IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510145

KubeVision AI: Kubernetes-Native Observability and Root Cause Analysis Platform

Shubham More[1], Prajwal Mandlecha[2], Kaustubh Gharat[3], Kaushal Waykole[4], Prof. P.A.Jain[5]

Department of Computer Engineering, Pune Institute of Computer Technology,[1] Dhankawadi, Pune, 411043, Maharashtra, India[1-5]

Abstract: Managing containerized workloads on Kubernetes simplifies deployment and horizontal scaling, but diagnosing failures in production remains challenging. When a service experiences an issue, errors often propagate across dependent components, triggering waves of correlated alerts that conceal the underlying fault. Conventional Site Reliability Engineering (SRE) workflows depend on bytecode instrumentation agents or sidecar proxies, which introduce runtime latency and memory overhead. At the same time, automated Root Cause Analysis (RCA) techniques based on static thresholds or causal dependency graphs struggle under high-dimensional metric noise and bursty traffic patterns. While Large Language Models (LLMs) offer strong contextual reasoning, single-prompt implementations suffer from hallucinations, context window exhaustion, and alert flooding.

This survey paper reviews current methodologies across cloud-native observability, anomaly detection, and automated root cause analysis. We evaluate telemetry collection mechanisms (bytecode injection, sidecars, and kernel-level eBPF), time-series anomaly detection pipelines, causal dependency discovery, and emerging LLM-based agentic frameworks. We benchmark leading industry platforms (Dynatrace, Datadog, Pixie, K8sGPT) alongside academic systems (CauseInfer, MicroCause, CloudRanger, RCAAgent, TriFleetRCA). To address key research gaps around instrumentation overhead, alert storms, and ungrounded LLM outputs, we propose the architectural design of **KubeVision AI**: a Kubernetes-native framework combining kernel-level eBPF telemetry, a two-stage anomaly filter (Isolation Forest paired with an LSTM Autoencoder), an alert deduplication gateway, and a multi-agent LangGraph diagnostic pipeline with programmatic reflection gates. Finally, we discuss open research challenges and position time-series capacity forecasting as a planned extension.

Keywords: Kubernetes Observability, Extended Berkeley Packet Filter (eBPF), Root Cause Analysis (RCA), Multi-Agent Systems, Large Language Models (LLMs), Anomaly Detection, AIOps.

INTRODUCTION

Kubernetes has become the primary platform for deploying containerized workloads at scale. Its declarative API and automated scheduling allow engineering teams to manage independent release cycles and dynamic scaling policies. However, shifting from monolithic architectures to distributed topologies increases operational complexity. In production clusters, hundreds of services often run across thousands of ephemeral pods, collectively producing millions of telemetry events every second [1].

Under these conditions, runtime failures rarely remain confined to their origin. A slow database query, thread contention, or connection pool exhaustion in a downstream pod can quickly stall upstream services, producing cascading timeouts and HTTP 5xx errors across client-facing endpoints. During such incidents, monitoring systems frequently emit hundreds of simultaneous notifications—an effect commonly known as an alert storm [16]. Site Reliability Engineers (SREs) are forced to manually correlate disconnected dashboards across metrics, logs, and traces while under severe time pressure, leading to alert fatigue and prolonged Mean Time to Resolution (MTTR).

A. *Operational Bottlenecks in Existing Approaches*

Contemporary observability and incident triage solutions exhibit four core shortcomings:

1. **Instrumentation Overhead:** Traditional Application Performance Monitoring (APM) tools require modifying application source code, injecting runtime bytecode agents, or deploying sidecar proxies like Envoy. Elkhatib and Povedano Poyato [15] empirically evaluated service mesh sidecars in constrained environments, showing that Istio consumed 20% to 43% more memory, incurred 6% to 12% higher latency, and exhibited roughly 1.2× higher CPU usage compared to lighter alternatives like Linkerd, while both frameworks imposed significant overhead compared to baseline Kubernetes pods without proxies.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510145

2. **Telemetry Duplication and Storage Costs:** Many monitoring setups replicate high-frequency metrics, container logs, and trace spans into monolithic, centralized data stores. This practice drives up infrastructure bills without necessarily improving cross-modal correlation during incident triage.
3. **False Positives from Bursty Workloads:** Cloud-native applications naturally experience benign traffic spikes, including periodic batch routines, garbage collection cycles, and pod auto-scaling events. Single-stage thresholding and simple statistical detectors frequently flag these transient variations as severe outages, desensitizing on-call teams through false alarms.
4. **Context Exhaustion and LLM Hallucinations:** While Large Language Models can synthesize unstructured diagnostics effectively, dumping raw alert streams directly into single-agent prompts leads to context window overflow, high API costs, and unsupported or fabricated conclusions [2].

