2025. doi: 10.1109/MNET.2025.3579925.

[62] Yintong Huo, Cheryl Lee, Yuxin Su, Shiwen Shan, Jinyang Liu, and Michael R. Lyu. Evlog: Evolving log analyzer for anomalous logs identification. arXiv preprint arXiv:2306.01509, 2023. URL https://arxiv.org/abs/2306.01509.

[63] Jaeyoon Jeong, Insung Baek, Byungwoo Bang, Junyeon Lee, Uiseok Song, and Seoung Bum Kim. Fall: Prior failure detection in large scale system based on language model. *IEEE Transactions on Dependable and Secure Computing*, 22(1):279–291, 2025. doi: 10.1109/TDSC.2024.3396166.

[64] Xin Ji, Le Zhang, Wenya Zhang, Fang Peng, Yifan Mao, Xingchuang Liao, and Kui Zhang. Lemad: LLM-empowered multi-agent system for anomaly detection in power grid services. *Electronics*, 14(15):3008, 2025. doi: 10.3390/electronics14153008. URL https://doi.org/10.3390/electronics14153008.

[65] Yuhe Ji, Yilun Liu, Feiyu Yao, Minggui He, Shimin Tao, Xiaofeng Zhao, Chang Su, Xinhua Yang, Weibin Meng, Yuming Xie, Boxing Chen, Shenglin Zhang, and Yongqian Sun. Adapting large language models to log analysis with interpretable domain knowledge. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, CIKM '25, page 1135–1144, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400720406. doi: 10.1145/3746252.3761189. URL https://doi.org/10.1145/3746252.3761189.

[66] Ziwei Ji, Nayeon Lee, Rita Frieske, Tianle Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation, 2023. URL https://arxiv.org/abs/2304.04710.

[67] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. Lilac: Log parsing using llms with adaptive parsing cache. *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024. doi: 10.1145/3643733. URL https://doi.org/10.1145/3643733.

[68] Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, Dawn Drain, Nelson Chen, Yuntao Bai, Jared Kaplan, Sam McCandlish, Dario Amodei, Ethan Chen, and Catherine Olsson. Language models (mostly) know what they know, 2022. URL https://arxiv.org/abs/2207.05221.

[69] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. arXiv preprint arXiv:2001.08361, 2020. URL https://arxiv.org/abs/2001.08361.

[70] Crystal Karlsen, Denis Copstein, Yue Luo, Benjamin Schwartzentruber, Tim Niblett, and Olivier Rouyer. Exploring semantic vs. syntactic features for unsupervised learning on application log files. In *2023 International Conference on Cyber Security and Networks (CSNet)*, 2023. doi: 10.1109/CSNet59123.2023.10339765.

[71] Egil Karlsen, Xiao Luo, Nur Zincir-Heywood, and Malcolm Heywood. Benchmarking large language models for log analysis, security, and interpretation. *Journal of Network and Systems Management*, 32:59, 2024. doi: 10.1007/s10922-024-09831-x.

[72] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-main.550. URL https://aclanthology.org/2020.emnlp-main.550/.

[73] Karen Kent and Murugiah Souppaya. Nist special publication 800-92: Guide to computer security log management, 2006. URL https://csrc.nist.gov/publications/detail/sp/800-92/final.

[74] 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)*. IEEE/ACM, 2021.

[75] Van-Hoang Le and Hongyu Zhang. Log parsing with prompt-based few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*, pages 2438–2449. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00204. URL https://conf.researchr.org/details/icse-2023/icse-2023-technical-track/165/Log-Parsing-with-Prompt-based-Few-shot-Learning.

[76] Van-Hoang Le and Hongyu Zhang. Log parsing: How far can chatgpt go? In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, page 1699–1704. IEEE Press, 2024. ISBN 9798350329964. doi: 10.1109/ASE56229.2023.00206. URL https://doi.org/10.1109/ASE56229.2023.00206.

[77] Van-Hoang Le and Hongyu Zhang. Prelog: A pre-trained model for log analytics. *Proceedings of the ACM on Management of Data*, 2(3): 1–28, 2024. doi: 10.1145/3654966. URL https://dl.acm.org/doi/10.1145/3654966. Presented at SIGMOD 2024.

[78] Van-Hoang Le, Yi Xiao, and Hongyu Zhang. Unleashing the true potential of semantic-based log parsing with pre-trained language models. In *Proceedings of the 47th International Conference on Software Engineering (ICSE 2025)*. IEEE/ACM, 2025. URL https://conf.researchr.org/details/icse-2025/icse-2025-research-track/80/Unleashing-the-True-Potential-of-Semantic-based-Log-Parsing-with-Pre-trained-Language. To appear.

