TSGuard: Automated User-Centric Incident Diagnosis for AI Workloads in the Cloud

YITAO YANG*, The Chinese University of Hong Kong, Hong Kong
YANGTAO DENG, The Chinese University of Hong Kong, Hong Kong
YIFAN XIONG, Microsoft Research, Canada
BAOCHUN LI, University of Toronto, Canada
HONG XU, The Chinese University of Hong Kong, Hong Kong
PENG CHENG, Microsoft Research, USA

AI workloads incur frequent failures and incidents from the underlying infrastructure. The current incident management workflow follows a provider-centric paradigm, where users report incidents to the infrastructure provider who then conducts troubleshooting. Due to the large number of incidents and the manual nature of the troubleshooting process, the provider often takes several days to resolve an incident, resulting in operational delays and productivity loss.

To address these challenges, we present TSGuard, a user-centric multi-agent system that delivers immediate incident diagnosis to users who deploy the workloads. The core innovation of TSGuard is twofold: (1) constructing domain-specific knowledge bases by mining historical on-call experiences in the offline phase, and (2) mimicking human expert diagnosis via structured reasoning and iterative trial-and-error in the online phase. Evaluation using production incident records from Microsoft Azure demonstrates that TSGuard significantly outperforms state-of-the-art baselines, improving diagnostic accuracy by 19.8%. Furthermore, TSGuard reduces the average verification time by 63.4% compared to the sequential execution baseline.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; • **Computer systems organization → Cloud computing**; • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: Root Cause Analysis, Incident Diagnosis, Large Language Models, AIOps, Cloud Infrastructure

ACM Reference Format:
Yitao Yang, Yangtao Deng, Yifan Xiong, Baochun Li, Hong Xu, and Peng Cheng. 2026. TSGuard: Automated User-Centric Incident Diagnosis for AI Workloads in the Cloud. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE012 (July 2026), 24 pages. https://doi.org/10.1145/3797149

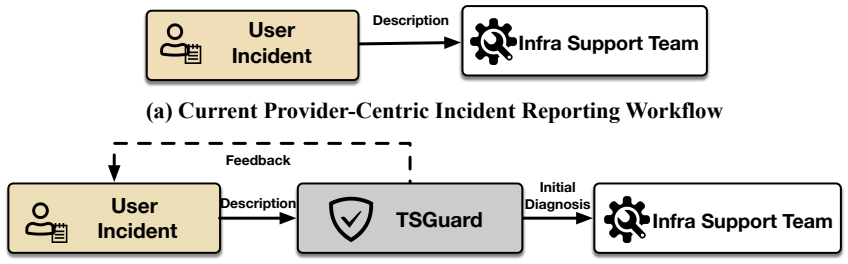
1 Introduction

With the exponential growth of AI workloads [3, 16, 38, 68], operational failures [12, 15] have become increasingly prevalent in AI infrastructure in the cloud. Moreover, due to the synchronous nature of these workloads, failures in large-scale AI infrastructure can have a disproportionately larger impact [13, 62]. Such failures, if not handled promptly, can lead to significant disruptions,

*Part of this work was done during an internship at Microsoft Research.

Authors' Contact Information: Yitao Yang, The Chinese University of Hong Kong, Hong Kong, Hong Kong, ytyang@cse.cuhk.edu.hk; Yangtao Deng, The Chinese University of Hong Kong, Hong Kong, Hong Kong, ytdeng25@cse.cuhk.edu.hk; Yifan Xiong, Microsoft Research, Vancouver, Canada, yifan.xiong@microsoft.com; Baochun Li, University of Toronto, Toronto, Canada, bli@ece.toronto.edu; Hong Xu, The Chinese University of Hong Kong, Hong Kong, Hong Kong, hongxu@cuhk.edu.hk; Peng Cheng, Microsoft Research, Redmond, USA, pengc@microsoft.com.

ACM 2994-970X/2026/7-ARTFSE012
https://doi.org/10.1145/3797149



(a) Current Provider-Centric Incident Reporting Workflow

(b) User-Centric Incident Reporting Workflow with TSGuard

Fig. 1. Comparison of provider-centric and user-centric incident reporting workflow.

causing substantial waste of computational and financial resources of users. For instance, Meta reported 466 job interruptions during a 54-day Llama3.1 405B pre-training session [16], which wasted over 2.12 million H100 GPU hours with an estimated cost of over 18 million dollars based on Azure cloud GPU pricing [40].

We focus on the cloud scenario where the infrastructure is provided to users for deploying their AI workloads. Note that users here can be both internal first-party teams that deploy the provider's own AI workloads, and external third-party users who rent the cloud. They lack direct control of the hardware and comprehensive knowledge about the software stack and the whole infrastructure, while providers are restricted from seeing the specific task settings. These constraints shape the current incident management workflow, illustrated in Figure 1(a), which is largely *provider-centric*: users report incidents often with very limited client-side diagnostic information to the provider, waiting for the provider to identify and resolve the issue.

This workflow, however, suffers from significant limitations due to an inherent knowledge gap between users and the provider, leading to low efficiency and inadequate diagnostic feedback. Users typically lack the technical expertise to provide relevant and detailed diagnostic information, resulting in incomplete and ambiguous incident reports. Further, the effectiveness of this workflow depends heavily on subsequent communication between the user and the on-call engineers (OCEs) handling the case, leading to additional delays. Our one-year analysis of the production incidents in Microsoft Azure underscores this inefficiency: the median time to mitigate (TTM) was 52.5 hours, with a mean TTM of 83.0 hours.

Recent advancements in natural language understanding and tool utilization capabilities [54, 70] of large language models (LLMs) show great potential in bridging this user-provider gap to enable more efficient and timely diagnosis. Several prior works have made initial explorations for using LLMs in cloud incidents, with a primary focus on provider-side automation. Their designs essentially map the incident description to the potential root cause using an LLM, providing a one-shot prediction without any intermediate reasoning or feedback [4, 6, 11, 21, 25, 66, 71, 74].

Our analysis reveals that these provider-centric schemes overlook the user's potential in incident management. Empowering users to self-diagnose incidents first and contact the provider only when needed could significantly reduce ticket volumes and operational labor, leading to a more efficient incident management process. Even for provider-related issues, the user's initial investigation can provide more precise incident descriptions, which can facilitate faster resolution. However, due to users' varying levels of technical expertise and skills, current provider-centric workflow is unable to leverage this potential and places all the burden on the provider. Second, existing methods also oversimplify the unique challenges of AI workloads and infrastructures. We find that the root cause distribution in AI workloads is heavily skewed toward hardware failures, particularly GPU-related issues, which constitute over 50% of incidents and exhibit high recurrence rates. In contrast, over 60% of failures in traditional cloud workloads stem from code and dependency issues [19],

highlighting a fundamentally distinct failure pattern. These hardware failures necessitate physical interaction and metrics for diagnosis [13, 41, 67], which are not used or supported by existing provider-centric diagnostic systems [4, 11, 24].

In this paper, we propose TSGuard, a user-centric system that automates AI incident diagnosis and reporting (Figure 1(b)). TSGuard provides two major benefits: (1) for users, it delivers real-time diagnostic feedback; (2) for providers, it automates initial diagnosis on the user side and generates preliminary diagnostic reports for unresolved cases. These capabilities enable TSGuard to reduce the burden on OCEs and expedite incident resolution.

Achieving these goals is challenging, primarily for two reasons: (1) off-the-shelf LLMs lack domain-specific knowledge of internal infrastructure for accurate incident diagnosis; (2) diagnosing AI incidents solely based on symptoms is insufficient, as root cause confirmation requires verification evidence [67]. TSGuard tackles these challenges in two complementary phases. During the offline phase, TSGuard constructs three structured knowledge components from historical incident records. In online diagnosis, TSGuard employs a tiered diagnostic pipeline to automatically identify potential root causes through systematic hypothesis generation and verification, delivering real-time diagnostic feedback to users. For unresolved incidents, TSGuard escalates them with preliminary diagnostic reports to the infrastructure support team, thereby reducing their burden.

We implement a prototype of TSGuard and evaluate its performance using one-year production incident records from Microsoft Azure. The evaluation demonstrates that TSGuard achieves average Micro F1 and Macro F1 scores of 0.854 and 0.816, respectively. These scores significantly outperform those of the state-of-the-art (SOTA) baseline, RCACopilot [11], by 19.8% and 43.6%. Moreover, TSGuard reduces the average verification time by 63.4% compared to the sequential benchmark execution baseline.

Our contributions are summarized as follows:

- We identify inefficiencies in the current provider-centric incident management workflows for AI workloads, where inherent knowledge gaps between users and infrastructure impact diagnostic accuracy and resolution times.
- We analyze existing LLM-based incident diagnosis systems and expose their limitations in addressing the complexities of AI workload incidents.
- We propose TSGuard, a user-centric system for automated AI workload incident diagnosis. TSGuard leverages domain-specific knowledge to mimic the reasoning and diagnostic process of OCEs and streamline incident diagnosis and reporting.
- We implement TSGuard prototype and evaluate its performance using real-world incident records, demonstrating its effectiveness compared to various baselines.

2 Background and Motivation

We start by introducing our production incident analysis of AI infrastructure and the limitations of the current provider-centric workflow. We then discuss the opportunities and challenges for leveraging LLMs in incident diagnosis.

2.1 AI Workload Incidents in the Wild

While prior studies have extensively investigated incidents in traditional cloud workloads [4, 11, 14, 19, 28, 74], AI workload incidents present substantially more complex challenges. To establish this complexity, we collect and analyze ~1,300 real-world incident data from production GPU clusters serving users' AI workloads at Microsoft Azure spanning a one-year period (2023-04 to 2024-03). The incidents are either reported by users or detected by the cloud provider's monitoring systems, and contain error symptoms (error messages, logs, etc.), root causes, postmortem discussions of OCEs, and resolution steps. More details about data collection and screening are provided in §5.1.

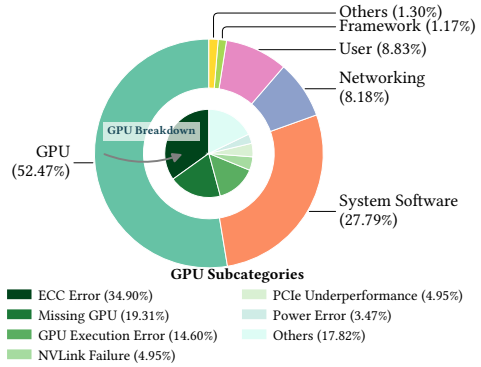


Fig. 2. Breakdown of root cause categories for AI infrastructure in Microsoft Azure. Detailed definitions of each category are provided in §5.1.

