AlertGuardian: Intelligent Alert Life-Cycle Management for Large-scale Cloud Systems

Guangba Yu[†], Genting Mai[†], Rui Wang[‡], Ruipeng Li[‡], Pengfei Chen[†*], Long Pan[‡], Ruijie Xu[‡]
[†] Sun Yat-sen University, Guangzhou, China [‡] Tencent, Shenzhen, China

***Abstract*—Alerts are critical for detecting anomalies in large-scale cloud systems, ensuring reliability and user experience. However, current systems generate overwhelming volumes of alerts, degrading operational efficiency due to ineffective alert life-cycle management. This paper details the efforts of *Company-X* to optimize alert life-cycle management, addressing alert fatigue in cloud systems. We propose AlertGuardian, a framework collaborating large language models (LLMs) and lightweight graph models to optimize the alert life-cycle through three phases: *Alert Denoise* uses graph learning model with virtual noise to filter noise, *Alert Summary* employs Retrieval Augmented Generation (RAG) with LLMs to create actionable summary, and *Alert Rule Refinement* leverages multi-agent iterative feedbacks to improve alert rule quality. Evaluated on four real-world datasets from *Company-X*'s services, AlertGuardian significantly mitigates alert fatigue (94.8% alert reduction ratios) and accelerates fault diagnosis (90.5% diagnosis accuracy). Moreover, AlertGuardian improves 1,174 alert rules, with 375 accepted by SREs (32% acceptance rate). Finally, we share success stories and lessons learned about alert life-cycle management after the deployment of AlertGuardian in *Company-X*.**

***Index Terms*—Alert Life-Cycle, Alert Reduction, Cloud Systems**

I. Introduction

Cloud systems, such as Microsoft Azure, Amazon AWS, and Tencent Cloud, provide critical services to users worldwide. Ensuring their reliability is paramount to prevent user dissatisfaction and revenue loss. However, failures, such as unexpected interruptions or service level degradation, are inevitable in complex cloud environments [1]–[4]. To detect and address these failures promptly, modern cloud systems employ comprehensive monitoring mechanisms that continuously track the health of services, generating diverse monitoring data (*e.g.*, key performance indicators (KPIs)) [5]–[8]. Site Reliability Engineers (SREs) leverage their expertise and historical monitoring data to configure alert rules [9]–[12]. When monitoring data meets these rule conditions, alerts are triggered to notify SREs for inspection and mitigation.

Although alert systems have evolved significantly from their humble beginnings, they continue to present substantial challenges that hinder operational effectiveness across organizations [13]–[18]. As cloud systems grow more complex and interconnected, the sheer volume of alerts generated by monitoring systems has reached unprecedented levels. Modern monitoring infrastructures generate thousands of alerts daily, creating an overwhelming deluge of notifications that often obscure critical issues rather than illuminating them. This

*Corresponding author (chenpf7@mail.sysu.edu.cn).

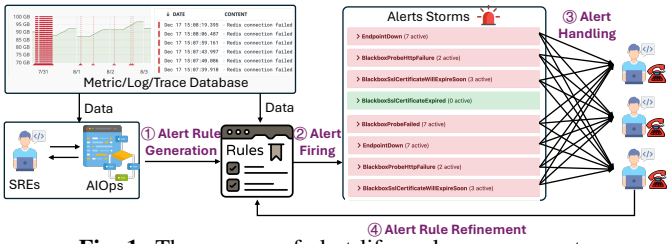

Fig. 1: The process of alert life-cycle management.

phenomenon, commonly referred to as "alert fatigue" [19], has become a significant impediment to operational efficiency across industries. The fundamental problem of alert fatigue lies in poor alert life-cycle management (§ II-B). As shown in Fig. 1, the alert life-cycle encompasses *Alert Rule Generation*, *Alert Firing*, *Alert Handling* and *Alert Rule Refinement*. Existing alert systems, such as Prometheus Alertmanager [20], focus primarily on alert rule configuration and alert firing, neglecting the quality of rules and alerts. This leads to poorly managed rules and frequent alert storms.

Recent efforts in alert denoising [17], [21]–[23] attempt to reduce SRE burden by identifying noise and correlating alerts during the Alert Handling stage. However, with the proliferation of alert attribute combinations, traditional denoising methods struggle to adapt to modern alert complexity (§ II-C1). Even after denoising, alerts typically flag deviations without providing the essential context needed to understand their implication or root causes. This contextual vacuum forces SREs to engage in time-consuming investigations, combining fragmented information from disparate sources to construct a coherent understanding of the underlying problem (§ II-C2). Moreover, real-world systems also experience dynamic changes in traffic patterns and operating environments, making static rules prone to persistent false alarms. Optimizing rules manually requires extensive expert knowledge and is labor-intensive, underscoring the need for intelligent automated solutions (§ II-C3).

AlertGuardian. To address the limitations of existing alert systems, this paper presents the alert life-cycle management practices at *Company-X*, a leading Internet service provider with hundreds of millions of users. We introduce AlertGuardian, a framework collaborating large language models (LLMs) and lightweight graph models to optimize the alert life-cycle. To meet real-time analysis demands, AlertGuardian first leverages graph learning model with the virtual noise to denoise alerts, accommodating variable attributes and filtering noise effectively (§ III-A). It then integrates Retrieval-

Augmented Generation (RAG) with LLMs based on internal knowledge (*e.g.*, system documents, alert rule explanations, incident tickets) to transform cryptic alerts into comprehensive narratives that provide both what and why of emerging issues (§ III-B). Additionally, to thoroughly address alert noise, an offline multi-agent workflow with iterative feedback optimizes rules through deduplication, threshold adjustments, and temporal analysis (§ III-C).

Results. We conduct a comprehensive study on four real-world datasets from representative services (gaming, office, media, and education) at *Company-X* (§ IV). Experimental results show that AlertGuardian achieves high alert reduction ratios (93.82% to 95.50%) across all systems, significantly outperforming baseline methods. Moreover, AlertGuardian provides high-quality alert summaries (98.5% action accuracy) that reduce failure diagnosis overhead. AlertGuardian also improves 1,174 alert rules, with 375 accepted by SREs (32% acceptance rate). The denoise module has been deployed at *Company-X* for more than one year, with summary and rule refinement modules in pilot use for over three months. The deployment of AlertGuardian offers practical insight for addressing alert fatigue in large-scale cloud systems.

Contribution. This paper makes following contributions.

- We propose addressing alert storm from an alert life-cycle perspective, and share practical experiences from *Company-X*, providing valuable insights for future research and implementations in alert systems.
- We propose AlertGuardian, a novel framework that collaborates LLMs and lightweight graph models to optimize the alert life-cycle management in practice.
- We evaluate AlertGuardian on four real-world datasets from *Company-X*, demonstrating robust performance and practical impact across industrial workloads.

II. Background and Motivation

A. Background

Alert mechanisms are fundamental to monitoring systems, enabling timely anomaly detection in cloud systems. We first introduce the background about alert management.

Alert Rules define conditions under which notifications are triggered, typically based on expressions written in a query language such as Prometheus Query Language (PromQL) [24] and LogsQL [25]. These rules periodically evaluate a specified query against a data source from monitor databases (*e.g.*, Promethuse [12]), retrieving monitor data for analysis. As examples in Fig. 2, an alert rule comprises four core components: (1) a query that selects the dataset to evaluate, with syntax dependent on the data source (*e.g.*, PromQL for Prometheus); (2) a condition, such as a threshold, that must be satisfied to activate the alert (*e.g.*, CPU Utilization $> 80\%$); (3) an evaluation interval and duration, determining how frequently the rule is checked (*e.g.*, 1 minute) and how long the condition must persist to trigger an alert instance; (4) some annotations provide extra details for alert responders to help them understand potential issues.

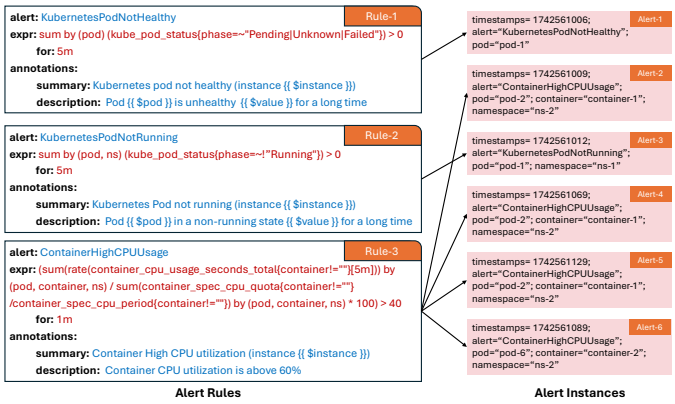

Fig. 2: Examples of alert rules and alert instances.