[79] Juyoung Lee, Yeonsu Jeong, Taehyun Han, and Taejin Lee. Logresp-agent: A recursive ai framework for context-aware log anomaly detection and ttp analysis. *Applied Sciences*, 15(13), 2025. ISSN 2076-3417. doi: 10.3390/app15137237. URL https://www.mdpi.com/2076-3417/15/13/7237.

[80] Yukyung Lee, Jina Kim, and Pilsung Kang. Lanobert: System log anomaly detection based on bert masked language model, 2021. URL https://arxiv.org/abs/2111.09564.

[81] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 7871–7880, 2020.

[82] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich K"uttler, Mike Lewis, Wen-tau Yih, Tim Rockt"aschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2020. URL https://arxiv.org/abs/2005.11401.

[83] 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. IEEE, 2020.

[84] Xinyu Li, Yingtong Huo, Chenxi Mao, Shiwen Shan, Yuxin Su, Dan Li, and Zibin Zheng. AnomalyGen: An automated semantic log sequence generation framework with LLM for anomaly detection, 2025. URL https://arxiv.org/abs/2504.12250.

[85] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, Lionel C Briand, and Michael R Lyu. Exploring the effectiveness of llms in automated logging statement generation: An empirical study. *IEEE Transactions on Software Engineering*, 2024.

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

10.1145/3643754. URL https://doi.org/10.1145/3643754.

[87] Yichen Li, Jinyang Liu, Junsong Pu, Zhihan Jiang, Zhuangbin Chen, Xiao He, Tieying Zhang, Jianjun Chen, Yi Li, Rui Shi, and Michael Lyu. Automated proactive logging quality improvement for large-scale codebases. In *Proceedings of the 2025 IEEE/ACM International Conference on Automated Software Engineering (ASE 2025) (Industry Showcase)*, 2025. URL https://conf.researchr.org/details/ase-2025/ase-2025-industry-showcase/28/Automated-Proactive-Logging-Quality-Improvement-for-Large-Scale-Codebases.

[88] Yichen Li, Yulun Wu, Jinyang Liu, Zhihan Jiang, Zhuangbin Chen, Guangba Yu, and Michael R Lyu. Coca: Generative root cause analysis for distributed systems with code knowledge. *arXiv preprint arXiv:2503.23051*, 2025.

[89] Zhenhao Li, Heng Li, Tse-Hsun Chen, and Weiyi Shang. Deeplv: Suggesting log levels using ordinal based neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1461–1472. IEEE, 2021.

[90] Zhenhao Li, An Ran Chen, Xing Hu, Xin Xia, Tse-Hsun Chen, and Weiyi Shang. Are they all good? studying practitioners' expectations on the readability of log messages. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*, 2023. doi: 10.48550/arXiv.2308.08836. URL https://arxiv.org/abs/2308.08836.

[91] Yifan Liao, Ming Xu, Yun Lin, Xiwen Teoh, Xiaofei Xie, Ruitao Feng, Frank Liaw, Hongyu Zhang, and Jin Song Dong. Detecting and explaining anomalies caused by web tamper attacks via building consistency-based normality. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 531–543, 2024.

[92] Ying Fu Lim, Jiawen Zhu, and Guansong Pang. Adapting large language models for parameter-efficient log anomaly detection. arXiv preprint, 2025. URL https://arxiv.org/abs/2503.08045. arXiv:2503.08045.

[93] Yifei Lin, H. Deng, and X. Li. Fastlogad: Log anomaly detection with mask-guided pseudo anomaly generation and discrimination, 2024. URL https://arxiv.org/abs/2404.08750.

[94] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, 2021. URL https://arxiv.org/abs/2107.13586.

[95] Yilun Liu, Shimin Tao, Weibin Meng, Jingyu Wang, Wenbing Ma, Yuhang Chen, Yanqing Zhao, Hao Yang, and Yanfei Jiang. Interpretable online log analysis using large language models with prompt strategies. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24)*, pages 35–46. Association for Computing Machinery, 2024. doi: 10.1145/3643916.3644408. URL https://doi.org/10.1145/3643916.3644408.

[96] Yilun Liu, Shimin Tao, Weibin Meng, Feiyu Yao, Xingyu Zhao, and Hao Yang. LogPrompt: Prompt engineering towards zero-shot and interpretable log analysis. In *Companion Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE Companion '24)*. Association for Computing Machinery, 2024. doi: 10.1145/3639478.3643108. URL https://doi.org/10.1145/3639478.3643108.

[97] Yilun Liu, Yuhe Ji, Shimin Tao, Minggui He, Weibin Meng, Shenglin Zhang, Yongqian Sun, Yuming Xie, Boxing Chen, and Hao Yang. Loglm: From task-based to instruction-based automated log analysis. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 401–412. IEEE, 2025.

[98] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[99] Yang Lu. *LogSage: Log Summarization Assistant with Guided Enhancement*, page 1979–1981. Association for Computing Machinery, New York, NY, USA, 2025. ISBN 9798400706295. URL https://doi.org/10.1145/3672608.3707990.