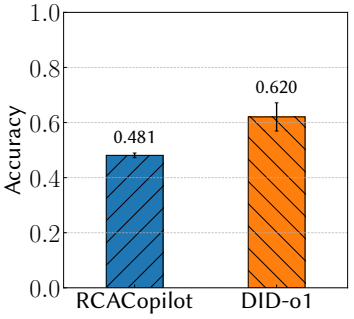


Fig. 3. Performance of two LLM-based diagnosis systems on AI workload incidents.

Root Cause Characterization. Modern AI infrastructure is built upon advanced hardware like high-performance GPUs (e.g., NVIDIA H100 [42], AMD MI200 [5]) and specialized interconnects (NVLink [44], InfiniBand [45]). Unlike traditional infrastructure that primarily rely on relatively stable CPU, memory, and storage resources, AI infrastructure faces more demanding conditions, especially during large-scale, extended model training with thousands of GPUs [16].

As a result, AI workload incidents exhibit distinctive root cause distributions compared to cloud workloads. Our one-year analysis (Figure 2) shows GPU-related failures comprise 52.47% of AI incidents. Among them, ECC errors (34.90%), missing GPUs (19.31%), and execution errors (14.60%) are the primary contributors. Conversely, production cloud workloads primarily face code/config bugs (~40%), dependency failures (16.4%), and general infrastructure issues (15.6%) according to [19]. This disparity highlights the distinct challenges of managing AI workload incidents, as the related GPU, IB, and even software issues (CUDA, PyTorch, etc.) are undergoing rapid development and require much expertise to understand [67].

Failure Pattern Analysis. We analyze the recurrence rate of incidents over the year to identify persistent failure patterns in AI workloads (Table 1). The recurrence rate, calculated as total incidents in a category within the year divided by its distinct fault types, measures the average reoccurrence of each distinct fault type. Higher values indicate more frequent repetition of the same underlying faults. Our analysis reveals that GPU-related incidents exhibit the highest recurrence rate (8.78), followed by networking (3.15) and system software (2.34). This pattern stems from the large-scale deployment of GPU clusters with thousands of interconnected devices, where hardware failures and network issues tend to recur systematically. In contrast, conventional workload incidents are typically caused by code or software errors. These errors often recur within a short time frame but tend to stop once the software is patched or updated, as noted in [11].

Non-Indicative Symptoms and Unreliable Symptomatic Diagnosis. Unlike traditional systems, the symptoms of AI incidents are often weak indicators of root causes. In traditional infrastructure, a symptom can map cleanly to a specific issue (e.g., a DNS resolution failure in a mail server typically signals UDP hub port exhaustion [11]). By contrast, AI workloads exhibit many-to-many symptom–cause relations: a single fault can produce multiple, seemingly unrelated signals (e.g., reduced CPU utilization and NVLink bandwidth with error codes), while a single symptom (e.g., a generic GPU error) may arise from hardware, driver, or resource issues [13]. This necessitates diagnosis beyond one-shot symptom matching.

For example, a user reported a CUDA error "`invalid device ordinal`" with an NVIDIA `Xid 119` in `kern.log`, alongside drops in CPU utilization and NIC throughput. Rebooting temporarily

Table 1. Per-category recurrent failure frequency (higher values indicate more frequent occurrences).

Main Category	Recurrence Rate
GPU	8.78
System Software	2.34
Networking	3.15
User Apps	1.86
Framework	1.12
Other	1.11

Table 2. Average count of semantically distinct incident descriptions per root cause category.

Root Cause Category	Distinct Descriptions (per 10 incidents)
ECC Error	5.6
NVLink Failure	5.2
IB Networking Error	7.6
Illegal Memory Access	4.8

cleared the error, but it recurred after training resumed. The final root cause was not GPU hardware failure, but a CUDA driver version mismatch between the Docker image and the VM.

2.2 Limitations of Current Workflow

Current Incident Management Workflow. In the typical incident lifecycle, users initiate the process by reporting an incident with a description of the issue through a ticketing system. The ticket is then routed to the appropriate team for follow-up, i.e., triaging [8, 10, 18]. OCEs investigate the incident by running additional tests (e.g., NCCL tests [43] and perftests [2]) to reproduce the issue, checking the error logs, and examining the hardware counters for cross-verification. This process may also include consulting with other teams when needed, as well as collaborating with the user to identify the root cause and implement a resolution [10, 11].

Limitations. The current workflow is *provider-centric*, i.e., designed from the provider's perspective since it is the sole party dealing with the incident here. Existing work on automated incident management is also provider-centric [4, 6, 8, 11, 14, 20, 25, 29, 55, 59, 63, 66, 71, 74]. However, this view overlooks the difficulties faced by users, which in turn compromise the overall efficiency of the process. We detail these limitations now.

L1: Inefficiency due to Reporting Quality. The effectiveness and efficiency of provider-centric workflow heavily depend on the quality of initial incident description reporting and subsequent user communication. However, we find that the incident description varies significantly among users. While some provide comprehensive documentation, including detailed error messages, logs, and reproduction steps, others submit minimal information such as basic error stack traces. To quantify the inconsistency in incident descriptions, we randomly selected ten incidents from each specific root cause category and analyzed their descriptions. Using cosine similarity-based clustering [23], we calculate the number of unique clusters, representing semantically distinct incident descriptions. Our analysis (Table 2) reveals that for a single root cause, users provide an average of 4.8 to 7.6 distinct descriptions. This diversity in reporting requires additional communication and troubleshooting, which in turn significantly delays resolution.

L2: Hidden Costs due to User Mistakes. The current workflow requires the provider to perform full troubleshooting for all reported incidents, regardless of their root cause and origin. For incidents caused by user-side misconfigurations or code errors, this leads to much wasted time and resources, as the provider's support efforts are spent on handling issues outside their responsibility. This also causes prolonged resolution time for users in turn.

L3: Underutilized User Potential in Incident Management. If users were able to diagnose the incidents themselves and handle their own mistakes before contacting the provider, they could significantly reduce the time to resolution, and the OCEs' workload could also be reduced. Further, even for incidents caused by provider issues, user-side initial investigation can provide valuable information and better incident tickets to the provider, facilitating faster resolution. Yet, because

users possess varying levels of technical expertise and many lack the skills needed to troubleshoot complex issues, the current workflow is unable to leverage this potential and forces all burden on the provider's side.

2.3 User-Centric Incident Diagnosis

In light of the above limitations of provider-centric workflow, we propose to develop a *user-centric* incident diagnosis framework that runs on the client side to diagnose incidents as soon as they appear. It serves the dual benefits of (1) efficiency improvement: identifying user errors without unnecessarily relying on the provider, and (2) diagnosis feedback: providing more comprehensive and useful initial investigations for the users and provider, especially when the incident cannot be resolved by users. Both benefits lead to faster incident resolution for users and reduced workload for the provider compared to the provider-centric workflow.

More specifically, we leverage LLMs with strong natural language understanding and tool usage capabilities [46, 54, 70] to close the user knowledge gap and empower them to diagnose incidents independently like an expert. The prevalence of text-heavy artifacts in incident workflows (e.g., reports, error logs, traces, and OCE discussions) makes LLMs a natural fit for this diagnostic role.

However, using LLM to solve AI workload incidents still faces key, under-addressed challenges:

Ch1: Lack of Iterative Feedback and Self-verification Ability. Many LLM-based root cause analysis (RCA) systems attempt to map incident descriptions to root causes in a single, non-iterative step using fine-tuning or retrieval augmented generation (RAG) [4, 6, 8, 11, 20, 25, 29, 55, 59, 63, 66, 71, 74]. This "one-shot" paradigm omits the critical processes of hypothesis iteration and evidence-backed confirmation. Consequently, it struggles when initial symptoms are non-indicative, as is common in AI workloads. For instance, on our dataset, the direct prediction models RCACopilot (GPT-4o + RAG) and DID-o1 (o1-preview) achieve low accuracies of only 48.1% and 62.0%, respectively (Figure 3). These results reveal a fundamental process limitation: without an embedded mechanism for verification and feedback, diagnoses are less transparent and inherently less trustworthy.

Ch2: Semantic-Based Retrieval Alone is Unreliable. The reliance on semantic retrieval from incident descriptions is a primary source of diagnostic errors. Our analysis of RCACopilot's [11] incorrect predictions reveals that its RAG-based design frequently retrieves historical incidents that are semantically similar but contextually and fundamentally irrelevant to the ongoing incident. While methods exist to enhance retrieval accuracy [53, 60], the fundamental limitations of retrieval-based methods remain since semantic similarity is not a reliable proxy for causal relevance in the AI incident domain (§2.1) and can be attributed to multiple factors [67]. Thus, improving diagnostic coverage and reducing errors remain critical challenges.

Ch3: Domain-Specific Knowledge is Missing. LLMs often lack proprietary, rapidly evolving infrastructure knowledge, which is critical for diagnosing AI incidents with specialized hardware/-software. We showcase such an example here from our empirical dataset. In Microsoft Azure, one VM type uses eight InfiniBand (IB) NICs for the data plane and one Ethernet NIC for the control plane. Due to security limitations, the standard IB tools (i.e., `ibv_devinfo`) are unavailable inside containers deployed on these VMs, even under normal operating conditions. When troubleshooting an incident with `NCCL WARN NET/IB "Unable to open device mlx5_1"` error, an LLM (GPT-4o) incorrectly determines hardware failure as the most likely root cause and suggests using `ibv_devinfo` to verify the issue. The actual root cause, found by OCEs, is a user misconfiguration of the `NCCL_IB_HCA` environment variable, which mistakenly involves the Ethernet device (`mlx5_1`). The OCEs are aware that `mlx5_1` is the designated interface for the control plane, whereas LLMs do not know.