B. *Contributions and Paper Outline*

This survey reviews current observability and RCA methodologies, assesses existing commercial and academic tools, and presents a proposed architectural framework to address these challenges. The primary contributions include:

- A comparative analysis of telemetry acquisition mechanisms, contrasting bytecode agents, sidecars, and kernel-level eBPF probes.
- A technical review of anomaly detection algorithms and RCA approaches, including causal graph discovery, distributed tracing, and LLM-agentic frameworks.
- A structured taxonomy benchmarking state-of-the-art systems across key operational and architectural dimensions.
- The architectural formulation of **KubeVision AI**, a proposed Kubernetes-native framework combining zero-instrumentation eBPF telemetry, a two-stage anomaly filter with confidence gating, an alert-storm mitigation gateway, and a reflection-gated multi-agent LangGraph RCA runtime.
- An overview of open research challenges, identifying time-series capacity forecasting as a planned future direction.

LITERATURE SURVEY

We examine the existing literature across four foundational areas that define modern cloud-native observability and root cause analysis.

A. *Traditional APM and Runtime Instrumentation*

Enterprise Application Performance Monitoring (APM) platforms such as Dynatrace, Datadog, and New Relic have traditionally relied on language-specific bytecode injection to inspect internal application states. In this model, an agent is attached to the application runtime—such as a Java agent modifying class bytecode at class-loading time to measure method execution times and intercept unhandled exceptions. Although bytecode injection provides detailed call-stack profiles and transaction traces, it creates practical operational challenges: separate agents must be maintained and verified for each programming language (Java, Node.js, Python, Go, .NET), agent updates risk runtime regressions, and the instrumentation logic itself consumes CPU cycles and memory within the application container.

An alternative approach emerged with service mesh architectures such as Istio and Linkerd. These frameworks deploy a dedicated proxy container (typically Envoy) alongside every application pod, using iptables rules to intercept inbound and outbound network traffic transparently. This avoids the need for language-specific agents, since the proxy extracts Layer 7 protocol metadata (HTTP headers, gRPC status codes, request latencies) at the network perimeter. However, running a proxy beside every application container incurs measurable resource penalties. Elkhatib and Povedano Poyato [15] evaluated service mesh frameworks in resource-constrained Kubernetes environments, finding that Istio increased memory consumption by 20% to 43% and added 6% to 12% to request latency compared to Linkerd, while also requiring approximately 1.2× more CPU resources. In large clusters containing hundreds of microservices, running dedicated proxies in every pod can accumulate enough overhead to occupy the equivalent of several worker nodes.

B. *Kernel-Level eBPF Telemetry*

To avoid the overhead of bytecode injection and sidecar proxies, recent observability research has focused on the Extended Berkeley Packet Filter (eBPF) [10, 9]. eBPF allows custom, sandboxed programs to execute directly within the Linux kernel without changing kernel source code or loading external kernel modules. The in-kernel bytecode verifier checks each program before execution to ensure memory safety, termination, and absence of unbounded loops. These programs attach to kernel tracepoints, system calls (sys_enter_connect, sys_enter_write, sys_enter_read), and socket buffers, observing network and system activity across all processes on the host. Because observation occurs at the kernel boundary, telemetry can be gathered without modifying application binaries, container images, or pod deployment specifications.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510145

Cassagnes et al. [10] evaluated eBPF for non-intrusive performance monitoring of containerized workloads, demonstrating that kernel-level probes introduce minimal runtime overhead compared to user-space profilers. Soldani et al. [9] surveyed eBPF applications across cloud-native networking, security, and observability, highlighting its utility as a foundational layer for non-invasive system tracking.

Modern open-source tools such as Pixie [14] and Grafana Beyla [17] demonstrate the practical potential of this approach. Deployed as a single DaemonSet per Kubernetes node, Beyla inspects socket events to automatically extract Layer 7 performance metrics—including HTTP, gRPC, Redis, Kafka, and SQL query latencies—directly from kernel socket buffers. This shifts telemetry collection from inside the application runtime down to the kernel layer, providing uniform golden signals (latency percentiles, error rates, throughput) across polyglot workloads without code changes.