[100] Wengang Lyu, Yilun Wang, Wei Wei, Runkun Xiao, Youwei Li, Gen Li, Laiyuan Yang, and Zibin Zheng. Exploring variable potential for llm-based log parsing efficiency and reduced costs. In *Proceedings of the FSE 2025 Companion*, 2025. doi: 10.1145/3696630.3728506. URL https://doi.org/10.1145/3696630.3728506.

[101] Lipeng Ma, Weidong Yang, Sihang Jiang, Ben Fei, Mingjie Zhou, Shuhao Li, Mingyu Zhao, Bo Xu, and Yanghua Xiao. LUK: Empowering log understanding with expert knowledge from large language models, 2024.

[102] Lipeng Ma, Weidong Yang, Bo Xu, Sihang Jiang, Ben Fei, Jiaqing Liang, Mingjie Zhou, and Yanghua Xiao. Knowlog: Knowledge enhanced pre-trained language model for log understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3623304. URL https://doi.org/10.1145/3597503.3623304.

[103] Lipeng Ma, Yixuan Li, Weidong Yang, Mingjie Zhou, Xinyi Liu, Ben Fei, Shuhao Li, Xiaoyan Sun, Sihang Jiang, and Yanghua Xiao. LogReasoner: Empowering LLMs with expert-like coarse-to-fine reasoning for log analysis tasks, 2025.

[104] Lipeng Ma, Weidong Yang, Yixuan Li, Ben Fei, Mingjie Zhou, Shuhao Li, Sihang Jiang, Bo Xu, and Yanghua Xiao. Adaptivelog: An adaptive log analysis framework with the collaboration of large and small language model. *ACM Transactions on Software Engineering and Methodology*, 2025. doi: 10.1145/3749840. URL https://doi.org/10.1145/3749840. arXiv:2501.11031.

[105] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. Llmparser: An exploratory study on using large language models for log parsing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE 2024)*, pages 99:1–99:13, 2024. doi: 10.1145/3597503.3639150. URL https://doi.org/10.1145/3597503.3639150.

[106] Zeyang Ma, Dong Jae Kim, and Tse-Hsun Chen. Librelog: Accurate and efficient unsupervised log parsing using open-source large language models. In *Proceedings of the 47th International Conference on Software Engineering (ICSE 2025)*. IEEE/ACM, 2025.

URL https://conf.researchr.org/details/icse-2025/icse-2025-research-track/75/LibreLog-Accurate-and-Efficient-Unsupervised-Log-Parsing-Using-Open-Source-Large-Lan. To appear.

[107] Potsawee Manakul, Adian Liusie, and Mark J. F. Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models, 2023. URL https://arxiv.org/abs/2303.08896.

[108] Praveen Kumar Mannam. Optimizing software release management with gpt-enabled log anomaly detection. In *2023 26th International Conference on Computer and Information Technology (ICCIT)*, pages 1–6. IEEE, 2023.

[109] Mika V Mäntylä, Yuqing Wang, and Jesse Nyyssölä. Loglead-fast and integrated log loader, enhancer, and anomaly detector. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 395–399. IEEE, 2024.

[110] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*, pages 2279–2290, 2022. doi: 10.1145/3510003.3511561. URL https://doi.org/10.1145/3510003.3511561.

[111] Antonio Mastropaolo, Valentina Ferrari, Luca Pascarella, and Gabriele Bavota. Log statements generation via deep learning: Widening the support provided to developers. *Journal of Systems and Software*, 210:111947, 2024. doi: 10.1016/j.jss.2023.111947. URL https://doi.org/10.1016/j.jss.2023.111947.

[112] Maryam Mehrabi, Abdelwahab Hamou-Lhadj, and Hossein Moosavi. The effectiveness of compact fine-tuned llms in log parsing. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 438–448. IEEE, 2024.

[113] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christos Nalmpantis, Ram Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Damien Sileo, Lewis Tunstall, et al. Augmented language models: a survey, 2023. URL https://arxiv.org/abs/2302.07842.

[114] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[115] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. Rethinking the role of demonstrations: What makes in-context learning work?, 2022. URL https://arxiv.org/abs/2202.12837.

[116] Priyanka Mudgal, Bijan Arbab, and Swaathi Sampath Kumar. Crasheventllm: Predicting system crashes with large language models. In *2024 International Conference on Information Technology and Computing (ICITCOM)*, pages 72–76. IEEE, 2024.

[117] Gunho No, Yukyung Lee, Hyeongwon Kang, and Pilsung Kang. RAPID: Training-free retrieval-based log anomaly detection with pre-trained language model considering token-level information. *Engineering Applications of Artificial Intelligence*, 133:108613, 2024. doi: 10.1016/j.engappai.2024.108613.