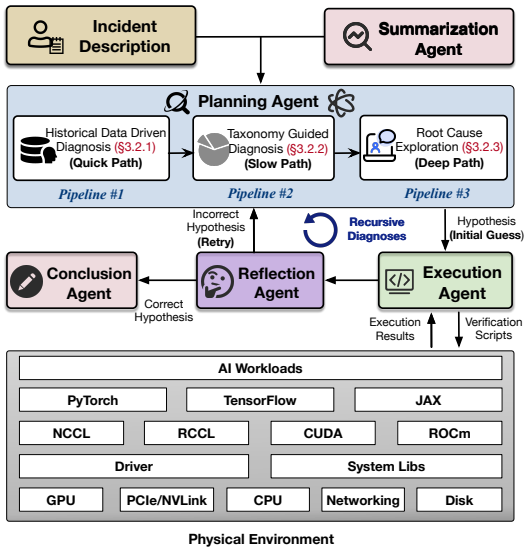


Fig. 4. Online phase overview in TSGuard.

3 TSGuard System

To address the challenges in §2, we propose TSGuard, a user-centric multi-agent system for automated incident diagnosis in AI workloads. TSGuard features a two-phase architecture: an offline phase for consolidating three types of internal knowledge from historical incidents (§3.1), and an online diagnosis phase that emulates expert diagnostics by iteratively testing hypotheses in real-time until a root cause is confirmed (§3.2). Figure 4 illustrates the multi-agent architecture of TSGuard in the online phase.

3.1 Offline: Knowledge Consolidation

To address the critical gap of missing domain knowledge in off-the-shelf LLMs (Ch3), TSGuard constructs three types of knowledge from on-call experience, including (1) Historical incident database: enables semantic failure pattern matching (§3.2.1) in pipeline #1; (2) Hierarchical taxonomy: organizes the root cause labels for structured and staged reasoning (§3.2.2) in pipeline #2; (3) Domain-specific rulebase: encapsulates expert knowledge as reusable prompt templates to guide both offline and all online decisions throughout pipelines #1, 2, and 3.

Incident Gathering and Labeling. Our methodology for creating a structured knowledge base begins with gathering and labeling historical incidents from Microsoft Azure. Each incident record consists of two primary components: **1. Incident Description**: Contains technical artifacts reported by users, such as symptom observations, execution stack traces, and preliminary cause analyses. **2. Postmortem Discussion**: Documents the iterative human investigation process, including multi-team discussions, hypothesis verification, and root cause confirmation. We synthesize these components as input for an LLM, which is tasked with generating a structured, hierarchical root cause label. This output label is formalized as a triplet: <*main_category*>.<*sub_category*>.<*detailed_error_msg*>, and use prompt engineering to restrict the *main_category* into six fixed categories: `GPU`, `System Software`, `Interconnect & Networking`, `Framework & Library`, `User`, and `Other`. These categories are chosen based on our debugging experience and align with the classifications of other works [13], while the *sub_category* and *detailed_error_msg* fields are dynamically generated by the LLM to capture granular failure context. Detailed definitions of these categories are provided in §5.1.

Algorithm 1: Hierarchical Taxonomy Construction

```
   Input: Incident dataset D;
   Output: Empty Taxonomy T;
1  Function TaxonomyConstruction(D):
2      foreach incident, root_cause ∈ D do
3          # [LLM] Classify root cause by taxonomy and semantics;
4          status ← LLM.ClassifyCause(T, root_cause)
5          if status == "EXIST" then
               ▷ Existing taxonomy node: Link incident ID
6              node = LLM.FindSemanticNode(root_cause)
7              node.incident_ids.append(incident.id)
8          else if status == "NEW" then
               ▷ Novel cause: Extend taxonomy with new node
9              node = T.CreateNode(incident, root_cause)
10             T.insert(node)
11         else if status == "NONE" then
               ▷ Ambiguous cause: Skip processing
12             Continue
13     foreach node ∈ T do
           ▷ Gather all incidents sharing the same root cause
14         info = GatherIncidents(node.incident_ids)
15         # [LLM] Generate detailed description for the root cause;
16         node.description = LLM.Synthesis(info)
17         # [Manual] Assign verification tools to the root cause;
18         node.verification = AssignVerification(tools)
19     return T ▷ Return the constructed taxonomy
```

Historical Incident Database Construction. The historical incident database is constructed to enable the rapid diagnosis of recurring issues through semantic pattern matching. The construction process involves two primary steps: **semantic embedding** and **efficient indexing**. First, we use pre-trained embedding models [65] to vectorize each incident description, encoding contextual semantics into a continuous vector space. Thus, geometric relationships (e.g., cosine distance) can quantify the semantic similarity of an incident. Second, these vectorized descriptions are stored as index items, with their corresponding root cause labels attached as metadata. During diagnosis, TSGuard computes semantic similarity scores between the target incident and historical incident vectors, retrieves the top-K nearest neighbors using an approximate nearest neighbor (ANN) retrieval, and formulates hypotheses by aggregating the root cause labels (metadata) from these retrieved neighbors.

Hierarchical Taxonomy Construction. The incident taxonomy organizes root cause labels hierarchically, which is a common practice in cloud services [14]. This structure improves maintainability by localizing updates to specific branches without affecting the whole framework. However, manual construction typically demands multi-person-year effort [14] (expert elicitation, hierarchy validation, and alignment with evolving services), which does not scale.

To address scalability issues, we propose a semi-automated framework that leverages LLMs to derive the taxonomy directly from unstructured incident records. As outlined in Algorithm 1, the workflow has two sequential phases: **initial structure generation** and **expert knowledge enrichment**. Starting with an empty taxonomy T and the incident set D, Phase 1 (lines 2–12) iterates over all the incidents to construct the initial taxonomy. For each root-cause label, the LLM

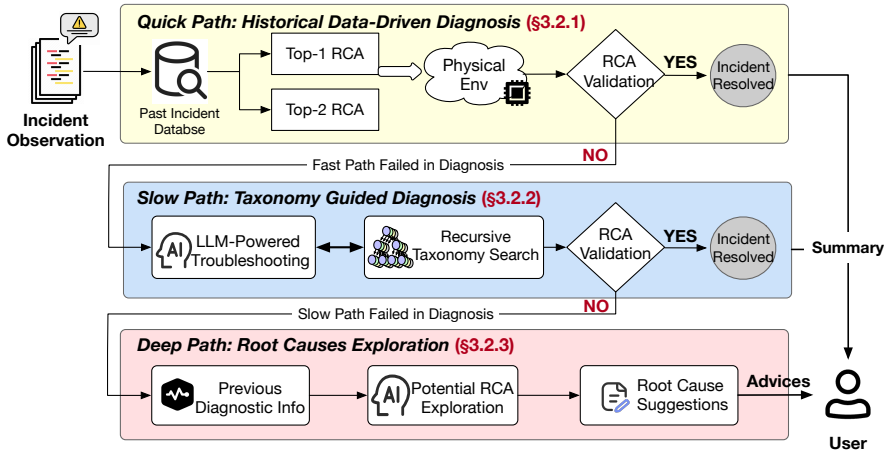


Fig. 5. Tiered pipeline design for online incident diagnosis.

determines whether and where its root-cause label should appear in T (line 4). Given the triplet-format label and the current taxonomy context, the model classifies the label as new, existing, or none. Based on this classification, it either creates a new node in T, links the incident to an existing node, or skips the label. This iterative process ensures the taxonomy remains comprehensive, flexible, and adaptable to emerging failure patterns.

Phase 2 (lines 13–19) enriches the taxonomy structure with actionable diagnostics. We first group incidents by root cause and use LLMs to synthesize concise descriptions for each node. Second, experts then assign verification steps per node using industry-standard tools, including SuperBench [67], NVIDIA DCGM [41], NCCL-test [43], dmesg [7], Azure Health Check [37], etc. Notably, user-related incidents are excluded from this step, as they are typically difficult to diagnose with automated verification tools. This semi-automated workflow substantially reduces manual effort, allowing experts to focus on high-value tasks like validating new failure types and assigning context-specific verification logic.

A visualization of the taxonomy, built from one year of production incidents in Microsoft Azure, is provided in Figure 7. The taxonomy has three levels: 6 main categories, 28 subcategories, and 97 detailed error categories. Main and detailed categories include actionable verification steps for diagnosis, whereas intermediate subcategories serve only as organizational groupings.

Domain-specific Rulebase. To enhance incident diagnosis, we empirically encapsulate a set of text-based rules derived from on-call experience. These rules are formulated as natural language statements that incorporate system configurations, internal software descriptions, common error categories, and their corresponding explanations. The rulebase serves two critical functions: (1) During offline processing, it guides automated labeling and taxonomy generation; (2) For online diagnosis, it provides guidance for the Planning Agent's hypothesis generation during real-time troubleshooting. In TSGuard implementation, the rulebase is embedded into the prompt templates for both offline and online phases, ensuring consistent rule enforcement across all stages of the workflow.

3.2 Online: Tiered Diagnosis Pipelines

This entire online phase (Figure 5) is architected to overcome key diagnostic challenges. To address the lack of iterative verification (Ch1), it employs a multi-agent framework that mimics expert reasoning through a dynamic cycle of hypothesis generation, evidence collection, and validation. To tackle unreliable semantic retrieval (Ch2), it uses a hierarchical pipeline that escalates from

Algorithm 2: Taxonomy-Guided Diagnosis

Input: Incident description I, Pre-built taxonomy T;
Output: Validated root cause R or None;

```
1  Function Diagnosis(traversal_node, M):
2      # [Reflection Agent] Goal Found & Backtracking;
3      if IsLeafNode(traversal_node) then
4          if LLM.Validation(traversal_node) then
5              return [traversal_node] ▷ Valid root cause found
6          return [] ▷ Backtrack to alternative paths
7      # [Planning Agent] State Expansion& Branch Pruning;
8      sub_categories ← LLM.Rank(traversal_node, M);
9      # [Execution Agent] Environment Interactions ;
10     foreach sub_category ∈ sub_categories do
11         V ← EnvInteraction(sub_category); ▷ Collect verification results
12         M ← UpdateMemory(M, sub_category, V);
13         R.extend(Diagnosis(sub_category, M)); ▷ Recursive Exploration
14         M ← ClearMemory(M, sub_category, V);
15     return R
16 Function TaxonomyGuidedDiagnosis(I, T):
17     # [State Initialization];
18     root_pointer ← root(T) ▷ Initialize at taxonomy root
19     M ← InitializeMemory(I, T) ▷ Store incident context in agent memory
20     return Diagnosis(root_pointer, M) ▷ Call Diagnosis to get result
```

quick, pattern-based diagnosis Pipelines #1 (§3.2.1) to systematic investigation in Pipelines #2 and #3 (§3.2.2 and §3.2.3) when simple matching is insufficient.