However, existing eBPF observability tools focus primarily on data collection and visualization. While they provide detailed protocol-level telemetry, they do not incorporate automated anomaly detection, alert correlation, or root cause diagnosis. Consequently, an operational gap remains between capturing kernel telemetry and determining why a failure occurred.

C. *Causal Graph and Statistical RCA*

Automated fault localization in distributed systems has historically relied on two primary analytical techniques: causal graph discovery from time-series metrics and critical path analysis from distributed traces.

CauseInfer, introduced by Chen et al. [3], applies the Peter-Clark (PC) algorithm to construct hierarchical causality graphs from system metrics. The PC algorithm begins with a fully connected undirected graph and systematically performs conditional independence tests between pairs of metric time-series. If two metrics are found to be conditionally independent given a subset of other metrics, the connecting edge is removed. CauseInfer organizes this analysis into a two-level hierarchy: a service-level dependency graph that identifies inter-service propagation paths, and a localized metric-level causal graph that isolates the failing resource (such as CPU saturation, disk I/O bottlenecks, or network packet drops). For sparse graphs with maximum in-degree bounded by k, the PC algorithm requires $O(|V|^k)$ independence tests, but in general dense topologies, it scales exponentially as $O(|V|^2\, 2^{|V|-2})$, making real-time computation prohibitive for large microservice clusters with hundreds of interacting components.

MicroCause, developed by Meng et al. [4], addressed the temporal delays inherent in distributed failure propagation. Rather than relying on standard static graphs, MicroCause employs the PCMCI algorithm—combining the PC algorithm's skeleton discovery phase with Momentary Conditional Independence (MCI) testing to identify time-lagged causal dependencies across metrics. This accounts for scenarios where a failure in one component (e.g., database connection pool exhaustion at t=0) takes several seconds to manifest as an API gateway timeout downstream. After generating the dynamic causal graph, MicroCause uses a Temporal Cause-Oriented Random Walk (TCORW) to rank candidate root causes. When evaluated against historical failure tickets from a commercial online retail platform, MicroCause achieved a top-5 accuracy (AC@5) of 98.7%, improving over previous baseline methods by 33.4%.

CloudRanger, proposed by Wang et al. [5], eliminates the requirement for predefined topology maps by dynamically deriving impact graphs from telemetry streams. The system uses second-order Granger causality tests to evaluate whether historical variations in one metric improve predictions of another, followed by a second-order random walk to identify culprit services. In empirical evaluations using production incident data from IBM Bluemix, CloudRanger achieved a 10% accuracy improvement over prior fault localization techniques.

Despite their algorithmic grounding, causal graph approaches present several practical challenges in cloud environments:

- **Computational Scalability:** Running conditional independence tests across high-dimensional metric streams is computationally demanding, especially as cluster size expands.
- **Historical Baseline Requirements:** Accurate graph construction generally requires extended observation windows under steady-state conditions, which are difficult to maintain in dynamic clusters with continuous deployments.
- **Absence of Qualitative Context:** Causal models output abstract numeric scores or ranked node lists rather than interpretable diagnostic narratives explaining why a failure occurred.
- **Omission of Unstructured Signals:** Purely metric-driven causal graphs do not incorporate container error logs, stack traces, or Kubernetes lifecycle events (OOMKilled, CrashLoopBackOff), which often contain the most direct evidence of the root cause.

D. *Generative AI and Agentic Incident Triage*

Recent work in AIOps has explored Large Language Models (LLMs) to bridge the gap between quantitative telemetry and human-readable incident explanations.

K8sGPT [13] applies LLMs to Kubernetes control-plane diagnostics. It scans cluster resources via the Kubernetes API server, identifies failing objects (such as pods in CrashLoopBackOff, failing readiness probes, or misconfigured Services), and submits the error descriptions to an LLM to generate plain-English explanations and remediation steps.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510145

While helpful for misconfigurations and resource quota issues, K8sGPT only observes Kubernetes object metadata. It has no visibility into application-level latency spikes, database query slowdowns, or distributed traces, limiting its ability to diagnose runtime performance degradations.

RCAgent, developed by Wang et al. [2], implements an autonomous, tool-augmented diagnostic agent evaluated on Alibaba Cloud's Real-time Compute Platform for Apache Flink. RCAgent uses the ReAct (Reasoning and Acting) framework [11], where the LLM alternates between generating diagnostic reasoning steps and executing tool calls—such as querying Prometheus metrics, fetching container logs, and checking cloud resource configurations. By iteratively gathering evidence and assessing causal hypotheses, RCAgent automates multi-step troubleshooting. However, relying on a single agent creates a practical bottleneck during complex outages: a single context window must absorb metric outputs, log snippets, and trace details simultaneously, which quickly leads to context limits, lost details, and degraded diagnostic accuracy.