[118] OpenAI. Gpt-4 technical report, 2023. URL https://arxiv.org/abs/2303.08774.

[119] Oracle. Java platform, standard edition `java.util.logging`. Online documentation, 2024. URL https://docs.oracle.com/en/java/javase/.

[120] Harold Ott, Jasmin Bogatinovski, Alexander Acker, Sasho Nedelkoski, and Odej Kao. Robust and transferable anomaly detection in log data using pre-trained language models, 2021. URL https://arxiv.org/abs/2102.11570.

[121] Youssef Esseddiq Ouatiti, Mohammed Sayagh, Bram Adams, and Ahmed E. Hassan. Omnillp: Enhancing llm-based log level prediction with context-aware retrieval, 2025. URL https://arxiv.org/abs/2508.08545.

[122] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. URL https://arxiv.org/abs/2203.02155.

[123] Jonathan Pan, Swee Liang Wong, and Yidi Yuan. Raglog: Log anomaly detection using retrieval augmented generation, 2023. URL https://arxiv.org/abs/2311.05261.

[124] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[125] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pages 1310–1318, 2013.

[126] Changhua Pei, Zihan Liu, Jianhui Li, Erhan Zhang, Le Zhang, Haiming Zhang, Wei Chen, Dan Pei, and Gaogang Xie. Self-evolutionary group-wise log parsing based on large language model. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 49–60. IEEE, 2024.

[127] Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, et al. Flow-of-action: Sop enhanced llm-based multi-agent system for root cause analysis. In *Companion Proceedings of the ACM on Web Conference 2025*, pages 422–431, 2025.

[128] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. doi: 10.18653/v1/N18-1202.

[129] Reiner Pope, Yash Jain, Rohan Peri, Jeff Hankins, Baolin Lin, and Ion Stoica. Efficiently scaling transformer inference with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

[130] Przemek Pospieszny, Wojciech Mormul, Karolina Szyndler, and Sanjeev Kumar. ADALog: Adaptive unsupervised anomaly detection in logs with self-attention masked language model. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2505.13496.

[131] Python Software Foundation. Python `logging` — logging facility for python. Online documentation, 2024. URL https://docs.python.org/3/library/logging.html.

[132] Jiaxing Qi, Shaohan Huang, Zhongzhi Luan, Shu Yang, Carol J. Fung, Hailong Yang, Depei Qian, Jing Shang, Zhiwen Xiao, and Zhihui Wu. LogGPT: Exploring ChatGPT for log-based anomaly detection. In *2023 IEEE International Conference on High Performance Computing & Communications, Data Science & Systems, Smart City & Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 273–280, 2023. doi: 10.1109/HPCC-DSS-SMARTCITY-DEPENDSYS60770.2023.00045. URL https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys60770.2023.00045.

[133] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences*, 63(10):1872–1897, 2020. doi: 10.1007/s11431-020-1647-3. URL https://arxiv.org/abs/2003.08271.

[134] QOS.ch. Logback. Online documentation, 2024. URL https://logback.qos.ch/.

[135] QOS.ch. Slf4j: Simple logging facade for java. Online documentation, 2024. URL https://www.slf4j.org/.

[136] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2023. URL https://arxiv.org/abs/2305.18290.

[137] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[138] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[139] Hengyi Ren, Kun Lan, Zhi Sun, and Shan Liao. Clogllm: A large language model enabled approach to cybersecurity log anomaly analysis. In *2024 4th International Conference on Electronic Information Engineering and Computer Communication (EIECC)*, pages 963–970. IEEE, 2024.

[140] Mayra Sofia Ruiz Rodriguez, SayedHassan Khatoonabadi, and Emad Shihab. Automated file-level logging generation for machine learning applications using llms: A case study using gpt-4o mini, 2025. URL https://arxiv.org/abs/2508.04820.

[141] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

[142] Hynek Schlawack and contributors. structlog: Structured logging for python. Online documentation, 2024. URL https://www.structlog.org/.

[143] Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, volume 1, pages 1073–1083, 2017. doi: 10.18653/v1/P17-1099.

[144] Pratheek Senevirathne, Samindu Cooray, Jerome Dinal Herath, and Dinuni Fernando. Virtual machine proactive fault tolerance using log-based anomaly detection. *IEEE Access*, 2024.

[145] Febrian Setianto, Erion Tsani, Fatima Sadiq, Georgios Domalis, Dimitris Tsakalidis, and Panos Kostakos. Gpt-2c: A parser for honeypot logs using large pre-trained language models. In *Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2021)*, pages 649–653. ACM, 2021. doi: 10.1145/3487351.3492723. URL https://dl.acm.org/doi/10.1145/3487351.3492723.

[146] 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*, pages 13–25, 2024.

[147] Shiwen Shan, Yintong Huo, Yuxin Su, Zhining Wang, Dan Li, and Zibin Zheng. Conflogger: Enhance systems' configuration diagnosability through configuration logging. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*. ACM, 2026. doi: 10.1145/3744916.3764570. URL https://arxiv.org/abs/2508.20977.