3.2.1 Quick Path: Historical Data-Driven Diagnosis. The first diagnostic pipeline resolves recurring incidents through failure pattern matching, as illustrated in Figure 5. The input is the incident description. The workflow begins with vectorizing the description using the BGE embedding model [65] from the offline phase. This vector queries the pre-built historical incident database to retrieve the top five most similar historical incidents. These candidates are then processed by an LLM reranker [48], which selects up to two most relevant cases based on textual similarity and incident context. The root causes of the selected cases become diagnostic hypotheses. Each hypothesis is represented as a triplet-format root cause label. Subsequently, the execution agent executes the corresponding scripts and benchmarks for the specific root cause hypothesis and collects the results. Then the reflection agent validates the hypotheses based on the collecting evidences. When hypotheses are confirmed, such as the results indicating an obvious GPU fault or network degradation, the conclusion agent compiles a comprehensive diagnostic report for users. If no hypothesis is validated, the planning agent activates the taxonomy-guided diagnosis pipeline stage for further investigation.

3.2.2 Slow Path: Taxonomy-Guided Diagnosis. The slow path complements the quick path (§3.2.1) by performing a comprehensive, taxonomy-guided diagnosis for more complex incidents. We formalize this process as a recursive taxonomy search algorithm (Algorithm 2) and exemplify its key procedures in Figure 6.

State Initialization. The process begins with two inputs: the incident description I and the pre-built taxonomy T. As shown in Algorithm 2 (lines 19-21), the planning agent initializes three key components: (1) a traversal pointer starting at the taxonomy's root, (2) a structured memory to

store the diagnostic state (e.g., conversation history and current hypotheses), and (3) the incident description, which serves as a contextual anchor for further diagnosis.

Recursive Search. From the initialized root, memory, and incident description, the planning agent initiates a recursive exploration of the taxonomy, progressively refining the search space to identify the most probable root causes. The recursive search process integrates three tightly coupled phases: path selection (planning agent), environment interaction (execution agent), and backtracking (reflection agent), as illustrated in Figure 6.

Path Selection (Planning Agent). This process is driven by the LLM's capabilities and guided by the taxonomy structure. At each search step, the planning agent selects up to three subcategories from the current traversal node, with selection criteria based on their relevance to both the incident description and verification results from the execution agent (line 8 in Algorithm 2). In TSGuard, the searching strategy follows the Depth-first Search (DFS), prioritizing the most likely subcategories as initial hypotheses and invoking the execution agent for deeper exploration. When none of the subcategories are deemed relevant, the planning agent terminates the current branch and backtracks to evaluate alternative paths.

Environment Interaction (Execution Agent). For each subnode hypothesis generated by the planning agent, the execution agent interacts with the environment by running predefined verification scripts (lines 11-14 in Algorithm 2), collecting diagnostic data such as verification results and error logs. Nodes without associated scripts bypass execution, directly returning control to the planning agent. All collected evidences are stored at TSGuard's memory, serving dual purposes: (1) guiding the planning agent's hypothesis refinement and (2) enabling the reflection agent's hypothesis validation. This dynamic interplay between taxonomy-driven logic and real-world interaction ensures accurate and systematic diagnoses.

Hypothesis Validation & Backtracking (Reflection Agent). Upon reaching a leaf node, the reflection agent synthesizes accumulated verification evidence to validate the current hypothesis (lines 3-6 in Algorithm 2). It employs LLMs to determine whether the hypothesis is valid or should be rejected. If the hypothesis is confirmed, TSGuard returns the validated root cause. Otherwise, the planning agent backtracks to the parent node and explores untested branches, ensuring no viable path is prematurely excluded.

Figure 6 exemplifies the slow path traversal through a hierarchical incident taxonomy. The process starts from an initial incident description. At each internal node, the *Planning Agent* acts as a navigator, selecting the most likely sub-paths for exploration. Following the chosen path, the *Execution Agent* gathers real-world feedback for each hypothesis. Upon reaching a leaf node, the *Reflection Agent* decides either confirming the hypothesis as the root cause or triggering a backtracking step to explore alternative branches.

3.2.3 Deep Path: Root Cause Exploration. The first two pipelines are designed to diagnose incidents within predefined categories via pattern matching and structured reasoning based on histories data. However, evolving AI infrastructure services and the increasing complexity of AI workloads make it challenging to cover all possible root causes in the taxonomy. To address this limitation, we designed a root cause exploration pipeline as a last resort for diagnosing incidents not covered by the first two pipeline stages.

This pipeline does not run extra scripts or tools, but rather collects verification results from the execution agent from the first two pipelines, using them as context to generate additional potential root causes. If all suggestions prove ineffective, the incident is escalated to the infrastructure support team for specialized investigation. In this case, a conclusion agent automatically compiles a comprehensive diagnostic report, detailing all actions taken by TSGuard, to expedite the support team's understanding and reduce further investigation time.

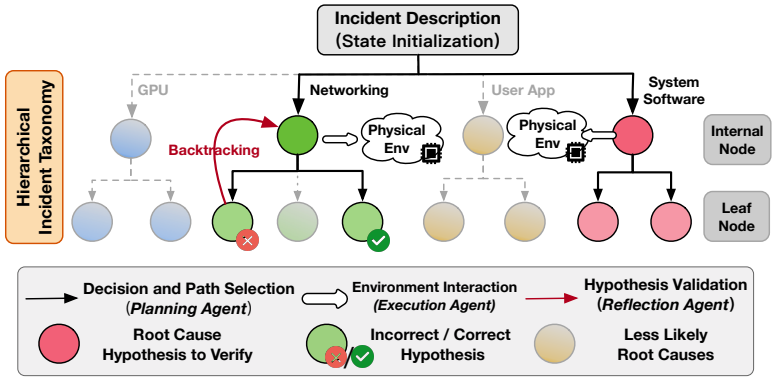


Fig. 6. Illustration of recursive search in taxonomy-guided diagnosis.

4 Implementation

We have developed TSGuard using Python, totaling ~5,200 lines of code (LoC). For historical incident retrieval, we employ Llama-index [31] for data storage and utilize the BGE embedding model [65] and a customized LLM-based reranker [48] to enhance retrieval accuracy. For taxonomy management, we use anytree library [1] to store the taxonomy in JSON format. In our taxonomy generation framework, we use prompt engineering to restrict the <main_category> into six fixed categories: `GPU`,`System Software`,`Interconnect & Networking`, `Framework & Library`,`User`,and `Other`, while the <sub_category> and <detailed_error_msg> are generated by LLM. TSGuard also allows users to extend the taxonomy by adding new categories. Users are required to provide labels, detailed error descriptions, and corresponding verification scripts to facilitate accurate diagnosis.

5 Evaluation

We aim to systematically evaluate the TSGuard design by answering the following four research questions (RQs) and presenting real-world case studies (§5.4) to illustrate how it can disentangle interdependent symptoms.

- RQ1: How does TSGuard compare to baseline methods in terms of diagnostic performance across different incident categories (§5.2)?
- RQ2: How do individual components contribute to TSGuard's overall performance (§5.2)?
- RQ3: What is the computational efficiency and runtime overhead of TSGuard during online diagnosis (§5.3)?
- RQ4: What is the offline overhead for constructing TSGuard's knowledge base (§5.3)?

5.1 Experiment Setup

Incident Ticket Datasets. We collect real-world incident tickets from production-scale GPU clusters in Microsoft Azure, spanning from 2023-04 to 2024-03. For focusing on AI workload incidents, we filtered the incident tickets using keywords such as `GPU`, `CUDA`, `NCCL`, `PyTorch`, `TensorFlow`, etc. For scientific validity, tickets with no clear root causes were manually excluded. This process resulted in a dataset of 778 incidents. Each incident ticket includes a detailed description, identified root causes, on-call engineers' discussion histories, and resolution steps. Following the chronological order, the latter 25% of the incidents (208) were set aside as the test set, while the former 75% were used to construct the historical incident database and taxonomy. The test set covers a diverse range of incident categories: 115 GPU-related incidents, 53 system software issues,

22 interconnect and networking problems, 16 user application incidents, and 2 framework and library-related incidents.

Baselines. We compare TSGuard against the following four baselines:

• ***RCACopilot***: A SOTA LLM-based root cause analysis design for cloud services [11]. Our implementation does not include the information collection stage mentioned in RCACopilot, as it is specifically tailored for email services.

• ***Taxonomy-Guided Diagnosis (TGD)***: This baseline only uses the slow path for diagnosis, taking the incident description as input and using the LLM to traverse the taxonomy for diagnosis (§3.2.2).

• ***Direct Incident Diagnosis with o1 (DID-o1)***: This baseline leverages the advanced o1-preview model [39] for direct incident diagnosis. By employing the "reasoning" model, it accepts incident descriptions as input and offers users the most likely root cause label.

• ***Comprehensive Verification Diagnosis (CVD)***: This baseline adopts a non-selective, brute-force approach. For any given incident, it sequentially executes all available verification scripts to gather comprehensive diagnostic data. The LLM then receives the incident description along with the full output of these scripts, and makes a final judgment on the root cause based on this complete context.

We use GPT-4o [38] from the Azure OpenAI service [35] as the default LLM for all baselines, except for DID-o1.

Metrics. We evaluate TSGuard using Precision, Recall, and Micro/Macro F1-scores, comparing its performance against baseline methods. The Micro F1-score aggregates performance across all classes, weighting each sample equally, while the Macro F1-score evaluates performance on a per-class basis, treating all classes equally regardless of their size. Additionally, we assess its efficiency in terms of verification time and LLM invocation overhead.

Fidelity of Incident Categories. To ensure label accuracy in the offline stage, we perform a rigorous fidelity check on the taxonomy labels used throughout TSGuard's knowledge base. Specifically, three senior on-call engineers (OCEs), each with over three years of production experience, independently validated the taxonomy labels of 50 randomly sampled test set tickets against post-mortem ground truth—i.e., the verified root cause documented in the resolution report. Each OCE assessed whether the LLM-assigned taxonomy label correctly matched the ground truth root cause for each incident. Out of the 50 sampled incidents, 48 labels were unanimously confirmed as correct by all three OCEs, yielding a 96% accuracy rate (48/50). The two mismatched cases involved ambiguous symptom descriptions where the LLM selected a closely related but incorrect sub-category. This validation confirms that the taxonomy is both *complete* (covering production failure modes) and *unambiguous* (labels are semantically distinct enough for independent annotators to agree), supporting its reliability for both knowledge construction and evaluation.