TriFleetRCA, introduced by Patel et al. [12], evaluated an on-premise multi-agent RCA architecture designed to operate without sending proprietary logs to external cloud APIs. Using the quantized open-source model Qwen2.5-14B-Instruct hosted on a single GPU node via vLLM, the system analyzes diagnostic evidence across pod, namespace, and cluster scopes. It incorporates template deduplication and BM25 ranking to index logs, alongside an ingest guard to protect against adversarial prompt injection. While demonstrating that localized LLM inference is feasible for Kubernetes diagnostics, TriFleetRCA remains focused on log analysis and cluster state, without incorporating eBPF kernel telemetry or distributed tracing.

METHODOLOGIES IN CLOUD-NATIVE OBSERVABILITY AND RCA

This section reviews the core technical methodologies used in modern cloud-native observability and automated incident triage.

A. *Multi-Modal Telemetry and Polyglot Persistence*

Troubleshooting distributed systems requires correlating data across four distinct telemetry modalities:

1. **Time-Series Metrics:** Numeric values representing resource consumption (CPU, memory), traffic rates, and Layer 7 golden signals (latency percentiles p50/p95/p99, error rates) scraped at standard intervals (typically 15 to 30 seconds) by Prometheus and queried using PromQL. Metrics are effective for identifying when an anomaly starts and quantifying its magnitude.
2. **Structured Container Logs:** Text emitted by application processes to stdout/stderr, captured and tagged with Kubernetes pod metadata by log shippers (such as Fluent Bit) and stored in Grafana Loki. Queried via LogQL, logs provide the specific error messages, stack traces, and application exceptions needed to understand what went wrong.
3. **Distributed Request Traces:** End-to-end transaction flows represented as spans across microservices, stored in Grafana Tempo and queried using TraceQL. Traces reveal the call sequence and latency accumulation along the critical path, answering where in the architecture delays or errors originated.
4. **Kubernetes Lifecycle Events:** Cluster status changes recorded by the Kubernetes API server and exported by kube-state-metrics—including pod restarts, OOMKilled signals, node disk pressure, and failed health checks. These signals capture infrastructure context that application-level telemetry alone cannot explain.

Storing all of these telemetry streams in a single generic database is rarely cost-effective at scale. Metrics, structured logs, and distributed traces have fundamentally different access patterns, compression profiles, and index structures. A more practical design preserves each data type within its specialized engine (Prometheus for time series, Loki for chunked logs, Tempo for object-stored traces) while storing only lightweight pointers—such as trace IDs, timestamp windows, and pod labels—inside incident records. This avoids duplicate writes while keeping diagnostic references intact.

B. *Two-Stage Anomaly Detection and Confidence Gating*

Application workloads in containerized environments frequently exhibit bursty traffic patterns. Routine operations, including scheduled jobs, garbage collection, autoscaling events, and rolling updates, can produce temporary metric deviations that may be misclassified as anomalies by simple threshold-based detectors. To reduce false-positive alerts, the proposed pipeline combines statistical anomaly screening with temporal verification using a sequence reconstruction model.

1. **Stage 1 — Statistical Screening (Isolation Forest):** Isolation Forest [6] is an unsupervised anomaly detection algorithm that identifies observations that can be isolated using relatively short paths in an ensemble of randomly partitioned trees. Its computational cost depends on the number of trees and the subsample size, enabling efficient anomaly detection through subsampling.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510145

The anomaly score is defined as:

$$s(x,n)=2^{\frac{-E[h(x)]}{c(n)}}$$

$\mathbb{E}[h(x)]$ denotes the expected path length of observation x across the isolation trees, and $c(n)$ is the normalization factor corresponding to the average path length of an unsuccessful search in a binary search tree:

$$c(n)=2(ln(n\text{-}1)+\gamma)-\frac{2(n\text{-}1)}{n}$$

where $\gamma \approx 0.5772156649$ is the Euler–Mascheroni constant. This approximation is used for sufficiently large subsamples; small subsample sizes require appropriate edge-case handling.
In the original scoring formulation, values below 0.5 generally indicate normal observations, whereas values approaching 1 indicate increasingly anomalous observations. These interpretations provide a reference rather than universal operational thresholds.