[148] Yangyi Shao, Wenbin Zhang, Peishun Liu, Ren Huyue, Ruichun Tang, Qilin Yin, and Qi Li. Log anomaly detection method based on bert model optimization. In *2022 7th International conference on cloud computing and big data analytics (ICCCBDA)*, pages 161–166. IEEE, 2022.

[149] Noah Shinn, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

[150] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019. URL https://arxiv.org/abs/1909.08053.

[151] Honglin Shu, Dong Wang, Antonio Mastropaolo, Gabriele Bavota, and Yasutaka Kamei. An empirical study on language models for generating log statements in test code. *ACM Transactions on Software Engineering and Methodology*, 2025. doi: 10.1145/3759915.

[152] Chengyu Song, Linru Ma, Jianming Zheng, Jinzhi Liao, Hongyu Kuang, and Lin Yang. Audit-LLM: Multi-agent collaboration for log-based insider threat detection. *arXiv preprint*, 2024. URL https://arxiv.org/abs/2408.08902.

[153] Chengyu Song, Lin Yang, Jianming Zheng, and Fei Cai. Parse-llm: A prior-free llm parser for unknown system logs. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management (CIKM 2025)*. ACM, 2025. doi: 10.1145/3746252.3761363. URL https://dl.acm.org/doi/10.1145/3746252.3761363.

[154] Shuang Song, Yifei Zhang, and Neng Gao. Confront insider threat: Precise anomaly detection in behavior logs based on LLM fine-tuning. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert, editors, *Proceedings of the 31st International Conference on Computational Linguistics*, pages 8589–8601, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.574/.

[155] Abhay Srivatsa and Venkatasai Gudisa. Logsense: Scalable real-time log anomaly detection architecture.

[156] Yicheng Sui, Xiaotian Wang, Tianyu Cui, Tong Xiao, Chenghao He, Shenglin Zhang, Yuzhi Zhang, Xiao Yang, Yongqian Sun, and Dan Pei. Bridging the gap: LLM-powered transfer learning for log anomaly detection in new software systems. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*, pages 4414–4427. IEEE, 2025. doi: 10.1109/ICDE65448.2025.00331. URL https://doi.org/10.1109/ICDE65448.2025.00331.

[157] Danyu Sun, Jinghuai Zhang, Jiacen Xu, Yu Zheng, Yuan Tian, and Zhou Li. From alerts to intelligence: A novel LLM-aided framework for host-based intrusion detection, 2025.

[158] Yicheng Sun, Jacky Wai Keung, Zhen Yang, Shuo Liu, and Yihan Liao. SemiSMAC: A semi-supervised framework for log anomaly detection with automated hyperparameter tuning. *Information and Software Technology*, 187:107869, 2025. doi: 10.1016/j.infsof.2025.107869.

[159] Yicheng Sun, Jacky Wai Keung, Zhen Yang, Shuo Liu, and Hi Kuen Yu. Semirald: A semi-supervised hybrid language model for robust anomalous log detection. *Information and Software Technology*, 183:107743, July 2025. doi: 10.1016/j.infsof.2025.107743. URL https://doi.org/10.1016/j.infsof.2025.107743.

[160] Yicheng Sun, Jacky Keung, Zhen Yang, Shuo Liu, and Hi Kuen Yu. Improving anomaly detection in software logs through hybrid language modeling and reduced reliance on parser. *Automated Software Engineering*, 33(1):12, 2026.

[161] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 27, 2014.

[162] Sayed Taheri, Achintha Ihalage, Prateek Mishra, Sean Coaker, Faris Muhammad, and Hamed Al-Raweshidy. Domain tailored large language models for log mask prediction in cellular network diagnostics. *IEEE Transactions on Network and Service Management*, 22(3): 2370–2381, 2025. doi: 10.1109/TNSM.2025.3541384.

[163] Boyin Tan, Junjielong Xu, Zhouruixing Zhu, and Pinjia He. Al-bench: A benchmark for automatic logging, 2025. URL https://arxiv.org/abs/2502.03160.

[164] Pan Tang, Shixiang Tang, Huanqi Pu, Zhiqing Miao, and Zhixing Wang. Microrca-agent: Microservice root cause analysis method based on large language model agents. *arXiv preprint arXiv:2509.15635*, 2025.

[165] Shimin Tao, Weibin Meng, Yimeng Chen, Yichen Zhu, Ying Liu, Chunning Du, Tao Han, Yongpeng Zhao, Xiangguang Wang, and Hao Yang. Logstamp: Automatic online log parsing based on sequence labelling. *ACM SIGMETRICS Performance Evaluation Review*, 49(4): 93–98, 2022. doi: 10.1145/3543146.3543168. URL https://dl.acm.org/doi/10.1145/3543146.3543168.