Taxonomy Visualization. The hierarchical incident taxonomy is visualized in Figure 7. Note that this taxonomy is generated by LLMs based on the historical incident records and expert knowledge. It still needs to be validated by domain experts. The main categories in the taxonomy include: **GPU** (hardware-related), **System Software** (driver, CUDA, infrastructure), **Interconnect & Networking** (network and high-speed interconnects), **Framework & Library** (AI frameworks), **User Application** (user errors, configuration conflicts), and **Other**.

5.2 Overall Performance

Effectiveness of Diagnosis. We evaluate the overall performance of TSGuard and all baseline methods using the Micro and Macro F1-scores. The error bars represent the standard deviation of F1 scores across multiple experimental runs.

As shown in Figure 8, TSGuard consistently outperforms all baseline methods in both Micro F1 and Macro F1 scores, demonstrating its superior diagnostic capability. Specifically, TSGuard

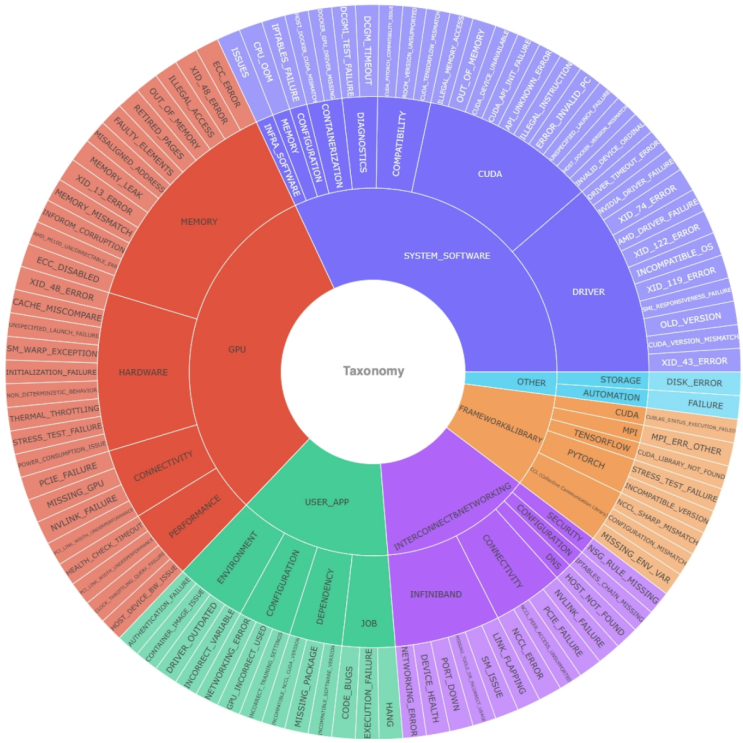


Fig. 7. Visualization of the incident taxonomy.

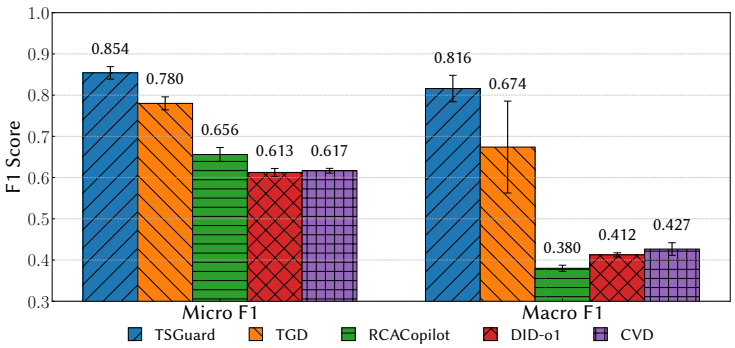


Fig. 8. Comparison of Micro F1 and Macro F1 scores between TSGuard and baseline methods. Error bars represent the standard deviation across experimental runs.

achieves the highest Micro F1 score of 0.854, significantly surpassing the second-best method, TGD, which achieves a score of 0.780. In terms of the Macro F1 score, the gap in TSGuard (0.816) and remaining baseline methods is even more pronounced, with RCACopilot, DID-o1, and CVD scoring 0.380, 0.424, and 0.427, respectively. Notably, despite the integration of comprehensive verification in CVD, its diagnostic accuracy remains limited (0.427 Macro-F1). By checking the results of CVD, we find that the LLM is easy to be misled by trivial testing jitter results and ignores the pivotal benchmarks, indicating that structured reasoning and feedback mechanisms are

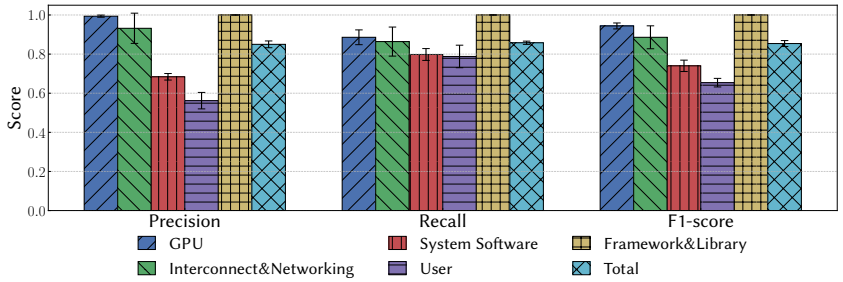


Fig. 9. The Precision, Recall, and Micro F1 Score of TSGuard across incident categories

Table 3. Performance contribution factors analysis in TSGuard.

TSGuard	Average Resolved Incidents	Cumulative Percentage
Historical Data Driven **(Pipeline #1)**	107.0	51.4%
Historical + Taxonomy-Guided **(Pipeline #1 + #2)**	174.9	84.0%
Historical + Taxonomy + Exploration **(Pipeline #1 + #2 + #3)**	179.6	86.3%

Table 4. Performance of TSGuard on unseen incidents.

Tickets	Removed Taxonomy Label	Accuracy After Removal	Accuracy Before Removal
Simple	GPU.MEMORY. InfoROM_Corruption	75%	100%
	SYSTEM_SOFTWARE.CUDA. Illegal_Mem_Access	93.8%	100%
Complex	SYSTEM_SOFTWARE.CUDA. Host_VM_Version_Mismatch	0%	42.5%

essential for prioritizing root causes within hierarchical taxonomies. Moreover, we conduct each experiment five times and find that TSGuard exhibits smaller error bars compared to most of the baseline methods in Micro F1. While for Macro F1, the error bars of TSGuard are slightly larger than RCACopilot, DID-o1, and CVD, this is due to certain categories having fewer samples, leading to higher variance in the evaluation results. Overall, these results underscore the effectiveness of TSGuard in accurately predicting root cause categories while maintaining stability under varying conditions.

Performance Across Incident Categories. To comprehensively assess TSGuard's effectiveness, we conduct a category-wise performance analysis as shown in Figure 9. While TSGuard achieves strong Precision, Recall, and Micro-F1 scores in most categories, its performance significantly degrades for user-related incidents. This limitation stems from inherent biases in the incident collection methodology: these collected "user-related" incidents predominantly represent errors triggered by internal infrastructure components (e.g., deployment scripts or configuration APIs) during user operations. Although these incidents contain infrastructure-originated error messages, their root causes are actually user-side operational errors. This semantic mismatch between observed symptoms (infrastructure errors) and underlying causes (user actions) causes TSGuard to incorrectly prioritize infrastructure-level root causes, thereby reducing its effectiveness on user-related cases.

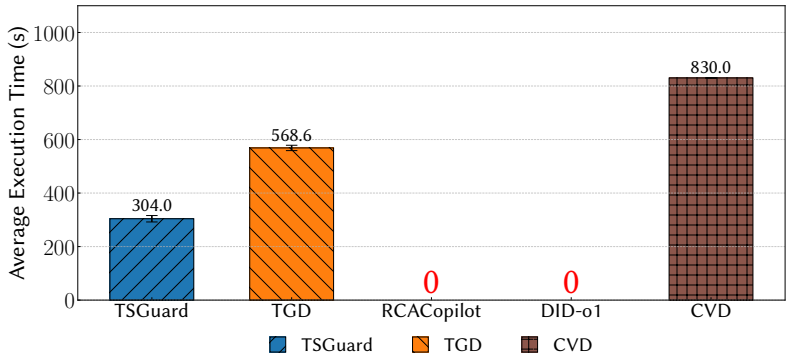


Fig. 10. Average verification time across different diagnostic methods on one single machine.

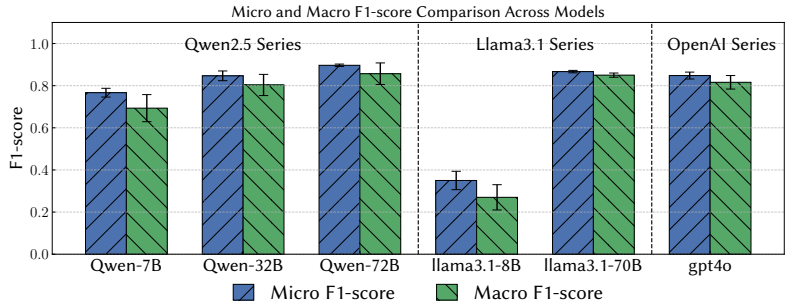


Fig. 11. The F1 scores of TSGuard when using different LLMs.

In-depth Component Performance Analysis. To quantify component contributions, we perform a granular evaluation of the tiered pipeline architecture. As detailed in Table 3, the standalone historical data-driven diagnosis component (Pipeline #1) addresses 51.4% of incidents, demonstrating the foundational value of leveraging past incident data for new incidents. By augmenting this with taxonomy-guided diagnosis (Pipeline #1 + #2), we observe a 32.6 percentage point increase in resolution rate (from 51.4% to 84.0%), validating the indispensable role of structured reasoning in improving diagnostic precision. The full TSGuard design (Pipeline #1 + #2 + #3) further improves performance to 86.3%, with the 2.3-point gain attributable to the exploration stage. These results validate the cumulative efficacy of our tiered design, where each component provides complementary enhancements to TSGuard's overall diagnostic capability.