Alert Instances (hereafter referred to simply as "alert") are fired when the condition of alert rules is met. Each alert is identified by a unique set of attribute pairs, enabling the system to differentiate among multiple alerts triggered by the same rule. As examples in Fig. 2, an alert rule monitoring pod CPU usage could produce alerts with attribute sets {`alert="PodHighCPUUsage", pod="pod-1"`} and {`alert="PodHighCPUUsage", pod="pod-2"`}, representing distinct events due to differing `pod` attribute. The attribute facilitates searching, silencing, and routing of notifications, ensuring precise alert management. Furthermore, annotations, which are another set of key-value pairs, provide contextual details (*e.g.*, alert explanation) to aid responders in diagnosing and resolving issues.

Alert Life-Cycle Management covers the entire process from alert rule creation to alert firing, handling, and feedback, ensuring continuous monitoring and optimization of alerts. As illustrated in Fig. 1, alert management generally follows a cyclical progression comprising four stages:

① **Alert Rule Generation.** In this phase, alert rules are established to monitor various data sources, including metrics, logs, and traces stored in monitoring databases. These rules are often configured by SREs to detect potential failures.

② **Alert Firing.** Once the monitor data meet designated rules, the alerting system fires the corresponding alerts. This step forms a logical bridge between passive monitoring and active response, converting raw measurements into actionable signals. The design of thresholds and conditions here is crucial, as it directly influences both the volume and the fidelity of triggered alerts.

③ **Alert Handling.** Following the firing of alerts, they are routed to SRE teams or automated workflows for thorough investigation. During this stage, SREs review the alerts to determine root causes, assess severity, and execute remediation plans. Detailed logging of all diagnostic and corrective actions is essential for knowledge retention, collaboration, and post-incident analysis. By efficiently triaging and addressing alerts, organizations can minimize downtime and optimize resource allocation.

④ **Alert Rule Refinement.** Insights gleaned from the alert handling process inform the continuous refinement of alert rules. By analyzing patterns of false positives, false nega-

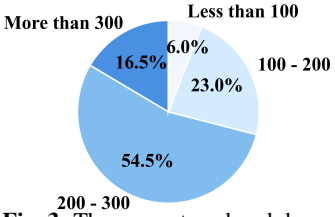

Fig. 3: The percentage breakdown of average per-minute alert volume in *Company-X*.

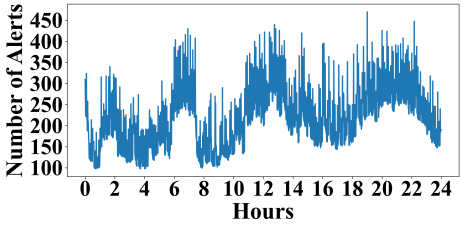

Fig. 4: Change trend of alert volume under 59,607 alert rules in system *A*.

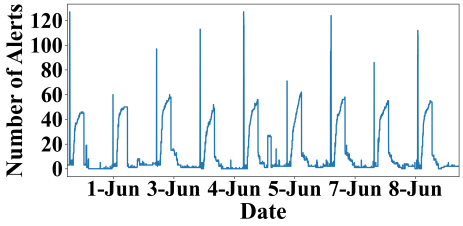

Fig. 5: Change trend of alert volume under 3,544 alert rules in system *B*.

tives, and duplicate notifications, teams can adjust thresholds or query parameters. This iterative practice reduces alert fatigue by eliminating noisy alerts while retaining sufficient sensitivity to capture anomalies.

B. Problems in Our Existing Alert Management System

Our institution, a large-scale Internet company (hereafter referred to as *Company-X*), maintains extensive cloud infrastructures to support a variety of globally popular products with hundreds of millions of users. These products span multiple domains, including gaming, office productivity, and education. In an effort to manage the alerts generated by more than 200 distinct systems, our SRE team at *Company-X* built an in-house alert system. This platform facilitates the creation of alert rules, firing of alerts, and routing of notifications to relevant teams, thereby alleviating some of the operational load on SREs. However, as our systems continue to grow in both size and complexity, several significant limitations have surfaced in the existing alert management framework.

To better understand these issues, we collected nine days worth of alert data and all alert rules from more than 200 systems in *Company-X*, producing a dataset that exceeds 20 GB of alert and 200,000 alert rules in total. In the following, we show the key problems revealed by this investigation.

1) ***Problem 1: Frequent Alert Storms****:* According to the experience of our SRE team, they are inundated daily by alert storms that generate an overwhelming volume of notifications in short timeframes. This phenomenon not only increases the risk of missing critical alerts due to cognitive overload, but also exacerbates the potential for alert fatigue, wherein genuinely critical signals can be overlooked amid a deluge of non-critical ones. As shown in Fig. 3, more than 70% systems fire at least 200 alerts per minute on average, resulting in daily alert counts that can exceed 288,000 per system. Such high-frequency alerts strain SREs and delay diagnosis and remediation.

To better determine which alerts warrant reduction, we correlated system incident records with detailed alert data. The results reveal a marked discrepancy between the number of actual failures and total alerts: some systems generate tens of thousands of daily alerts without experiencing any failures. Further investigation indicates that many of these alerts represent mere "noise", triggered independently of failures. They can be classified into two main categories:

- **Persistent Alerts** remain active whether or not the system is functioning normally. As shown in Fig. 4, a high volume of these alerts recurs daily without any concrete anomalies in the system. Typically, they arise from overly sensitive thresholds in alert rules (*e.g.*, a threshold is set so low that normal fluctuations are flagged as anomalies), resulting in continuous false positives.
- **Periodic Alerts** emerge at regular intervals, leading to recurring alert storms. For example, as depicted in Fig. 5, scheduled night-time restarts on a particular host cause a surge in alerts each day. Because these restarts are not indicative of failures, these alerts similarly constitute noise.

Such noisy alerts consume SRE resources during the Alert Handling phase, hampering effective diagnosis. Consequently, a dedicated denoising step is essential to filter out irrelevant alerts while retaining those associated with system failures.

2) ***Problem 2: Low-quality Alert Rules****:* To investigate the root causes behind frequent alert storms, we analyzed the alert rules responsible for triggering these alerts. Our findings reveal that the system contains an overwhelming number of alert rules. For example, as shown in Fig. 4, system *A* alone has 59,607 active alert rules. A primary contributor to excessive alert noise lies in the low quality of these rules, which are manually created and suffer from two major deficiencies:

- **Functionally Redundant Rules.** Due to the lack of standardized rule design guidelines, multiple alert rules within *Company-X* often overlap in functionality. For instance, `Rule-1` and `Rule-2` in Fig. 2 both aim to detect whether a `pod` is not in the *Running* state, yet they are implemented separately and generate distinct alerts. In addition, different engineers can define rules using varying combinations of attributes (*e.g.*, `pod`, `namespace`, `container`), resulting in duplicate alerts that differ only in attribute values.
- **Inappropriate Static Thresholds.** Alert rules in *Company-X* frequently rely on fixed thresholds (*e.g.*, a CPU utilization threshold of 40% as shown in Fig. 2), which do not adapt to dynamic system behaviors or workload variations. As a result, these rigid thresholds often lead to false positives (*i.e.*, normal fluctuations being misclassified as anomalies) which ultimately flood SREs with irrelevant alerts.

These issues highlight the urgent need for a *Alert Rule Refinement* process to eliminate redundancy, adapt thresholds to evolving environments, and reduce unnecessary alert noise.

C. Why Our Existing Alert Systems Do Not Help?

1) ***Limitation 1: Obsolete Alert Denoising Mechanism****:* In our existing Alert Management System, the alert denoising module predominantly handles alerts defined by fixed attribute combinations [17], [21], [23]. The primary denoising strategy

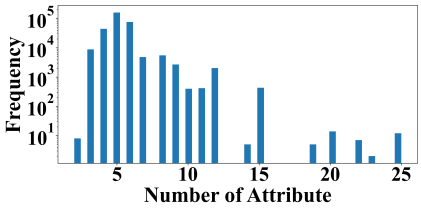

Fig. 6: Distribution of the number of alert attributes in *Company-X*.