[166] Shimin Tao, Yilun Liu, Weibin Meng, Zuomin Ren, Hao Yang, Xun Chen, Liang Zhang, Yuming Xie, Chang Su, Xiaosong Oiao, Weinan Tian, Yichen Zhu, Tao Han, Ying Qin, and Yun Li. 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, 2023. doi: 10.1109/IWQoS57198.2023.10188759.

[167] Subin Thali, R. Chethan, G. Supreeth, M. Sangamesh, and S. Swami. LLM-based detection of cyber anomalies in industrial control systems. *Zhuzao/Foundry*, 28(6):237–241, 2025. doi: 10.29014/FJ-2025-1001-4977.2073.

[168] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[169] Mohamed Trabelsi, Zenghui Yuan, Jiuyong Han, and Fatima Zohra Ros. LogGPT: Log anomaly detection via GPT. *arXiv preprint*, 2023. URL https://arxiv.org/abs/2309.14482.

[170] Risto Vaarandi and Hayretdin Bahsi. Using large language models for template detection from security event logs. *International Journal of Information Security*, 24, 2025. doi: 10.1007/s10207-025-01018-y. URL https://doi.org/10.1007/s10207-025-01018-y.

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

[172] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning (ICML)*, pages 1096–1103, 2008. doi: 10.1145/1390156.1390294.

[173] Jiabo Wang, Guojun Chu, Jingyu Wang, Haifeng Sun, Qi Qi, Yuanyi Wang, Ji Qi, and Jianxin Liao. Logexpert: Log-based recommended resolutions generation using large language model. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, page 42–46, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705007. doi: 10.1145/3639476.3639773. URL https://doi.org/10.1145/3639476.3639773.

[174] Kehan Wang, Siqin Zhang, Haijing Nan, Xueyu Hou, Jiaqi Zou, and Zicong Miao. *Empirical Analysis of LLMDPP: Advancing Log Parsing in the LLM Era*, page 753–758. Association for Computing Machinery, New York, NY, USA, 2025. ISBN 9798400714535. URL https://doi.org/10.1145/3711875.3736683.

[175] Xin Wang, Zhenhao Li, and Zishuo Ding. Defects4log: Benchmarking llms for logging code defect detection and reasoning, 2025. URL https://arxiv.org/abs/2508.11305.

[176] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2022. URL https://arxiv.org/abs/2203.11171.

[177] Yilun Wang, Pengfei Chen, Haiyu Huang, Zilong He, Gou Tan, Chuanfu Zhang, Jingkai He, and Zibin Zheng. Inferlog: Accelerating llm inference for online log parsing via icl-oriented prefix caching. In *Proceedings of the 48th International Conference on Software Engineering (ICSE 2026)*, 2026. URL https://arxiv.org/abs/2507.08523. Accepted.

[178] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8696–8708. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685/.

[179] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pages 4966–4974, 2024.

[180] Jason Wei, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Jeff Dean, Yuhuai Li, Adam Roberts, Anna Rumshisky, Noam Shazeer, et al. Emergent abilities of large language models. arXiv preprint arXiv:2206.07682, 2022. URL https://arxiv.org/abs/2206.07682.

[181] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2022. URL https://arxiv.org/abs/2201.11903.

[182] Yifan Wu, Siyu Yu, and Ying Li. Log parsing using llms with self-generated in-context learning and self-correction. In *Proceedings of the 33rd IEEE/ACM International Conference on Program Comprehension (ICPC 2025)*. IEEE/ACM, 2025. URL https://arxiv.org/abs/2406.03376. Method name: AdaParser.

[183] Pei Xiao, Tong Jia, Chiming Duan, Minghua He, Weijie Hong, Xixuan Yang, Yihan Wu, Ying Li, and Gang Huang. Clslog: Collaborating large and small models for log-based anomaly detection. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25)*, pages 686–690, Trondheim, Norway, June 2025. ACM. doi: 10.1145/3696630.3728524. URL https://doi.org/10.1145/3696630.3728524.

[184] Yi Xiao, Van-Hoang Le, and Hongyu Zhang. Demonstration-free: Towards more practical log parsing with large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024)*, pages 1243–1255. ACM, 2024. doi: 10.1145/3691620.3694994. URL https://dl.acm.org/doi/10.1145/3691620.3694994. Method name: LogBatcher.

[185] Xiaoyuan Xie, Zhipeng Cai, Songqiang Chen, and Jifeng Xuan. Fastlog: An end-to-end method to efficiently generate and insert logging statements. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*, pages 26–37. ACM, 2024. doi: 10.1145/3650212.3652107. URL https://arxiv.org/abs/2311.02862.

[186] Andy Xu and Arno Gau. Help: Hierarchical embeddings-based log parsing. arXiv preprint arXiv:2408.08300, 2024. URL https://arxiv.org/abs/2408.08300.