Handling Unseen Incidents. To assess TSGuard's robustness against out-of-distribution incidents, we simulate label scarcity scenarios by iteratively removing critical labels from the taxonomy and measuring diagnostic accuracy degradation. Results in Table 4 reveal a bimodal behavior: Simple cases (e.g., `infoROM corruption`, `illegal memory access`) maintain 75%–93.75% accuracy upon post-label removal, indicating resilience through the exploration diagnostic path. Complex interdependent failures (e.g., `Host VM CUDA version mismatch`) suffer catastrophic accuracy drops (42.5% → 0%), exposing reliance on explicit taxonomy guidance for multi-fault reasoning. This phenomenon reveals a fundamental trade-off in TSGuard's design: although the root cause exploration phase enables robust performance on routine incidents, the absence of taxonomic anchors critically limits its ability to resolve domain-specific incidents. Nevertheless, most of the incidents were repetitive during our one-year observation, and no significant new incidents occurred in practice.

Impact of LLMs on Diagnostic Accuracy. We also change the LLMs in TSGuard to evaluate the impact of different LLMs on diagnostic accuracy. We use the most recent open-source LLMs:

Table 5. Cost of knowledge consolidation during offline processing.

Processing Stage	Model Used	Invocations	Input Tokens	Output Tokens	Times (Sec)	Cost (USD)
Labeling	GPT-4o [38]	775	1530.2	120.8	60.8	$7.79
Indexing	BGE-large [65]	570	✗	✗	25.4	$0
Taxonomy	GPT-4o [38]	570	2898.1	142.8	448.4	$9.89

Llama3.1 [16] (8B [34] and 70B [33]) and Qwen2.5 [68] (7B [51], 32B [49], and 72B [50]), measure the performance and compare them with the default GPT-4o[38] in TSGuard. Note that, we use the GPTQ [17] quantization version and locally deploy them with SGLang [76].

Figure 11 compares the Micro and Macro F1-scores across these models. The results show that the Qwen2.5 series consistently achieves higher scores compared to other models, with the Qwen-72B demonstrating the best overall performance. The Llama3.1 series, while competitive, shows a noticeable gap in the Macro F1-score, particularly for the smaller 8B model, indicating that model size and architecture significantly impact diagnostic accuracy. These results show that the TSGuard design can also be locally deployed with open-source LLMs and larger models tend to provide better diagnostic accuracy.

5.3 Cost Efficiency Analysis

Verification Time Benchmarking. To quantify operational efficiency, we measure end-to-end verification latency, defined as the total time from diagnostic trigger to actionable result delivery, including script execution and telemetry collection phases. As shown in Figure 10, TSGuard achieves a 63.4% reduction in latency compared to the CVD baseline (304.0s vs. 830.0s), and outperforms TGD by 46.5% (304.0s vs. 568.6s). RCACopilot and DID-o1 are excluded from this benchmark as they lack automated verification modules. This efficiency gain stems from TSGuard's incremental verification strategy (Algorithm 2), which prioritizes high-probability fault candidates identified during tiered diagnosis, thereby avoiding exhaustive checks.

Cost of Constructing Domain-specific Knowledge Base. We quantify offline overhead across three processes: (1) LLM-based incident labeling, (2) embedding/indexing for the historical database, and (3) taxonomy construction. We focus on model invocation costs (other compute is negligible), using GPT-4o [38] for labeling/taxonomy and a locally deployed BGE-large [65] for indexing. Table 5 reports invocations, tokens, processing time, and cost.

The analysis breaks down as follows: First, ticket labeling required 775 GPT-4o invocations, averaging 1530.2 input and 120.8 output tokens per call, which took 60.8 seconds and cost $7.79. Second, indexing the historical incident database involved 570 BGE invocations, a process that took 25.4 seconds and incurred negligible cost due to local model deployment. Third, taxonomy construction utilized 570 GPT-4o invocations, with an average of 2898.1 input and 142.8 output tokens per iteration, taking 448.4 seconds and costing $9.89. While labeling and indexing are parallelizable, taxonomy construction is inherently sequential, as each step depends on the previous one. This sequential nature explains its longer execution time. The resulting taxonomy is visualized in Figure 7.

Cost of Online Diagnosis. We quantify the overhead for online diagnosis by analyzing the average number of LLM invocations, total input tokens, and total output tokens per incident over all baselines, as shown in Table 6. Using Azure OpenAI Service's pricing model [36], we further convert these metrics to monetary costs, which reveals stark contrasts: TGD emerges as the most expensive baseline ($0.253/incident), 2.5× higher than TSGuard. Lightweight methods (RCACopilot, DID-o1, CVD) operate at $0.003 - $0.051/incident, but sacrifice diagnostic accuracy (per Figure 8). TSGuard falls between these two extremes, with a cost of $0.102/incident. However, considering the

Table 6. Average LLM invocations, total input tokens, total output tokens, and price of API calls for each diagnostic method per incident.

Diagnostic Method	LLM Model	# of LLM Invocations	Input Tokens	Output Tokens	Price (USD)
TSGuard	**GPT-4o** [38]	**10.2**	**31801.2**	**2224.9**	**$0.102**
TGD	GPT-4o [38]	18.17	82070.0	4735.5	$0.253
RCACopilot	GPT-4o [38]	1.13	817.1	109.6	$0.003
DID-o1	o1-preview [39]	1	2877.6	136.9	$0.051
CVD	GPT-4o [38]	1	8037.2	160.5	$0.022

higher Micro/Macro F1 score of TSGuard, this overhead is justifiable. Moreover, this overhead can be further reduced using optimization techniques from existing compound LLM systems [30, 58].

5.4 Case Study with Real Incidents

This section presents real-world case studies to demonstrate how TSGuard's iterative diagnosis and structured reasoning can disentangle interdependent symptoms. We also analyze the cases unresolved by TSGuard, revealing their inherent complexity, which poses challenges even for human experts.

Incident #1: NCCL Error: Connection Refused. Figure 12 shows an NCCL connection refused error that caused failures in distributed training. Due to insufficient information in the incident description, TSGuard's first pipeline (§3.2.1) failed to retrieve relevant past incidents. Consequently, TSGuard activated its taxonomy-guided diagnosis pipeline (§3.2.2), visualized in Figure 14(a). Based on the description, the planning agent initially hypothesized `Interconnect & Networking`, `System Software`, and `User Applications` as potential root cause categories. A subsequent DFS search, starting from the most likely category, ultimately identified `NCCL Error` and `NVLink_Failure` as the root causes.

Incident #2: Uncorrectable ECC Error. The second example (Figure 13) involves an uncorrectable ECC error encountered during gpt2 pretraining. The raw incident report exhibited multi-modal symptoms: CUDA kernel failures, GPU memory ECC error, and PyTorch allocation fragmentation. Consequently, using this incident summary as input, Pipeline#1 (§3.2.1) failed to retrieve relevant historical cases, as the incident description diluted the critical ECC error signals. TSGuard then switched to Pipeline #2 (§3.2.2) for deeper analysis, as shown in Figure 14(b). The planning agent first hypothesized potential causes: `GPU`, `Framework & Library`, and `System Software`, ordered by likelihood. Following a DFS approach, TSGuard explored these possibilities and confirmed the root causes as `ECC Error`, `Page Retirement`, and `Xid 48 Error`. The ECC error was the primary trigger, while the other two were its downstream effects.

Unsolved Incidents. Despite TSGuard's strong diagnostic capabilities, some incidents remain challenging. We identified 28 failure cases (~13.5%) that TSGuard failed to resolve in a single run. Their TTM analysis confirms their inherent complexity: while the median TTM for all incidents is 52.50 hours (mean 83.00 hours), these unresolved cases by TSGuard had a median TTM of 122.56 hours (mean 164.39 hours). This significantly elevated TTM indicates that the incidents TSGuard failed to resolve are, by nature, more complex and time-consuming even for human experts.

6 Discussion and Threats to Validity

Internal validity. We audited 50 randomly sampled test tickets and observed 96% agreement between LLM and human annotations (Section 5); the remaining 4% and taxonomy triplet errors may still introduce noise into knowledge construction and evaluation. Our dataset comes from a single provider and is keyword-filtered (GPU/CUDA/NCCL), which may over-represent infrastructure

Example 1 Incident Description - NVLink Error (Root Cause)

Customer failed to deploy distributed job on xx due to rank-0 can't talk to peer node. They tried this for six times (or more) and the bad instance are all point to the same physical node.
Sample user log: `[4] <job_id>:684:1526 [0] include/socket.h:406 NCCL WARN Connect to ip_addr<port> failed : Connection refused`

OCE Summary: All links to GPU1 are inactive. See pasted `nvlink -s` results.

Fig. 12. Incident description of Example 1. The root cause is NVLink failure due to inactive links.

Example 2 Incident Description - ECC Error (Root Cause)

```
node-92: Traceback (most recent call last):
node-92: File "pretrain_gpt2.py", line 227, in <module>
node-92: pretrain(
...
node-92: RuntimeError: CUDA error: CUBLAS_STATUS_EXECUTION_FAILED
node-92: when calling ‘cublasGemmEx(...)‘
node-92: terminate called after throwing an instance of ’c10::Error’
node-92: what(): CUDA error: uncorrectable ECC error encountered
node-92: CUDA kernel errors might be asynchronously reported at some other API call.
node-92: For debugging consider passing CUDA_LAUNCH_BLOCKING=1.
node-92: Exception raised from c10_cuda_check_implementation at
../c10/cuda/CUDAException.cpp:31 (most recent call first):
frame #0: c10::Error::Error... (omitted)
```

OCE Summary: GPU ECC health check failed. This issue was fixed by reboot.

Fig. 13. Incident description of Example 2. The root cause is an ECC Error due to a health check failure.

incidents. We mitigate via per-category reporting, a fixed split, and repeated runs, but cannot eliminate the threat entirely.