The screening stage employs two empirically calibrated thresholds, τ_{screen} and τ_{high}, with $\tau_{screen} < \tau_{high}$:

– *Low-score region ($s \leq \tau_{screen}$):* Observations are provisionally classified as nominal and are not forwarded to the temporal verification stage, subject to the system's independent safety and monitoring rules.

– *High-severity fast-track ($s > \tau_{high}$):* Observations exceeding the high-severity threshold are routed directly to the incident gateway to reduce detection latency. This decision is subject to independent validation of the threshold and its false-positive rate.

– *Temporal verification band ($\tau_{screen} < s \leq \tau_{high}$):* Observations with intermediate anomaly scores are forwarded to Stage 2 for temporal verification.

Threshold values, potentially including $\tau_{screen} \in [0.55,0.60]$ and $\tau_{high} \in [0.75,0.85]$, are treated as illustrative initial settings rather than established defaults. They must be calibrated and evaluated using representative validation data, preferably including labeled anomalies or controlled fault injection. ROC analysis can be used when ground-truth labels are available; precision–recall analysis may additionally be useful when anomalies are rare.

2. **Stage 2 — Temporal Verification (LSTM Autoencoder):** Observations forwarded from Stage 1 are evaluated using an LSTM Autoencoder [7], trained on representative normal operating sequences. The encoder maps a rolling time-series window of length T into a latent representation, and the decoder reconstructs the input sequence.
For a normalized multivariate observation at time step t, the reconstruction error is calculated as:

$$e_t=\|x_t - \hat{x}_t\|_2$$

where x_t and $\hat{x}_t$ denote the observed and reconstructed metric vectors, respectively. Feature normalization parameters are estimated from the training data to prevent metrics with larger numerical scales from dominating the reconstruction error.
The reconstruction error is standardized relative to its distribution on a held-out normal validation set:

$$z_t = \frac{e_t - \mu_e}{\sigma_e}$$

where μ_e and σ_e denote the mean and standard deviation of the validation reconstruction errors. The resulting score is interpreted as a standardized deviation, not as an anomaly probability. If the error distribution is substantially skewed or heavy-tailed, a robust normalization method or an empirically calibrated error quantile may be more appropriate.

A temporal persistence rule is applied to reduce alerts caused by isolated reconstruction-error spikes. Specifically, an anomaly is confirmed when the selected error threshold is exceeded in at least w consecutive evaluation windows, where $w \geq 3$ is an empirically selected design parameter. The window length, overlap, threshold, and persistence requirement are calibrated against representative normal workloads and injected or labeled anomalies. This mechanism is intended to suppress transient deviations while retaining sensitivity to sustained degradation.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510145

C. *Multi-Agent RCA and Reflection Gates*

Single-agent LLM systems often struggle with context limits when diagnosing complex microservice incidents. A supervisor-driven multi-agent framework addresses this by dividing diagnostic work across specialized agents coordinated by an orchestrator:

- An **Orchestrator Agent** receives an alert event and assigns targeted diagnostic tasks to specialized subagents operating in parallel:
  - *Metrics Agent:* Formulates and executes PromQL queries against Prometheus to check for golden signal deviations (latency increases, elevated error rates, drop in request throughput).
  - *Log Agent:* Queries Loki using LogQL to find error messages, stack traces, and unexpected log bursts from affected containers within the incident window.
  - *Network Agent:* Queries Tempo using TraceQL to trace critical-path latencies and pinpoint failing downstream dependencies.
  - *K8s Event Agent:* Queries the Kubernetes API server for pod lifecycle signals (OOMKilled, failed liveness probes, node pressure, eviction events).
- A programmatic **Reflection Gate** checks whether the collected evidence is sufficient before any diagnosis is drafted. Rather than relying on an LLM to judge completeness, the gate applies a deterministic rule requiring corroboration from at least three of the four telemetry sources (a majority quorum). If evidence is incomplete (for instance, if metrics indicate high latency but logs and traces return no corresponding errors), the Orchestrator directs the relevant subagents to adjust their queries—expanding the time window, checking adjacent pods, or relaxing trace filtering criteria.
- A **Self-Validation Critic** reviews the resulting diagnostic report against the raw query outputs and log snippets collected during the investigation. Each assertion in the proposed summary (such as identifying database connection exhaustion as the root cause) must link back to specific query results or log records. Any unsupported statements are pruned before the report is delivered to the engineering team, reducing the risk of hallucinated explanations.

COMPARATIVE EVALUATION OF EXISTING SYSTEMS

Table 1 compares existing commercial platforms, open-source utilities, and academic frameworks alongside the proposed KubeVision AI architecture across key operational dimensions.