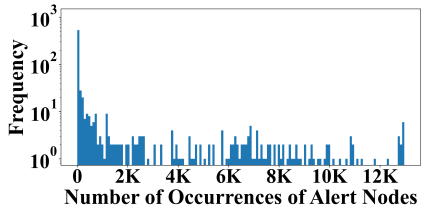

Fig. 7: Distribution of occurrences for alert monitoring nodes in system *C*.

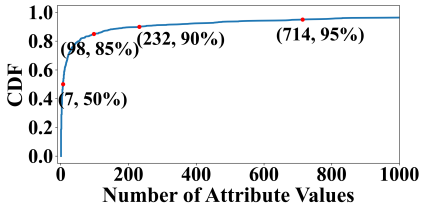

Fig. 8: CDF of the number of values of all alert attributes in *Company-X*.

relies on pairwise co-occurrence frequencies among monitored entities, which was adequate for earlier simplistic systems. However, as the number of managed systems increases and their underlying architectures become more complex, alerts in *Company-X* now encompass significantly broader and more diverse attributes (see Fig. 6). Consequently, the proliferation of attribute combinations exacerbates the issue of low pairwise co-occurrence frequencies, with many alert entities co-occurring only a few times (see Fig. 7). In addition to the high volume of attribute combinations, certain attributes, such as IP addresses and Pod IDs, have extensive value ranges. As illustrated in Fig. 8, over 90% of attributes possess more than 200 potential values, exceeding the capability of the current denoising approach to handle such large-scale variability. As illustrated in Fig. 8, over 90% of attributes possess more than 200 potential values, which exceeds the current denoising approach's capability to handle such large-scale variability, leading to Problem 1 (§II-B1). Therefore, a more adaptable alert denoising method is imperative, one capable of accommodating the vast number of attributes and their increasingly diverse value distributions.

*2) **Limitation 2: Missing Alert Summary Process**:* Even after denoising, individual systems can still generate numerous alerts that require further interpretation. In current practice, these alerts are listed one by one and missing an overarching narrative or aggregated view. Moreover, these individual alerts flag deviations from normal patterns without providing the essential context needed to understand their significance or root causes. This contextual vacuum forces SREs to engage in time-consuming investigations, manually correlating fragmented information from disparate sources, such as system documents, alert explanations, and incident tickets, to understand underlying issues. The heavy reliance on human expertise creates bottlenecks in the diagnosis process, delaying resolution. To address this, an automated alert summarization module is needed to translate cryptic alerts into actionable insights, leveraging internal knowledge (*e.g.*, system documents, alert rule explanations, incident tickets) to provide a coherent summarization explaining what the issue is, why it occurred, and how to resolve it.

*3) **Limitation 3: Missing Alert Rule Refinement Process**:* In addition to the challenges encountered during alert handling, our current system suffers from limited alert rule management (§ II-B2). Once an alert rule is generated in *Company-X*, it is often considered valid indefinitely, lacking an iterative *Alert Refinement* phase in its lifecycle. However, real-world systems frequently undergo changes in traffic patterns and operating

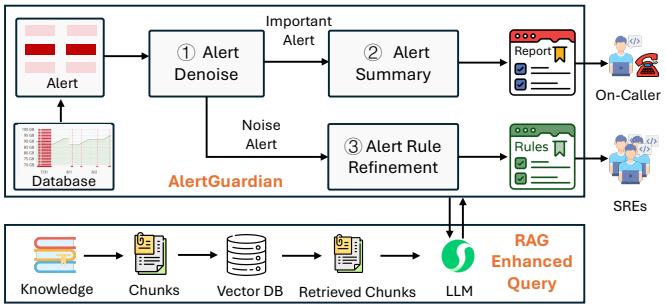

Fig. 9: Overview of AlertGuardian in *Company-X*.

environments, rendering static rules increasingly susceptible to persistent false alarms. Because SREs are often preoccupied with real-time production incidents, there is little opportunity to feed operational insights (*e.g.*, diagnosis result) back into rule design and maintenance, leading to a gradual decline in overall alert policy effectiveness.

III. Experience of Designing AlertGuardian

To overcome the limitations of our alert system (§II-C), we propose a new alert system, named AlertGuardian, designed to manage the entire alert life-cycle. Fig. 9 shows the overview of AlertGuardian, comprising three key phases: ① *Alert Denoise* (§ III-A) filters out high-frequency and periodic alerts based on a graph model, cutting down on noise in real-time. ② For remaining important alerts, *Alert Summary* (§ III-B) provides concise summaries (*e.g.*, potential causes and recommended solutions) for on-call engineers based on a RAG-enhanced LLM. ③ For noisy alerts, *Alert Rule Refinement* (§ III-C) optimizes rules for noisy alerts as an offline task based a RAG-enhanced LLM and multi-agent workflow, driving continuous improvement of alert rule quality.

A. Alert Denoise

To address the obsolete alert denoising mechanism identified in our existing system (§II-C1), this study proposes a graph learning model to effectively handle the vast number of alert attributes and their increasingly diverse attribute value distributions. Instead of leveraging LLM, we opted for a traditional lightweight model for two key reasons: (1) Cost Efficiency: Our systems generate a massive number of alerts daily (§II-B1), and using LLM would incur significant token costs, making it economically unfeasible. (2) Inference Speed: The inherent latency of LLMs, stemming from their autoregressive, token-by-token generation process and substantial computational overhead, presents a critical bottleneck [26]. When processing high-volume alert streams, this latency

makes it infeasible to meet the stringent requirements of real-time diagnosis.

1) Alert Preprocess: For the given alert dataset, we initially segment it into discrete 1-minute time windows to facilitate analysis. Notably, certain alert rules incorporate a pre-defined duration. For example, alerts triggered by Rule-1 in Fig. 2, if unresolved, persist as active for 5 minutes post-firing. To accurately represent this persistence, we duplicated these alerts across the subsequent 5 time windows following their initial occurrence. To isolate noisy alerts from important ones, we introduce a *virtual noisy alert*, fired every minute to emulate persistent noise patterns. The core intent behind mandating the virtual noisy alert's presence each minute is to pinpoint real alerts exhibiting high co-occurrence frequencies with it. These alerts are likely indicative of persistent noise (*e.g.*, false positives from heartbeat checks) that requires exclusion.

For time window i, we construct a relationship matrix M_i to quantify the co-occurrence frequency among the alerts. If alert u and alert v co-occur in this window, the corresponding element $M_i[u, v]$ is marked as 1; otherwise, it is 0. Spanning the time windows from 1 to τ, we obtain a sequence of τ co-occurrence matrices, denoted $\{M_1, M_2, \ldots, M_\tau\}$. These matrices are subsequently aggregated to form a statistical co-occurrence matrix $\mathcal{M}$, encapsulating the co-occurrence relationships among alerts over the entire period. This aggregation yields a sparse matrix $\mathcal{M}$, offering a concise representation of alert co-occurrence suitable for further analysis.

Subsequently, we seek to derive embeddings for the alerts to support advanced modeling efforts. Directly computing embeddings for all alert presents challenges due to their vast quantity. To address this, we compute embeddings for individual alert attributes and derive each alert's embedding by aggregating its constituent attribute embeddings. However, certain attributes, such as Pod ID, possess extensive value spaces and primarily function as entity identifiers rather than indicators of anomalous behavior. Directly embedding these attributes would necessitate processing an excessive number of values, thereby diluting the focus on features pertinent to anomalies. To counter this, we anonymize such identifier attributes by converting their values to a standardized format, "ANON_" appended with the attribute name (*e.g.*, "ANON_POD_ID"). This anonymization reduces the cardinality of attribute values, allowing the algorithm to prioritize computational resources on features directly associated with system anomalies.