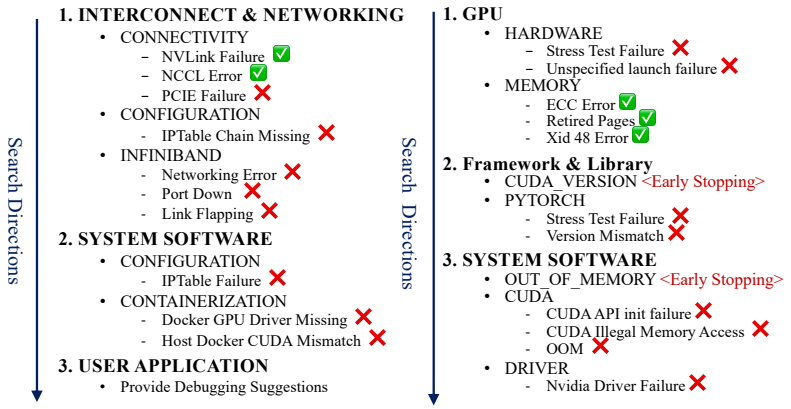
External validity. Our empirical data comes from Microsoft Azure. While the TSGuard methodology is provider-agnostic, the concrete taxonomy, rules, and scripts are influenced by hardware SKU, software stack, and operational processes. Porting to other clouds or on-prem clusters likely requires re-grounding the taxonomy and verification scripts. Likewise, incidents beyond GPU-centric AI workloads (e.g., CPU-only analytics, storage-heavy services) may require new categories/tools before comparable accuracy holds.

Mitigation Suggestions. For safety, TSGuard's scope is strictly diagnosis, not mitigation, as allowing LLM-based systems to perform unsupervised mitigation poses significant risks. During an incident, users still need to wait for cloud provider OCEs to remediate system errors.

Limitations and Future Work. TSGuard's multi-tiered approach inherently supports detecting unseen incidents: its third pipeline targets unknown root causes by analyzing accumulated verification evidence without relying on historical patterns. The current performance gap in the "User Error" category stems from attempting to pinpoint specific user-side root causes, which is impractical due to privacy and permission boundaries. As future work, we plan to establish "Infrastructure Healthy" as an explicit diagnostic conclusion—if TSGuard verifies that hardware, network, and drivers are all functional, this reverse proof confirms the problem lies on the user side, still providing actionable guidance.

7 Related Work

Significant efforts have been dedicated to automated incident troubleshooting for cloud services, with a primary focus on reducing the burden on OCEs [4, 6, 8, 9, 11, 14, 18, 20, 22, 24, 25, 28, 29, 32, 47, 55–57, 59, 63, 66, 69, 71, 72, 74, 75]. Traditional root cause analysis often employs deep learning models trained on historical incident data for prediction [8, 24, 56, 69, 75]. Chen et al. [9] provided an early vision for intelligent incident management, and He et al. [22] proposed graph-based methods for incident extraction and diagnosis. More recently, LLMs have been explored



(a) Search Path for Incident Example 1 (b) Search Path for Incident Example 2

Fig. 14. Diagnosis visualization for the two examples in Figure 12 and 13. Green checkmarks (✓) and red crosses (✗) indicate the reflection agent's confirmation or rejection of hypotheses, while the red <Early Stopping> marker denotes branch pruning by the planning agent.

for automated incident diagnosis, primarily using retrieval-based methods to match incoming incidents with historical records [6, 20, 25, 29, 59, 66, 71]. Advanced approaches now utilize LLM agents for iterative diagnosis, combining retrieval with feedback-driven analysis via tools for logs [26, 52, 73], configuration update [61], or physical environment interaction [20, 63]. Flow-of-Action [47] further enhances multi-agent root cause analysis with SOP-guided workflows. Systems such as Minder [13], MegaScale [27], and Aegis [15] utilize monitoring metrics to detect failover or failslow scenarios in large-scale distributed AI training. XPUTimer [12] and FALCON [64] trace the communication process for detailed diagnosis. Additionally, solutions like Nissist [6], AutoTSG [57], and NetAssistant [59] automate traditional troubleshooting guides for infrastructure-level incident diagnosis. However, these efforts still focus on the provider perspective and certain data sources that they utilize are inaccessible to users. In contrast, TSGuard represents a shift from the "provider-centric" to "user-centric" paradigm, offering two key distinctions: (1) TSGuard acts as a *pre-ticket interception layer* at the moment of ticket submission, enabling user-side diagnosis to filter out resolvable issues before submission, whereas existing works [11, 47] are designed for post-ticket assistance after an incident has been created and assigned; and (2) TSGuard employs *active verification* by running verification scripts to confirm hypothesized faults, while systems like RCACopilot [11] primarily rely on one-shot reasoning over static incident descriptions and logs.

8 Conclusion

In this paper, we present TSGuard, a user-centric system that automates incident diagnosis for AI workloads. During the offline phase, TSGuard constructs internal knowledge bases from historical on-call experiences. For online diagnosis, TSGuard employs sequential diagnostic pipelines that mimic the diagnostic processes of human experts, providing immediate feedback to users and facilitating the creation of accurate incident tickets for unresolved problems. Extensive evaluations using real-world incidents demonstrate the effectiveness of TSGuard against various baselines.

Acknowledgments

We would like to thank the reviewers for their valuable comments. This work is supported in part by the Research Grants Council of the University Grants Committee under Grant 2151323 and by The Chinese University of Hong Kong under Grants 4937007, 4937008, 5501329, and 5501517.

References

[1] 2024. AnyTree. https://anytree.readthedocs.io/en/latest/. Accessed Dec 16, 2024.

[2] 2025. Infiniband Verbs Performance Tests. https://github.com/linux-rdma/perftest. Accessed May 12, 2025.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023). https://doi.org/10.48550/arXiv.2303.08774

[4] Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. 2023. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. https://doi.org/10.1109/icse48619.2023.00149

[5] AMD. 2024. AMD Instinct MI200 Series Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi200.html. Accessed Dec 12, 2024.

[6] Kaikai An, Fangkai Yang, Junting Lu, Liqun Li, Zhixing Ren, Hao Huang, Lu Wang, Pu Zhao, Yu Kang, Hua Ding, et al. 2024. Nissist: An incident mitigation copilot based on troubleshooting guides. *arXiv preprint arXiv:2402.17531* (2024). https://doi.org/10.48550/arXiv.2402.17531

[7] Karim Buzdar. 2024. Linux dmesg command. https://linuxhint.com/dmesg_tutorial/. Accessed Dec 12, 2024.

[8] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. Continuous incident triage for large-scale online service systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ase.2019.00042

[9] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2020. Towards Intelligent Incident Management: Why We Need It and How We Make It. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1487–1497. https://doi.org/10.1145/3368089.3417055

[10] Junjie Chen, Shu Zhang, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Yu Kang, Feng Gao, Zhangwei Xu, Yingnong Dang, et al. 2020. How incidental are the incidents? characterizing and prioritizing incidents for large-scale online service systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1145/3324884.3416624

[11] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 674–688. https://doi.org/10.1145/3627703.3629553

[12] Weihao Cui, Ji Zhang, Han Zhao, Chao Liu, Wenhao Zhang, Jian Sha, Quan Chen, Bingsheng He, and Minyi Guo. 2025. XPUTimer: Anomaly Diagnostics for Divergent LLM Training in GPU Clusters of Thousand-Plus Scale. *arXiv preprint arXiv:2502.05413* (2025). https://doi.org/10.48550/arXiv.2502.05413

[13] Yangtao Deng, Xiang Shi, Zhuo Jiang, Xingjian Zhang, Lei Zhang, Zhang Zhang, Bo Li, Zuquan Song, Hang Zhu, Gaohong Liu, et al. 2025. Minder: Faulty machine detection for large-scale distributed model training. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 505–521.

[14] Pradeep Dogga, Chetan Bansal, Richard Costleigh, Gopinath Jayagopal, Suman Nath, and Xuchao Zhang. 2023. AutoARTS: Taxonomy, Insights and Tools for Root Cause Labelling of Incidents in Microsoft Azure. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*.

[15] Jianbo Dong, Kun Qian, Pengcheng Zhang, Zhilong Zheng, Liang Chen, Fei Feng, Yichi Xu, Yikai Zhu, Gang Lu, Xue Li, et al. 2025. Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 865–881.

[16] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024). https://doi.org/10.48550/arXiv.2407.21783

[17] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323* (2022). https://doi.org/10.48550/arXiv.2210.17323

[18] Jiaqi Gao, Nofel Yaseen, Robert MacDavid, Felipe Vieira Frujeri, Vincent Liu, Ricardo Bianchini, Ramaswamy Aditya, Xiaohang Wang, Henry Lee, David Maltz, et al. 2020. Scouts: Improving the diagnosis process through domain-customized incident routing. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 253–269. https://doi.org/10.1145/3387514.3405867

[19] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. 2022. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*. https://doi.org/10.1145/3542929.3563482

[20] Pouya Hamadanian, Behnaz Arzani, Sadjad Fouladi, Siva Kesava Reddy Kakarla, Rodrigo Fonseca, Denizcan Billor, Ahmad Cheema, Edet Nkposong, and Ranveer Chandra. 2023. A Holistic View of AI-driven Network Incident Management. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 180–188. https://doi.org/10.1145/

3626111.3628176

[21] Yongqi Han, Qingfeng Du, Ying Huang, Jiaqi Wu, Fulong Tian, and Cheng He. 2024. The potential of one-shot failure root cause analysis: Collaboration of the large language model and small classifier. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 931–943. https://doi.org/10.1145/3691620.3695475

[22] Zilong He, Pengfei Chen, Yu Li, Qiuyu Chen, Hanzhang Li, and Yongfeng Zheng. 2022. Graph Based Incident Extraction and Diagnosis in Large-Scale Online Systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1145/3551349.3556904

[23] Anna Huang et al. 2008. Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, Vol. 4. 9–56.

[24] Junjie Huang, Jinyang Liu, Zhuangbin Chen, Zhihan Jiang, Yichen Li, Jiazhen Gu, Cong Feng, Zengyin Yang, Yongqiang Yang, and Michael R Lyu. 2024. FaultProfIT: Hierarchical Fault Profiling of Incident Tickets in Large-scale Cloud Systems. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. https://doi.org/10.1145/3639477.3639754

[25] Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, et al. 2024. Xpert: Empowering incident management with query recommendations via large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. https://doi.org/10.1145/3597503.3639081

[26] Zhihan Jiang, Junjie Huang, Guangba Yu, Zhuangbin Chen, Yichen Li, Renyi Zhong, Cong Feng, Yongqiang Yang, Zengyin Yang, and Michael Lyu. 2025. L4: Diagnosing large-scale LLM training failures via automated log analysis. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 51–63. https://doi.org/10.1145/3696630.3728531

[27] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[28] Pengxiang Jin, Shenglin Zhang, Minghua Ma, Haozhe Li, Yu Kang, Liqun Li, Yudong Liu, Bo Qiao, Chaoyun Zhang, Pu Zhao, et al. 2023. Assess and summarize: Improve outage understanding with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. https://doi.org/10.1145/3611643.3613891

[29] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, Lionel Briand, and Michael R Lyu. 2023. Exploring the effectiveness of LLMs in automated logging generation: An empirical study. *arXiv preprint arXiv:2307.05950* (2023). https://doi.org/10.48550/arXiv.2307.05950

[30] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. *arXiv preprint arXiv:2405.19888* (2024). https://doi.org/10.48550/arXiv.2405.19888

[31] Jerry Liu. 2024. LlamaIndex. https://github.com/run-llama/llama_index. Accessed Dec 12, 2024.