[187] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Unilog: Automatic logging via LLM and in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE '24)*. ACM, 2024. doi: 10.1145/3597503.3623326.

[188] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. Divlog: Log parsing with prompt enhanced in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE '24)*, pages 199:1–199:12, 2024. doi: 10.1145/3597503.3639155. URL https://doi.org/10.1145/3597503.3639155.

[189] Junjielong Xu, Qinan Zhang, Zhiqing Zhong, Shilin He, Chaoyun Zhang, Qingwei Lin, Dan Pei, Pinjia He, Dongmei Zhang, and Qi Zhang. Openrca: Can large language models locate the root cause of software failures? In *The Thirteenth International Conference on Learning Representations*, 2025.

[190] Weiyuan Xu, Juntao Luo, Tao Huang, Kaixin Sui, Jie Geng, Qijun Ma, Isami Akasaka, Xiaoxue Shi, Jing Tang, and Peng Cai. A two-staged llm-based framework for ci/cd failure detection and remediation with industrial validation. *arXiv preprint arXiv:2506.03691*, 2025.

[191] Yuuki Yamanaka, Tomokatsu Takahashi, Takuya Minami, and Yoshiaki Nakajima. LogELECTRA: Self-supervised anomaly detection for unstructured logs. *arXiv preprint*, 2024. URL https://arxiv.org/abs/2402.10397.

[192] Tiankai Yang, Yi Nian, Li Li, Ruiyao Xu, Yuangang Li, Jiaqi Li, Zhuo Xiao, Xiyang Hu, Ryan A. Rossi, Kaize Ding, Xia Hu, and Yue Zhao. AD-LLM: Benchmarking large language models for anomaly detection. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 1524–1547, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.79. URL https://aclanthology.org/2025.findings-acl.79/.

[193] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2022. URL https://arxiv.org/abs/2210.03629.

[194] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL https://arxiv.org/abs/2305.10601.

[195] Xian Yu, Shengxi Nong, Dongbiao He, Weijie Zheng, Teng Ma, Ning Liu, Jianhui Li, and Gaogang Xie. Loggenius: An unsupervised log parsing framework with zero-shot prompt engineering. In *2024 IEEE International Conference on Web Services (ICWS)*, pages 1321–1328, 2024. doi: 10.1109/ICWS62655.2024.00159. URL https://doi.org/10.1109/ICWS62655.2024.00159.

[196] Chenbo Zhang, Wenying Xu, Jinbu Liu, Lu Zhang, Guiyang Liu, Jihong Guan, Qi Zhou, and Shuigeng Zhou. Logbase: A large-scale benchmark for semantic log parsing. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2025)*. ACM, 2025. doi: 10.1145/3728969. URL https://dl.acm.org/doi/10.1145/3728969.

[197] Chenbo Zhang, Wenying Xu, Jinbu Liu, Lu Zhang, Guiyang Liu, Jihong Guan, Qi Zhou, and Shuigeng Zhou. Semanticlog: Towards effective and efficient large-scale semantic log parsing. *IEEE Transactions on Software Engineering*, 2025. doi: 10.1109/TSE.2025.3524891. URL https://ieeexplore.ieee.org/abstract/document/11216353.

[198] Hao Zhang, Dongjun Yu, Lei Zhang, Guoping Rong, Yongda Yu, Haifeng Shen, He Zhang, Dong Shao, and Hongyu Kuang. Aucad: Automated construction of alignment dataset from log-related issues for enhancing llm-based log generation. In *Proceedings of the 16th International Conference on Internetware*, Internetware '25, page 413–425, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400719264. doi: 10.1145/3755881.3755889. URL https://doi.org/10.1145/3755881.3755889.

[199] Lingzhe Zhang, Tong Jia, Mengxi Jia, Yifan Wu, Aiwei Liu, Yong Yang, Zhonghai Wu, Xuming Hu, Philip Yu, and Ying Li. A survey of aiops in the era of large language models. *ACM Computing Surveys*, 2025.

[200] Lingzhe Zhang, Tong Jia, Mengxi Jia, Yifan Wu, Hongyi Liu, and Ying Li. Scalalog: Scalable log-based failure diagnosis using llm. In *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5, 2025. doi: 10.1109/ICASSP49660.2025.10888670.

[201] Lingzhe Zhang, Tong Jia, Mengxi Jia, Yifan Wu, Hongyi Liu, and Ying Li. Xraglog: A resource-efficient and context-aware log-based anomaly detection method using retrieval-augmented generation. In *AAAI 2025 Workshop on Preventing and Detecting LLM Misinformation (PDLM)*, 2025.

[202] Lingzhe Zhang, Tong Jia, Kangjin Wang, Weijie Hong, Chiming Duan, Minghua He, and Ying Li. Adaptive root cause localization for microservice systems with multi-agent recursion-of-thought. *arXiv preprint arXiv:2508.20370*, 2025.