2) Graph Learning Model Training: Alerts in a system exhibit complex temporal and content-based relationships [27]. AlertGuardian model temporal correlation using graph structures, where alerts are nodes, and virtual noisy nodes connect to all alerts to simulate persistent alerts. For each node pair (u, v), edge attributes are defined by co-occurrence frequency k and total occurrence count c across all time points. To address content correlation, we adopt a unique attribute-value encoding approach to capture similarity among alerts. Unlike traditional bag-of-words encoding [28], we assign a unique code to each attribute-value pair without considering semantics, focusing on identical pairs. Specifically, we sequentially

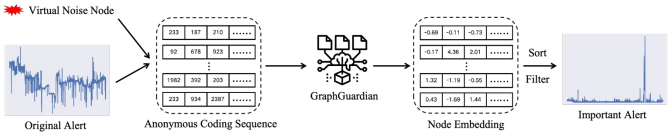

Fig. 10: Example of alert denoise process.

encode attribute-value pairs based on their dataset appearance order. For two alerts, if they share more identical attribute-value pair codes, they are considered more similar.

Our model GraphGuardian integrates Large-Scale Information Network Embedding (LINE) [29] with the Transformer architecture [30] to capture local and global alert relationships. LINE directly utilizes the co-occurrence frequency of alerts to learn embeddings. If two alert nodes are frequently directly connected (high co-occurrence count), their embedding vectors will be very close in space. This is local structural information based on "first-order" and "second-order" proximity. Transformer analyzes all attribute-value pair codes of an alert simultaneously through its self-attention mechanism. It is capable of learning more complex higher-order dependencies that go beyond simple pairwise co-occurrences.This ability allows it to understand the global contextual information within an alert.

For alert pairs (u, v), embeddings h_u and h_v are generated, with similarity measured by the squared cosine distance:

$$d(h_u, h_v) = \left(1 - \frac{h_u \cdot h_v^T}{\|h_u\|\|h_v\|}\right)^2, \tag{1}$$

where distance $d(h_u, h_v) \to 0$ indicates high dissimilarity and $d(h_u, h_v) \to 1$ indicates high similarity.

To handle low-frequency alerts, we use maximum likelihood estimation (MLE) as the loss function, which is less sensitive to infrequent alerts than traditional methods (*e.g.*, mean squared error (MSE), mean absolute error (MAE)). MLE treats co-occurrence frequency as a binomial distribution, optimizing parameters by maximizing the likelihood:

$$\text{Binomial}(k, c, p) = \binom{c}{k} p^k (1-p)^{c-k}, \tag{2}$$

where $p = d(h_u, h_v)$. We chose the binomial distribution because it is less sensitive to low-frequency events, which is very important in our sparse data scenario. The loss is the negative logarithm of the probability mass function:

$$\text{loss} = -\log(\text{Binomial}(k, c, p)). \tag{3}$$

We use the Adam optimizer [31] for efficient parameter updates, combining stochastic gradient descent with adaptive learning rates and momentum. In each iteration, a random alert pair is processed, with the Transformer capturing global dependencies and LINE preserving local connectivity. Minimizing MLE loss improves model performance and generalization.

3) Model inference: During online inference, our graph-based model processes incoming alerts to distinguish critical alerts from noise in real-time. AlertGuardian measures the similarity between alert and virtual alerts based on node-to-node associations, as detailed in Eq. 1. The higher similarity indicates a higher likelihood of being a noise. To determine

noisy alerts, we introduce a similarity threshold $\theta \in [0, 1]$. Alerts with $d(h_u, h_{v_{\text{noise}}}) \geq \theta$ are classified as noise, while others are deemed important. A higher θ increases precision by filtering more alerts, but risks missing critical ones, while a lower θ improves recall at the cost of retaining more noise. In practice, a default θ value of 0.7 is recommended. This value strikes a balance by classifying alerts with a moderately high similarity to the virtual noisy node as noise, while preserving alerts with a lower similarity as potentially critical. Fig. 10 shows an example of an alert denoise process.

Deployment Experience 1: A virtual persistent noisy alert identifies persistent noise patterns via co-occurrence analysis. Anonymizing high-cardinality attributes reduces computational overhead, focusing on anomaly features.

B. Alert Summary

Even after the denoising stage, AlertGuardian can still produce multiple critical alerts (*e.g.*, more than a dozen), posing a risk of overwhelming SREs. As shown in Limitation 2 (§ II-C2), individual alerts flag deviations from normal patterns without providing the essential context needed to understand their implications or root causes. To address this limitation, *Alert Summary* module employs RAG to incorporate internal knowledge (*e.g.*, system documents, alert rule explanations, incident tickets) of *Company-X* into an LLM, thereby generating concise yet actionable alert summaries (*e.g.*, fault explanations, localization, and resolutions).

Why LLM? LLMs are sophisticated neural networks, extensively trained to comprehend and generate text with high contextual awareness. Their key advantage for alert contextualization lies in their natural language processing capabilities, which enable them to interpret technical alert data and translate it into coherent, human-readable narratives, highlighting both symptoms and potential causes. This bridge between machine-generated outputs and human operators significantly accelerates comprehension and response.

Why Connect LLMs to RAG? LLMs have static, general-purpose knowledge, but lack access to private operational insights (*e.g.*, alert rule definitions or tailored troubleshooting steps) of *Company-X*. While fine-tuning an LLM for alert summary is possible, it demands significant computational resources, domain-specific datasets, and risks becoming outdated or overfitted. In contrast, RAG provides a straightforward alternative: it embeds relevant internal documents in a vector database and retrieves Top-K related excerpts to contextualize the LLM prompts dynamically [32]. Because *Company-X* already hosts a mature RAG system for SRE queries, AlertGuardian simply reuses this existing infrastructure. Considering the real-time nature of diagnosis, we employ the general LLM model *Deepseek V3* [33] instead of reasoning model for faster inference speed.

Once *Alert Denoise* module (§III-A) produces its refined alert set, *Alert Summary* module proceeds as follows:

(1) *Per-Alert Retrieval*: For each critical alert in a one-minute window, a RAG query uses its attributes (*e.g.*, rule, component) to retrieve relevant chunks (*e.g.*, alert rule explanation, incident tickets).

(2) *Context Fusion*: Chunks are filtered for relevance and augmented with a cached system context summary (*e.g.*, recent system state).

(3) *LLM Actionability and Summary*: LLM evaluates actionability across the batch, analyzing alerts, knowledge fragments, and relationships (*e.g.*, temporal or causal). If an alert is deemed non-actionable (*e.g.*, transient anomaly), the system remains silent, requiring no SRE intervention. Conversely, actionable alerts yield a consolidated fault summary that describes the root cause, an explanation, a proposed solution, and references to pertinent knowledge.

This pipeline applies chain-of-thought reasoning [34]. Upon determining an alert to be actionable, the module outputs a structured report encompassing:

- **Alerts.** Lists up to five alerts most relevant to the current diagnostic process.
- **Root Cause.** Root cause component leading to the anomaly.
- **Explanation.** A rationale underpinning the fault diagnosis.
- **Solution.** Concrete remediation steps for the root cause.
- **Reference.** Pertinent knowledge link to documents.

Deployment Experience 2: By leveraging RAG, Alert Summary dynamically retrieves internal knowledge, enabling context-aware summaries without costly fine-tuning.

C. Alert Rule Refinement

To overcome Limitation 3 (§II-C3), *Alert Rule Refinement* module optimizes rules responsible for noise alerts identified by denoising module (§III-A). This process reduces recurring noise while maintaining sensitivity to critical alerts. Rather than creating new rules, which could introduce syntax errors or inaccuracies due to LLM randomness [11], the module refines existing rules using the LLM capacity, leveraging alert context and current rule definitions for enhanced precision.

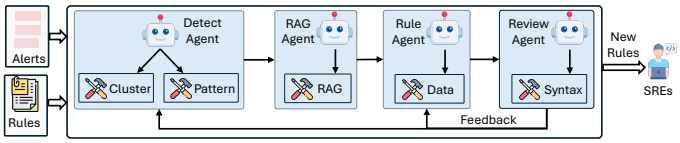

Fig. 11: Multi-agent workflow of Alert Rule Refinement.