[32] Yuhe Liu, Changhua Pei, Longlong Xu, Bohan Chen, Mingze Sun, Zhirui Zhang, Yongqian Sun, Shenglin Zhang, Kun Wang, Haiming Zhang, Jianhui Li, Gaogang Xie, Xidao Wen, Xiaohui Nie, Minghua Ma, and Dan Pei. 2025. OpsEval: A Comprehensive Benchmark Suite for Evaluating Large Language Models' Capability in IT Operations Domain. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 503–513. https://doi.org/10.1145/3696630.3728572

[33] Meta. 2024. Meta Llama 3.1 70B Instruct. https://huggingface.co/neuralmagic/Meta-Llama-3.1-70B-Instruct-FP8. Accessed Dec 6, 2024.

[34] Meta. 2024. Meta Llama 3.1 8B Instruct. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct. Accessed Dec 6, 2024.

[35] Microsoft Azure. 2024. Azure OpenAI Service. https://azure.microsoft.com/en-us/products/ai-services/openai-service. Accessed Nov 3, 2024.

[36] Microsoft Azure. 2024. Azure OpenAI Service pricing. https://azure.microsoft.com/en-us/pricing/details/cognitive-services/openai-service/#pricing. Accessed Oct 12, 2024.

[37] Microsoft Azure. 2024. AzureHPC Node Health Check. https://github.com/Azure/azurehpc-health-checks. Accessed Dec 12, 2024.

[38] Microsoft Azure. 2024. GPT-4o and GPT-4 Turbo. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models?tabs=global-standard%2Cstandard-chat-completions#gpt-4o-and-gpt-4-turbo. Accessed Dec 12, 2024.

[39] Microsoft Azure. 2024. o1 and o1-mini models. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models?tabs=global-standard%2Cstandard-chat-completions#o1-and-o1-mini-models-limited-access. Accessed Dec 12, 2024.

[40] Microsoft Azure. 2025. Linux Virtual Machines Pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/#pricing. Accessed May 3, 2025.

[41] NVIDIA. 2024. NVIDIA Data Center GPU Manager. https://github.com/NVIDIA/DCGM. Accessed Dec 12, 2024.
[42] NVIDIA. 2024. NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/. Accessed Dec 12, 2024.
[43] NVIDIA. 2024. NVIDIA NCCL Tests. https://github.com/NVIDIA/nccl-tests. Accessed Dec 12, 2024.
[44] NVIDIA. 2024. NVIDIA NVLink and NVLink Switch. https://www.nvidia.com/en-us/data-center/nvlink/. Accessed Dec 12, 2024.
[45] NVIDIA. 2024. The NVIDIA Quantum InfiniBand Platform. https://www.nvidia.com/en-us/networking/products/infiniband/. Accessed Dec 12, 2024.
[46] Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. ART: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014* (2023). https://doi.org/10.48550/arXiv.2303.09014
[47] Changhua Pei, Zhihan Wang, Fei Liu, Zhanhao Li, Yongqian Liu, Xin He, and Dan Pei. 2025. Flow-of-Action: SOP Enhanced LLM-Based Multi-Agent System for Root Cause Analysis. In *Companion Proceedings of the ACM on Web Conference 2025*. 422–431. https://doi.org/10.1145/3701716.3715225
[48] Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Le Yan, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, et al. 2023. Large language models are effective text rankers with pairwise ranking prompting. *arXiv preprint arXiv:2306.17563* (2023). https://doi.org/10.48550/arXiv.2306.17563
[49] Qwen. 2024. Qwen 2.5 32B Instruct. https://huggingface.co/Qwen/Qwen2.5-32B-Instruct-GPTQ-Int4. Accessed Dec 6, 2024.
[50] Qwen. 2024. Qwen 2.5 72B Instruct. https://huggingface.co/Qwen/Qwen2.5-72B-Instruct-GPTQ-Int4. Accessed Dec 6, 2024.
[51] Qwen. 2024. Qwen 2.5 7B Instruct. https://huggingface.co/Qwen/Qwen2.5-7B-Instruct-GPTQ-Int4. Accessed Dec 6, 2024.
[52] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring llm-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. https://doi.org/10.1145/3663529.3663841
[53] Kunal Sawarkar, Abhilasha Mangal, and Shivam Raj Solanki. 2024. Blended RAG: Improving RAG (Retriever-Augmented Generation) Accuracy with Semantic Search and Hybrid Query-Based Retrievers. *arXiv preprint arXiv:2404.07220* (2024). https://doi.org/10.48550/arXiv.2404.07220
[54] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2023). https://doi.org/10.52202/075280-2997
[55] Shiwen Shan, Yintong Huo, Yuxin Su, Yichen Li, Dan Li, and Zibin Zheng. 2024. 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*. https://doi.org/10.1145/3650212.3652106
[56] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, Nachiappan Nagappan, and Thomas Zimmermann. 2021. Neural knowledge extraction from cloud service incidents. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 218–227. https://doi.org/10.1109/icse-seip52600.2021.00031
[57] Manish Shetty, Chetan Bansal, Sai Pramod Upadhyayula, Arjun Radhakrishna, and Anurag Gupta. 2022. AutoTSG: learning and synthesis for incident troubleshooting. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1477–1488. https://doi.org/10.1145/3540250.3558958
[58] Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. 2024. Teola: Towards end-to-end optimization of LLM-based applications. *arXiv preprint arXiv:2407.00326* (2024). https://doi.org/10.48550/arXiv.2407.00326
[59] Haopei Wang, Anubhavnidhi Abhashkumar, Changyu Lin, Tianrong Zhang, Xiaoming Gu, Ning Ma, Chang Wu, Songlin Liu, Wei Zhou, Yongbin Dong, et al. 2024. NetAssistant: Dialogue Based Network Diagnosis in Data Center Networks. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*.
[60] Xiaohua Wang, Zhenghua Wang, Xuan Gao, Feiran Zhang, Yixin Wu, Zhibo Xu, Tianyuan Shi, Zhengyuan Wang, Shizheng Li, Qi Qian, et al. 2024. Searching for best practices in retrieval-augmented generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. https://doi.org/10.18653/v1/2024.emnlp-main.981
[61] Zehao Wang. 2025. Identifying Performance-Sensitive Configurations in Software Systems with LLM-Driven Agents. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 222–223. https://doi.org/10.1109/icse-companion66252.2025.00069
[62] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. 2023. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*. https://doi.org/10.1145/3600006.3613145

[63] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2024. 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*. 4966–4974. https://doi.org/10.1145/3627673.3680016

[64] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. 2024. FALCON: Pinpointing and Mitigating Stragglers for Large-Scale Hybrid-Parallel Training. *arXiv preprint arXiv:2410.12588* (2024). https://doi.org/10.48550/arXiv.2410.12588

[65] Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-Pack: Packaged Resources To Advance General Chinese Embedding. https://doi.org/10.48550/arXiv.2309.07597 arXiv:2309.07597

[66] Zhiqiang Xie, Yujia Zheng, Lizi Ottens, Kun Zhang, Christos Kozyrakis, and Jonathan Mace. 2024. Cloud Atlas: Efficient Fault Localization for Cloud Systems using Language Models and Causal Insight. *arXiv preprint arXiv:2407.08694* (2024). https://doi.org/10.48550/arXiv.2407.08694

[67] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, et al. 2024. SuperBench: Improving Cloud AI Infrastructure Reliability with Proactive Validation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*.

[68] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115* (2024). https://doi.org/10.48550/arXiv.2412.15115

[69] Fangkai Yang, Wenjie Yin, Lu Wang, Tianci Li, Pu Zhao, Bo Liu, Paul Wang, Bo Qiao, Yudong Liu, Mårten Björkman, et al. 2023. Diffusion-based time series data imputation for cloud failure prediction at Microsoft 365. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2050–2055. https://doi.org/10.1145/3611643.3613866

[70] Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. 2024. GPT4Tools: Teaching large language model to use tools via self-instruction. *Advances in Neural Information Processing Systems* 36 (2024). https://doi.org/10.52202/075280-3149

[71] Dylan Zhang, Xuchao Zhang, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2023. PACE: Prompting and augmentation for calibrated confidence estimation with GPT-4 in cloud incident root cause analysis. *arXiv preprint arXiv:2309.05833* (2023). https://doi.org/10.48550/arXiv.2309.05833

[72] Dylan Zhang, Xuchao Zhang, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. LM-PACE: Confidence estimation by large language models for effective root causing of cloud incidents. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 388–398. https://doi.org/10.1145/3663529.3663858

[73] Lingzhe Zhang, Yunpeng Zhai, Tong Jia, Xiaosong Huang, Chiming Duan, and Ying Li. 2025. AgentFM: Role-aware failure management for distributed databases with LLM-driven multi-agents. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 525–529. https://doi.org/10.1145/3696630.3728492

[74] Xuchao Zhang, Supriyo Ghosh, Chetan Bansal, Rujia Wang, Minghua Ma, Yu Kang, and Saravan Rajmohan. 2024. 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*. https://doi.org/10.1145/3663529.3663846

[75] Nengwen Zhao, Junjie Chen, Zhou Wang, Xiao Peng, Gang Wang, Yong Wu, Fang Zhou, Zhen Feng, Xiaohui Nie, Wenchi Zhang, et al. 2020. Real-time incident prediction for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 315–326. https://doi.org/10.1145/3368089.3409672

[76] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023. Efficiently programming large language models using SGLang. *arXiv e-prints* (2023), arXiv–2312. https://doi.org/10.48550/arXiv.2312.07104

Received 2025-09-05; accepted 2025-12-22