[203] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.

[204] Ting Zhang, Xin Huang, Wen Zhao, Shaohuang Bian, and Peng Du. Logprompt: A log-based anomaly detection framework using prompts. In *2023 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2023. doi: 10.1109/IJCNN54540.2023.10191948.

[205] Wanhao Zhang, Qianli Zhang, Enyu Yu, Yuxiang Ren, Yeqing Meng, Mingxi Qiu, and Jilong Wang. Leveraging rag-enhanced large language model for semi-supervised log anomaly detection. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 168–179, 2024. doi: 10.1109/ISSRE62328.2024.00026.

[206] Wei Zhang, Xiangyuan Guan, Lu Yunhong, Jie Zhang, Shuangyong Song, Xianfu Cheng, Zhenhe Wu, and Zhoujun Li. Lemur: Log parsing with entropy sampling and chain-of-thought merging, 2025. URL https://arxiv.org/abs/2402.18205.

[207] Xiao Zhang, Qi Wang, Mingyi Li, Yuan Yuan, Mengbai Xiao, Fuzhen Zhuang, and Dongxiao Yu. Tamo: Fine-grained root cause analysis via tool-assisted llm agent with multi-modality observation data in cloud-native systems. *IEEE Transactions on Services Computing*, 18 (6):4221–4233, 2025.

[208] Xuchao Zhang, Supriyo Ghosh, Chetan Bansal, Rujia Wang, Minghua Ma, Yu Kang, and Saravan Rajmohan. Automated root causing of cloud incidents using in-context learning with gpt-4. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 266–277, 2024.

[209] Zhaohui Zhang, Shu Chen, and Dandan Shi. LogST: Log semi-supervised anomaly detection based on sentence-BERT. In *2022 6th International Conference on System Innovation (ICSIP)*, pages 356–361, 2022. doi: 10.1109/ICSIP55141.2022.9886069.

[210] Haoyu Zheng, Guojun Chu, Haifeng Sun, Jingyu Wang, Shimin Tao, and Hao Yang. LogDAPT: Log data anomaly detection with domain-adaptive pretraining (industry track). In *Proceedings of the 24th International Middleware Conference: Industrial Track (Middleware '23 Industry)*, pages 15–21. Association for Computing Machinery, 2023. doi: 10.1145/3626562.3626830. URL https://doi.org/10.1145/3626562.3626830.

[211] Chen Zhi, Liye Cheng, Meilin Liu, Xinkui Zhao, Yueshen Xu, and Shuiguang Deng. Llm-powered zero-shot online log parsing. In *2024 IEEE International Conference on Web Services (ICWS)*, pages 877–887. IEEE, 2024.

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

[213] Renyi Zhong, Yichen Li, Jinxi Kuang, Wenwei Gu, Yintong Huo, and Michael R. Lyu. Logupdater: Automated detection and repair of specific defects in logging statements. *ACM Transactions on Software Engineering and Methodology*, 2025. doi: 10.1145/3731754.

[214] Renyi Zhong, Yichen Li, Guangba Yu, Wenwei Gu, Jinxi Kuang, Yintong Huo, and Michael R. Lyu. Larger is not always better: Exploring small open-source language models in logging statement generation. *ACM Trans. Softw. Eng. Methodol.*, October 2025. ISSN 1049-331X.

doi: 10.1145/3773287. URL https://doi.org/10.1145/3773287. Just Accepted.

[215] Denny Zhou, Nathanael Sch"arli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models, 2022. URL https://arxiv.org/abs/2205.10625.

[216] Yihan Zhou, Yan Chen, Xuanming Rao, Yukang Zhou, Yuxin Li, and Chao Hu. Leveraging large language models and bert for log parsing and anomaly detection. *Mathematics*, 12(17), 2024. ISSN 2227-7390. doi: 10.3390/math12172758. URL https://www.mdpi.com/2227-7390/12/17/2758.

[217] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*. IEEE/ACM, 2015. doi: 10.1109/ICSE.2015.60.

[218] Xuhang Zhu, Xiu Tang, Sai Wu, Gang Chen, Haobo Wang, Chang Yao, Quanqing Xu, and Jichen Li. CoLA: Model collaboration for log-based anomaly detection. *Proceedings of the VLDB Endowment*, 18(11):3979–3987, 2025. doi: 10.14778/3749646.3749668.

[219] Xinkai Zou, Xuan Jiang, Ruikai Huang, Haoze He, Parv Kapoor, Hongrui Wu, Yibo Wang, Jian Sha, Xiongbo Shi, Zixun Huang, and Jinhua Zhao. Towards generalizable context-aware anomaly detection: A large-scale benchmark in cloud environments, 2025. URL https://arxiv.org/abs/2508.01844.