Multi-Agent Workflow. As shown in Fig. 11, AlertGuardian introduces a multi-agent workflow to refine alert rules, executed every large time window (30 minutes by default) to balance efficiency and effectiveness. Agents collaborate in a pipeline without a central orchestrator, ensuring streamlined communication. Within a time window, all alerts triggered by a rule are aggregated and based on the denoising module's results, they are classified into noise and critical alerts. For each rule, the workflow is structured as follows:

(1) Detect Agent is used to identify noise patterns with two specialized tools: (i) Clustering Tool applies the HDBSCAN algorithm [35] to group noise alerts by attributes (*e.g.*, rule name, explanation, attributes), detecting patterns like semantically similar alerts without predefined cluster

counts. (ii) Periodicity Pattern Tool identifies persistent or periodic alerts (*e.g.*, daily CPU spikes) based on frequency and timing. Detect Agent categorizes alerts into clusters or patterns requiring refinement.

(2) RAG Agent queries the RAG Vector DB for each categorized group, aggregating key attributes to retrieve relevant knowledge chunks (*e.g.*, rule explanation).

(3) Rule Agent: Refines rules by analyzing categorized alerts and retrieved chunks, employing four policies:

- *Rule Deduplication*: Identifies and merges redundant rules by comparing their semantic intent and operational overlap, using LLM-driven natural language understanding to ensure functional equivalence while simplifying the rule set.
- *Rule Aggregation*: Consolidates rules targeting similar alerts (*e.g.*, Rule-1 and Rule-2 in Fig. 2 both focus on pod not-ready status) by generalizing attributes and introducing conjunctive conditions (*e.g.*, AND logic). The LLM infers common patterns across alerts and proposes unified rules that reduce redundancy while maintaining coverage.
- *Threshold Adjustment*: Analyzes statistical distributions of metrics (*e.g.*, CPU usage, latency) from the knowledge base, using LLM to recommend adaptive thresholds that align with traffic patterns, seasonal variations, or anomaly trends. For instance, it would adjust a CPU alert threshold from 40% to 60% in Fig. 2 based on historical data correlations, suppressing noise without missing critical events.
- *Temporal Analysis*: Incorporates time-based conditions for recurring patterns (*e.g.*, excluding maintenance windows or silencing daily spikes). The LLM enhances this by predicting optimal time windows from contextual cues and suggests precise temporal filters (*e.g.*, "trigger only if sustained for 5 minutes").

The Rule Agent's LLM also generates detailed rationales for each refinement (*e.g.*, "Threshold raised to 60% based on 95th percentile of last 30 days' data"), enhancing transparency and trust.

(4) Review Agent validates refined rules using a syntax checking tool to ensure rule integrity and a simulation tool that tests rules against historical alerts, confirming noise reduction without compromising critical alerts. If validation fails, the Review Agent triggers iterative refinement.

Iterative Feedback Loop. The Detect Agent, Rule Agent, and Review Agent form a feedback loop to iteratively refine alert rules until they meet precise stopping conditions, ensuring reliability and effectiveness. The loop terminates when all the following criteria are satisfied:

- **Syntax Integrity**: Refined rules must be free of syntax errors, verified by the Review Agent's syntax checking tool, ensuring they are executable by the alert system.
- **Preservation of Critical Alerts**: Refined rules must retain all critical alerts, confirmed by the Review Agent's simulation tool, which tests rules against historical data to ensure no critical alerts are suppressed.
- **Noise Reduction Threshold**: The noise alert ratio (proportion of non-critical alerts) must fall below a predefined threshold (5% by default), as evaluated by the Review Agent's simulation tool against historical data, while maintaining full coverage of critical alerts.

During each iteration, the Review Agent assesses the refined rules against these criteria using inputs from the Detect Agent. If any condition is not met, such as syntax errors, loss of critical alerts, or a noise ratio above the threshold, the Review Agent provides targeted feedback to the Rule Agent, prompting further refinement. This process continues until all stopping conditions are satisfied, ensuring robust optimization. If the above conditions are not met after 30 iterations, the optimization process for the rule is terminated and the rule remains unoptimized to prevent excessive computational overhead. The refined rules are then submitted to the SREs for final approval, maintaining a human-in-the-loop framework to ensure operational trust and accountability.

Deployment Experience 3: The iterative feedback loop and LLM-generated rationales provide SREs with transparent insights, boosting confidence in rule adjustments.

TABLE I: Detailed information on production datasets.

Dataset	Domain	#Rule	#Alert	#Incident
$\mathcal{A}$	Game	12,960	2,853,345	138
$\mathcal{B}$	Office	3,544	1,243,259	114
$\mathcal{C}$	Media	59,607	3,883,293	187
$\mathcal{D}$	Education	6,962	2,692,964	179

IV. Experimental Evaluation

To evaluate the effectiveness of AlertGuardian, we conducted an experimental study aimed at addressing the following research questions (RQ):

- **RQ1:** How does AlertGuardian perform in alert denoise?
- **RQ2:** How does AlertGuardian perform in alert summary?
- **RQ3:** How well does AlertGuardian refine alert rules?

A. Experiment Setup

Dataset. We collected alert and alert rule data from four large-scale cloud systems at *Company-X*, supporting gaming, office, media, and education services. As shwon in Table I, these datasets, denoted $\mathcal{A}, \mathcal{B}, \mathcal{C}$, and $\mathcal{D}$, each span 9 days and encompass thousands of services, generating hundreds of thousands of alerts daily. Alerts and alert rules in *Company-X* conform to the Prometheus generic alarm system format, ensuring the generalizability of our approach. Each dataset includes over 100 incidents, with annotations for critical alerts and root cause components. Table I summarizes the dataset characteristics.

Implementation. We implemented AlertGuardian using Python 3.7.2 and PyTorch 1.13.1 [36]. The graph learning model for alert denoising was trained on alerts from the first 6 days of each dataset and evaluated on the final 3 days to ensure temporal separation between training and testing. For alert

summarization and rule refinement, we leveraged *Company-X*'s existing RAG infrastructure, which stores operational knowledge in a vector database. The RAG system, widely adopted by 86% of enterprises deploying LLMs [37], enhances generalizability. We use Deepseek V3 [33] in alert summary and use Deepseek R1 [38] in alert rule refinement provided by *Company-X* by default. All experiments were conducted on a server running TencentOS Server 3.2, equipped with a 24-core Intel Xeon E5-2690 v3 CPU, 128 GB of RAM, and an NVIDIA Tesla V100 GPU.

B. RQ1: Performance in Alert Denoise

We evaluated the effectiveness of AlertGuardian in alert denoising from the following three aspects:

- **RQ1.1:** What is the extent of alert volume reduction achieved by AlertGuardian?
- **RQ1.2:** How accurately does AlertGuardian retain critical alerts for identifying failures?
- **RQ1.3:** How efficient is the alert denoise process?

Evaluation metrics. For RQ1.1, we quantified alert reduction using the reduction ratio, defined as: $\frac{N-M}{N}$, where N is the initial number of alerts, and M is the number of alerts remaining after denoising. This metric reflects the reduction in alert burden for SREs. For RQ1.2, we assessed the accuracy of retaining critical alerts using Precision (P), Recall (R), and F1-score (F1). Precision is the proportion of critical alerts among the retained alerts, Recall is the proportion of true critical alerts retained relative to all true critical alerts, and F1-score is their harmonic mean, providing a balanced measure of accuracy. Critical alerts were derived from SRE-provided incident reports, serving as ground truth. For RQ1.3, we measured computational efficiency by recording the average inference time per alert batch (processed in 1-minute windows) on the experimental hardware.

Baselines. We compared AlertGuardian against the following baseline methods to contextualize its performance:

- Severity-based method is a common heuristic where only alerts with high severity (*e.g.*, critical level in our datasets) are prioritized, reflecting typical SRE practices.
- Cluster-based method UHAS [17] uses Extreme Value Theory (EVT) [39] and Isolation Forest [40] for clustering, retaining only alerts corresponding to cluster centroids.
- Window-based method OAS [15] suppresses a new alert with $A_i^{t_1}$ if it is sufficiently similar to a precedent alert $A_i^{t_0}$ and $t_1 - t_0 < \omega$. We set the window size ω to 1 minutes.
- AlertGuardian without Anonymization (AlertGuardian w/o Non): A variant of AlertGuardian without attribute anonymization, to evaluate the impact of anonymization on denoising performance.

RQ1.1: Effort Reduction. The experimental results in Table. II show that AlertGuardian achieves the highest alert reduction ratios across four cloud systems, ranging from 93.82% to 95.50%, significantly outperforming all baseline methods and demonstrating its superior ability to filter noise. Its success is attributed to the graph learning model and attribute anonymization techniques. In contrast, AlertGuardian w/o Anon, without anonymization, exhibits a 5–6% reduction ratio drop (*e.g.*, from 95.10% to 90.20% in dataset $\mathcal{A}$), highlighting anonymization's importance for handling high-cardinality attributes like Pod IDs. Among baselines, severity-based method performs worst due to its reliance on simplistic heuristics, while UHAS and OAS show moderate performance but lack precision.

TABLE II: Comparison of alert reduction ratios.

Method	Alert Reduction Ratio(%)			
	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$
AlertGuardian	**95.10**	**93.82**	**95.50**	**95.00**
AlertGuardian w/o Anon	90.20	88.00	90.00	89.00
Severity	85.04	83.92	85.00	84.00
OAS [15]	88.00	87.00	88.00	87.00
UHAS [17]	91.00	89.00	91.00	90.00

RQ1.2: Denoise Accuracy. Figure 12 shows the denoise accuracy of AlertGuardian, AlertGuardian w/o Anon, UHAS [17], OAS [15], and Severity-Based across four datasets using Precision, Recall, and F1 Score. As shown in Fig. 12, UHAS and OAS achieve high precision but low recall, as their clustering (UHAS) and window-based suppression (OAS) mechanisms discard some critical alerts, limiting their effectiveness for fault diagnosis. In contrast, AlertGuardian outperforms both, yielding higher precision and recall while retaining fewer alerts, enhancing fault diagnosis efficiency in large-scale systems. The AlertGuardian w/o Anon variant shows reduced performance, underscoring the critical role of anonymization in handling high-cardinality attributes and improving noise filtering. These results highlight the superior balance of precision and recall, making it a robust solution for reliable alert management.

RQ1.3: Efficiency of Alert Denoise. Our objective is to swiftly handle online alert storms and assist SREs in fault remediation, necessitating a highly efficient, low-latency system. During the offline training phase, we employ a distributed framework (*i.e.*, Ray [41]) with data parallelism. The training complexity is approximately $O(N^2 \times M)$, where N is the number of alert entities and M is the number of training epochs. This process is highly efficient in practice: training on a dataset with one million alert ($N = 10^6$) takes only 20 minutes, and for a typical cloud system with data from the last seven days (under five million alerts), training completes within 100 minutes. For online inference, the alert volume for a single system per minute usually stays below 1000. The computational complexity for this stage is approximately $O(N^2)$, enabling the system to perform alert denoising and summary extraction in under 200 milliseconds. Furthermore, response times can be further optimized through techniques like caching for accelerated data access.

C. RQ2: Performance in Alert Summary

To evaluate the effectiveness of alert summary results, we conducted experiments without labeled summary data, combining quantitative and qualitative metrics. Quantitative

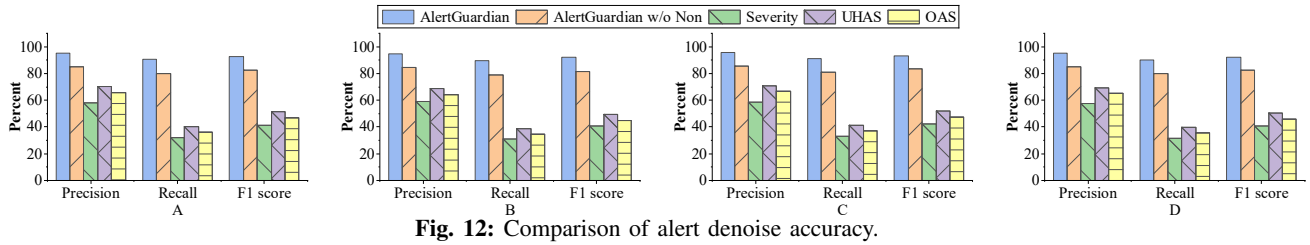

Fig. 12: Comparison of alert denoise accuracy.

TABLE III: Comparison of alert summary performance.

Method	Action Acc.(%)	RCA(%)	Action	Rele.
AlertGuardian (DeepSeek)	**98.5**	**90.5**	**4.8**	**4.9**
AlertGuardian w/o RAG	86.5	82.5	3.0	3.1
AlertGuardian (Qwen)	91.5	88.0	4.0	4.5
OAS [15]	70.0	64.5	-	-
UHAS [17]	72.5	67.0	-	-

metrics included Action/No-Action Accuracy (accuracy in classifying summaries as requiring action or not, validated against incident reports) and Root Cause Analysis Accuracy (accuracy in identifying root causes, assessed via heuristic matching with incident report annotations). Qualitative assessment involved two SREs scoring summaries for Actionability and Relevance (1–5 scale). Due to the high cost of manual labeling, we focused solely on Dataset $\mathcal{A}$. We compared AlertGuardian (RAG + Deepseek V3) against two methods OAS [15] and UHAS [17] that are not based on LLM and two methods AlertGuardian w/o RAG and AlertGuardian (RAG + Qwen 2.5 72B) that are based on LLM.

As shown in Table III, AlertGuardian achieves the best performance, with Action Accuracy of 98.5%, RCA Accuracy of 90.5%, Actionability of 4.8, and Relevance of 4.9. Its RAG-based approach with Deepseek V3 leverages retrieved context to produce concise, actionable summaries that effectively identify critical alerts and root causes. The AlertGuardian with Qwen 2.5 72B baseline performs well but is limited by the smaller model size of Qwen 2.5 72B, reducing its capacity for nuanced summarization. AlertGuardian w/o RAG shows significantly lower performance, as the lack of background knowledge hampers Deepseek V3's ability to contextualize alerts. Non-LLM baselines, UHAS and OAS, yield poor quantitative results and are incapable of generating interpretable summaries, rendering Actionability and Relevance scores inapplicable. Their clustering and window-based mechanisms oversimplify alert relationships, missing critical fault details. The superior quantitative and qualitative performance demonstrates its efficacy in reducing alert fatigue while enhancing fault diagnosis in large-scale cloud systems.

D. RQ3: Performance in Alert Rule Refinement

As described in §III-C, AlertGuardian employs four policies: Rule Deduplication, Rule Aggregation, Threshold Adjustment, and Temporal Analysis. New rules are recommended to SREs, who assess their acceptance. Table IV presents the performance of AlertGuardian, recommending 221–375 rules across datasets, with 73–117 accepted (accept rates 31.2–33.0%). Rule Deduplication achieves the highest accept rates (80.0–83.3%) due to its clear identification of redundant rules. Using LLM-driven semantic analysis, AlertGuardian

TABLE IV: Rule refinement performance of AlertGuardian.

Dataset	Policy	Recommend	Accept	Accept Rate(%)
$\mathcal{A}$	Rule Deduplication	50	40	80.0
	Rule Aggregation	100	28	28.0
	Threshold Adjustment	80	20	25.0
	Temporal Analysis	75	8	10.7
	Total	**305**	**96**	**31.5**
$\mathcal{B}$	Rule Deduplication	36	30	83.3
	Rule Aggregation	75	22	29.3
	Threshold Adjustment	60	14	23.3
	Temporal Analysis	50	7	14.0
	Total	**221**	**73**	**33.0**
$\mathcal{C}$	Rule Deduplication	62	50	80.6
	Rule Aggregation	120	35	29.2
	Threshold Adjustment	100	25	25.0
	Temporal Analysis	93	7	7.5
	Total	**375**	**117**	**31.2**
$\mathcal{D}$	Rule Deduplication	46	38	82.6
	Rule Aggregation	90	26	28.9
	Threshold Adjustment	70	17	24.3
	Temporal Analysis	67	8	11.9
	Total	**273**	**89**	**32.6**

detects overlapping rules, offering low-risk simplifications that SREs readily accept to reduce noise. Rule Aggregation and Threshold Adjustment yields moderate accept rates, as consolidating similar alerts and adjusting threshold risks over-generalization, requiring SRE validation. Temporal Analysis records the lowest accept rates, as time-based conditions are error-prone in variable workloads.

V. Discussion

A. Success Stories

In February 2024, alert denoise module of AlertGuardian was deployed at *Company-X*, remaining in stable operation for over a year. Its alert summary and rule refinement modules have been in pilot use for more than three months in System *A*, one of *Company-X* 's major gaming platforms supporting tens of millions of users in real-time multiplayer game sessions. Prior to implementing AlertGuardian, System *A* faced severe alert storms: over 30,0000 alerts were generated daily from more than ten thousand rules.

AlertGuardian effectively addresses the alert storms with three core features, significantly improving both system reliability and operational efficiency:

- **Alert Denoise.** Leveraging graph-based learning and attribute anonymization, AlertGuardian reduces the system's daily alerts by 95%, from 300,000 down to about 15,000 per day (averaging 10 alerts per minute). Meanwhile, critical alerts maintain an F1-score of 0.92, enabling downstream components to focus on pivotal alerts and filter out a large volume of irrelevant noise.

- **Alert Summary.** For the alerts remaining in each one-minute window after denoising, alert summary module determines whether SRE intervention is required. If not, the system remains silent; otherwise, it clusters key alerts, identifies root causes, generates succinct summaries, and recommends solutions. This action/non-action decision achieves 98% accuracy, substantially alleviating "alert fatigue" among SREs. The mean time to recovery (MTTR) dropped from an average of 156 minutes to 21 minutes, improving efficiency by a factor of 7.4.
- **Alert Rule Refinement.** By analyzing noisy alerts flagged by the alert denoise module, AlertGuardian generated over 300 rule optimization proposals, with near 100 adopted by SREs. Through continuous refinement, the system eliminates over 50,000 false positives per day.

B. Lessons Learned

The development and deployment of AlertGuardian at *Company-X* have provided valuable insights into effective alert life-cycle management for large-scale cloud systems.

Hybrid Model Approach Balances Efficiency and Contextual Depth. AlertGuardian combines a lightweight graph learning model for alert denoising with a LLM for alert summarization, optimizing both computational efficiency and narrative richness. The small model leverages graph-based techniques to process high-volume alert streams, filtering noise while preserving critical signals. In contrast, the LLM enhances summarization by generating human-readable narratives that contextualize alerts with system dependencies and operational insights. This hybrid strategy demonstrates that integrating small models for scalable processing with LLMs for advanced semantic understanding can address diverse requirements in alert management effectively.

RAG-Enabled Knowledge Integration Enhances Scalability. By employing RAG, AlertGuardian seamlessly incorporates private system documentation and operational knowledge without the need for resource-intensive fine-tuning. RAG retrieves relevant context, such as alert rule explanations and incident records, enabling the LLM to produce informed summaries that align with the organization's proprietary environment. This approach highlights the power of RAG as a cost-effective method to enhance alert contextualization, making it adaptable to domain-specific needs in complex cloud systems.

Iterative Feedback Loop Drives Rule Optimization. The rule refinement process relies on an iterative feedback loop that continuously improves alert rules by incorporating insights from noise detection and validation. Agents analyze noise patterns and propose rule optimizations, such as deduplication or threshold adjustments, with LLM-generated rationales for transparency. If proposed refinements fail to reduce noise or suppress critical alerts, feedback triggers further iterations until accuracy is achieved. Final proposals are validated by SREs, ensuring a human-in-the-loop approach. This mechanism underscores the importance of iterative refinement to adapt rules to evolving system dynamics, maintaining long-term effectiveness with minimal manual intervention.

C. Threats to Validity

Internal Threats. Potential biases in data labeling could affect evaluation metrics. To mitigate this, we relied on rigorous incident reports and annotations validated by multiple SREs, ensuring high-quality ground truth. Additionally, LLM hallucinations posed a risk to summarization and rule refinement. We addressed this by integrating RAG to anchor outputs in verified knowledge and employing iterative feedback loops to refine LLM-generated content, enhancing reliability.

External Threats. The datasets, sourced exclusively from *Company-X*, may limit generalizability. However, we mitigated this by including four diverse domains (*i.e.*, gaming, office, media, and education) demonstrating the effectiveness of AlertGuardian across varied workloads. Furthermore, *Company-X* 's use of Prometheus-based alert rules [12], a de facto industry standard widely adopted in cloud monitoring, ensures strong compatibility and applicability to other systems, bolstering the framework's external validity.

VI. Related Work

Alert Rule Management. Major cloud providers, such as Azure [10], Google Cloud [42], and open-source tools like Prometheus [12] and Grafana [43], offer robust alert rule management capabilities, enabling users to create, modify, and delete rules. These systems provide flexible interfaces for defining thresholds and conditions but lack intelligent optimization mechanisms to adapt rules dynamically. DEAR [44] proposes evaluating existing rules to balance high accuracy and low network traffic volume without administrative overhead. However, DEAR primarily focuses on optimizing traffic transmission rather than enhancing rule quality for alert reduction. In contrast, AlertGuardian is the first to leverage LLMs for intelligent rule refinement, enabling iterative rule improvement.

Alert Management. Alert storms in large-scale cloud systems have been widely studied, with several approaches proposed to mitigate their impact on SREs [13]–[17], [22]. Alert ranking methods [22], [45], [46] rank alerts by severity, directing SREs to focus on the most critical issues first. Other works employ alert correlation and aggregation techniques [13]–[15], [47], grouping related alerts into single incidents to reduce the number of actionable items. Additionally, alert denoising approaches [17], [19] identify and suppress noise alerts to alleviate SRE burden. However, these methods assume fixed alert attributes, failing to handle real-world scenarios where attribute counts vary due to system complexity. COLA [48] leverages LLMs' natural language understanding to group related alerts. However, COLA does not address the attribute combination explosion and requires substantial labeled data for supervised fine-tuning (SFT), which is often scarce in real-world alerts. AlertGuardian addresses these gaps with an unsupervised framework that accommodates variable attributes. Furthermore, by integrating RAG-enhanced LLM summarization, our approach provides actionable narratives for critical alerts, accelerating RCA.

VII. Conclusion

In this paper, we tackle the challenge of alert storms from the alert life-cycle management perspective. We introduce AlertGuardian, a novel framework that combines lightweight graph models with LLMs to optimize the entire alert life-cycle. Its central idea is twofold: On the online side, AlertGuardian first identifies and filters out noisy alerts, then leverages internal system knowledge to analyze and summarize the remaining critical alerts based on LLMs for rapid fault diagnosis. On the offline side, AlertGuardian applies LLM-based rule refinement to address the underlying causes of noisy alerts, ultimately preventing recurring alert storms at their source. This integrated strategy not only reduces the immediate burden of alert overload but also improves long-term system maintainability.

VIII. Acknowledge

We greatly appreciate the insightful feedback from the anonymous reviewers. We thank all SREs in the experiments for their feedback. This work was supported in part by National Key Research and Development Program of China (Grant Number: 2024YFB4505904), the National Natural Science Foundation of China under Grant 62272495 and the Guangdong Basic and Applied Basic Research Foundation under Grant 2023B1515020054. This work was also sponsored by Tencent. The corresponding author is Pengfei Chen.

References

[1] X. Li, G. Yu, P. Chen, H. Chen, and Z. Chen, "Going through the life cycle of faults in clouds: Guidelines on fault handling," in *ISSRE 2022*. IEEE, 2022, pp. 121–132. [Online]. Available: https://doi.org/10.1109/ISSRE55969.2022.00022

[2] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, "How to fight production incidents?: an empirical study on a large-scale cloud service," in *SoCC 2022*. ACM, 2022, pp. 126–141. [Online]. Available: https://doi.org/10.1145/3542929.3563482

[3] G. Yu, P. Chen, Z. He, Q. Yan, Y. Luo, F. Li, and Z. Zheng, "Changerca: Finding root causes from software changes in large online systems," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 24–46, 2024. [Online]. Available: https://doi.org/10.1145/3643728

[4] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE TCC*, vol. 10, no. 2, pp. 1100–1116, 2022. [Online]. Available: https://doi.org/10.1109/TCC.2020.2985352

[5] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *ESEC/FSE 2023*. ACM, 2023, pp. 553–565. [Online]. Available: https://doi.org/10.1145/3611643.3616249

[6] G. Yu, P. Chen, P. Li, T. Weng, H. Zheng, Y. Deng, and Z. Zheng, "Logreducer: Identify and reduce log hotspots in kernel on the fly," in *ICSE 2023*. IEEE, 2023, pp. 1763–1775. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00151

[7] Y. Li, G. Yu, P. Chen, C. Zhang, and Z. Zheng, "Microsketch: Lightweight and adaptive sketch based performance issue detection and localization in microservice systems," in *ICSOC 2022*, ser. Lecture Notes in Computer Science, vol. 13740. Springer, 2022, pp. 219–236.

[8] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *WWW 2021*. ACM, 2021, p. 3087–3098.

[9] Y. Chen, C. Zhang, Z. Dong, D. Yang, X. Peng, J. Ou, H. Yang, Z. Wu, X. Qu, and W. Li, "Dynamic graph neural networks-based alert link prediction for online service systems," in *ASE 2023*. IEEE, 2023, pp. 79–90. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00177

[10] "Azure rule management," https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-manage-alert-rules, 2025.

[11] Y. Gu, Y. Xiong, J. Mace, Y. Jiang, Y. Hu, B. Kasikci, and P. Cheng, "Argos: Agentic time-series anomaly detection with autonomous rule generation via large language models," 2025. [Online]. Available: https://arxiv.org/abs/2501.14170

[12] "The prometheus monitoring system and time series database," https://github.com/prometheus/prometheus, 2025.

[13] D. Man, W. Yang, W. Wang, and S. Xuan, "An alert aggregation algorithm based on iterative self-organization," *Procedia Engineering*, vol. 29, pp. 3033–3038, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877705812004456

[14] J. Xu, Y. Wang, P. Chen, and P. Wang, "Lightweight and adaptive service api performance monitoring in highly dynamic cloud environment," in *SCC 2017*. IEEE, 2017, pp. 35–43. [Online]. Available: https://doi.org/10.1109/SCC.2017.80

[15] J. Chen, P. Wang, and W. Wang, "Online summarizing alerts through semantic and behavior information," in *ICSE 2022*, 2022, pp. 1646–1657. [Online]. Available: https://doi.org/10.1145/3510003.3510055

[16] E. Mahdavi, A. Fanian, and F. Amini, "A real-time alert correlation method based on code-books for intrusion detection systems," *Computers & Security*, vol. 89, p. 101661, 2020. [Online]. Available: https://doi.org/10.1016/j.cose.2019.101661

[17] N. Zhao, J. Chen, X. Peng, H. Wang, X. Wu, Y. Zhang, Z. Chen, X. Zheng, X. Nie, G. Wang *et al.*, "Understanding and handling alert storm for online service systems," in *ICSE-SEIP 2020*. ACM, 2020, pp. 162–171. [Online]. Available: https://doi.org/10.1145/3377813.3381363

[18] Z. He, P. Chen, Y. Luo, Q. Yan, H. Chen, G. Yu, and F. Li, "Graph based incident extraction and diagnosis in large-scale online systems," in *ASE 2022*. ACM, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556904

[19] M. E. Aminanto, L. Zhu, T. Ban, R. Isawa, T. Takahashi, and D. Inoue, "Automated threat-alert screening for battling alert fatigue with temporal isolation forest," in *PST 2019*. IEEE, 2019, pp. 1–3. [Online]. Available: https://doi.org/10.1109/PST47121.2019.8949029

[20] "Alertmanager: An open-source monitoring system," https://github.com/prometheus/alertmanager, 2025.

[21] L. Turgeman, Y. Avrashi, G. Vagner, N. Azaizah, and S. Katkar, "Context-aware incremental clustering of alerts in monitoring systems," *Expert Systems with Applications*, vol. 210, p. 118489, 2022. [Online]. Available: https://doi.org/10.1016/j.eswa.2022.118489

[22] N. Zhao, P. Jin, L. Wang, X. Yang, R. Liu, W. Zhang, K. Sui, and D. Pei, "Automatically and adaptively identifying severe alerts for online service systems," in *INFOCOM 2020*. IEEE, 2020, pp. 2420–2429. [Online]. Available: https://doi.org/10.1109/INFOCOM41043.2020.9155219

[23] Z. Chen, J. Liu, Y. Su, H. Zhang, X. Wen, X. Ling, Y. Yang, and M. R. Lyu, "Graph-based incident aggregation for large-scale online service systems," in *ASE 2021*. IEEE, 2021, pp. 430–442. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678746

[24] "Promql (prometheus query language)," https://prometheus.io/docs/prometheus/latest/querying/basics/, 2025.

[25] "Logsql (logs query language)," https://docs.victoriametrics.com/victorialogs/logsql/, 2025.

[26] Z. Jiang, Y. Huang, G. Yu, J. Huang, J. Gu, and M. R. Lyu, "Hierarchical prediction-based management for lmaas systems," 2025. [Online]. Available: https://arxiv.org/abs/2504.03702

[27] Y. Lin, Z. Chen, C. Cao, L.-A. Tang, K. Zhang, W. Cheng, and Z. Li, "Collaborative alert ranking for anomaly detection," in *CIKM 2018*, 2018, pp. 1987–1995. [Online]. Available: https://doi.org/10.1145/3269206.3272013

[28] F. Almeida and G. Xexéo, "Word embeddings: A survey," *CoRR*, vol. abs/1901.09069, 2019. [Online]. Available: http://arxiv.org/abs/1901.09069

[29] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *WWW 2015*, 2015, pp. 1067–1077. [Online]. Available: https://doi.org/10.1145/2736277.2741093

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Neurips 2017*, 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR 2015*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[32] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, Q. Guo, M. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *CoRR*, vol. abs/2312.10997, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.10997

[33] "Deepseek v3," https://github.com/deepseek-ai/DeepSeek-V3, 2025.

[34] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *NeurIPS 2022*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[35] R. J. Campello, D. Moulavi, A. Zimek, and J. Sander, "Hierarchical density estimates for data clustering, visualization, and outlier detection," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 10, no. 1, pp. 1–51, 2015. [Online]. Available: https://doi.org/10.1145/2733381

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS 2019*, 2019, pp. 8024–8035. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[37] "Genai adoption 2024: The challenge with enterprise data," https://www.k2view.com/genai-adoption-survey/, 2025.

[38] "Deepseek r1," https://github.com/deepseek-ai/DeepSeek-R1, 2025.

[39] A. Siffer, P.-A. Fouque, A. Termier, and C. Largouet, "Anomaly detection in streams with extreme value theory," in *KDD 2017*, 2017, pp. 1067–1075. [Online]. Available: https://doi.org/10.1145/3097983.3098144

[40] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *ICDM 2008*. IEEE, 2008, pp. 413–422. [Online]. Available: https://doi.org/10.1109/ICDM.2008.17

[41] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging ai applications," in *OSDI 2018*, 2018, pp. 561–577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/nishihara

[42] "Google cloud rule management," https://cloud.google.com/distributed-cloud/hosted/docs/latest/appliance/admin/create-alert-rules, 2025.

[43] "Compose and scale your observability with one, some, or all of the grafana stack pieces," https://grafana.com/, 2025.

[44] M. Mormul, P. Hirmer, C. Stach, and B. Mitschang, "DEAR: distributed evaluation of alerting rules," in *CLOUD 2020*. IEEE, 2020, pp. 158–165. [Online]. Available: https://doi.org/10.1109/CLOUD49709.2020.00034

[45] G. Jiang, H. Chen, K. Yoshihira, and A. Saxena, "Ranking the importance of alerts for problem determination in large computer systems," in *ICAC 2009*, 2009, pp. 3–12. [Online]. Available: https://doi.org/10.1016/j.cose.2014.12.003

[46] L. Tang, T. Li, F. Pinel, L. Shwartz, and G. Grabarnik, "Optimizing system monitoring configurations for non-actionable alerts," in *2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 34–42. [Online]. Available: https://doi.org/10.1109/NOMS.2012.6211880

[47] D. Lin, R. Raghu, V. Ramamurthy, J. Yu, R. Radhakrishnan, and J. Fernandez, "Unveiling clusters of events for alert and incident management in large-scale enterprise it," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1630–1639. [Online]. Available: https://doi.org/10.1145/2623330.2623360

[48] J. Kuang, J. Liu, J. Huang, R. Zhong, J. Gu, L. Yu, R. Tan, Z. Yang, and M. R. Lyu, "Knowledge-aware alert aggregation in large-scale cloud systems: a hybrid approach," in *ICSE-SEIP 2024*. ACM, 2024, pp. 369–380. [Online]. Available: https://doi.org/10.1145/3639477.3639745