TABLE I COMPARATIVE EVALUATION: EXISTING OBSERVABILITY AND RCA SYSTEMS VS. PROPOSED KUBEVISION AI

System / Tool	Telemetry Collection	Anomaly Detection / SLO Handling	Alert-Storm Gating	RCA Engine / Methodology	Deployment Footprint
Dynatrace Davis AI	Proprietary Bytecode Agents & Sidecars	Deterministic thresholds + Baselining	Alert grouping by service	Dependency Graph Causality	Heavyweight, high cost, always-on pods
Datadog Watchdog	Host Agent + Sidecars + App Tracing	Algorithmic Seasonal Decomposition	Event aggregation by tag	Statistical correlation across tags	High cloud cost, vendor lock-in, always-on
Pixie (CNCF)	Kernel eBPF (Zero-Code DaemonSet)	None (Manual PromQL / PxL queries)	None	Manual script-driven investigation (PxL)	In-cluster eBPF, no scale-to-zero agents
K8sGPT (CNCF)	Kubernetes API Server Scrapers	Static analyzers on pod error states	None	Single LLM query per resource failure	Lightweight CLI / Operator, no eBPF / traces
CauseInfer [3]	Proxy / Sidecar Tracing	Statistical anomaly thresholding	None	Causal DAG via PC Algorithm	High offline compute $O(\|V\|^k$ to exponential
MicroCause [4]	APM Scrapers / Custom SDKs	Anomaly score thresholding	None	Dynamic Causal Graph + PCMCI Random Walk	High compute cost, no natural text reports

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510145

System / Tool	Telemetry Collection	Anomaly Detection / SLO Handling	Alert-Storm Gating	RCA Engine / Methodology	Deployment Footprint
RCAgent [2]	Cloud Monitoring APIs / Prometheus	Single-stage anomaly detector	None	Tool-augmented autonomous ReAct agent	Single-agent bottleneck, public LLM API risk
TriFleetRCA [12]	Cluster Logs & K8s State	Template frequency clustering	BM25 log deduplication	Multi-agent on-premise LLM pipeline	Single GPU node, no eBPF or trace support
Proposed System (KubeVision AI)	Kernel eBPF (Beyla) + Node Logs (Fluent Bit)	Configurable SLO evaluation and Two-stage ML pipeline: Isolation Forest → LSTM-AutoEncoder	Go Gateway: Validate, Dedupe, Rate limiting	LangGraph multi-agent system: 4 specialized agents + Reflection Gate	Single Go binary + KEDA scale-to-zero workers

A. *Key Takeaways from the Comparison*

Our comparative analysis highlights three practical tradeoffs in existing systems:

1. **Visibility vs. Overhead:** Systems that offer detailed application diagnostics often require intrusive bytecode injection or sidecar proxies that increase resource consumption (Dynatrace, MicroCause). Conversely, lightweight kernel-based tools like Pixie provide zero-overhead telemetry but lack automated root cause intelligence.
2. **Vulnerability to Alert Flooding:** Most academic RCA frameworks assume clean, pre-filtered input events. In actual production outages, a single fault can generate hundreds of simultaneous notifications, overwhelming LLM context windows and incurring high API costs unless an alert-filtering gateway is present.
3. **Explainability vs. Verification:** Purely statistical causal graphs produce numerical rankings that require manual interpretation by engineers, while unconstrained LLMs can produce fluent, natural-language explanations that contain hallucinated claims. Effective automated diagnosis requires combining multi-agent reasoning with verifiable evidence checks.

PROPOSED SOLUTION: KUBEVISION AI

To address these limitations, we propose the architectural design of **KubeVision AI**, a Kubernetes-native Observability and Root Cause Analysis platform. KubeVision AI is designed around a central operational goal: **minimizing runtime intrusion while delivering evidence-grounded incident diagnoses**. The system requires no modifications to application code, deploys without sidecar proxies, and operates across standard Kubernetes distributions and Container Network Interfaces (CNIs).

A. *Architectural Diagram and Component Layers*

As shown in Figure 1, KubeVision AI comprises five logical architectural layers: zero-intrusion telemetry collection, multi-modal telemetry storage, parallel anomaly evaluation through the ML and SLO paths, centralized alert management, multi-agent root cause analysis. The ML and SLO paths operate independently and converge at the Go Hub Gateway, enabling confirmed SLO violations to bypass ML processing while retaining the existing downstream incident-management workflow.

 This work is licensed under a Creative Commons Attribution 4.0 International License

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510145

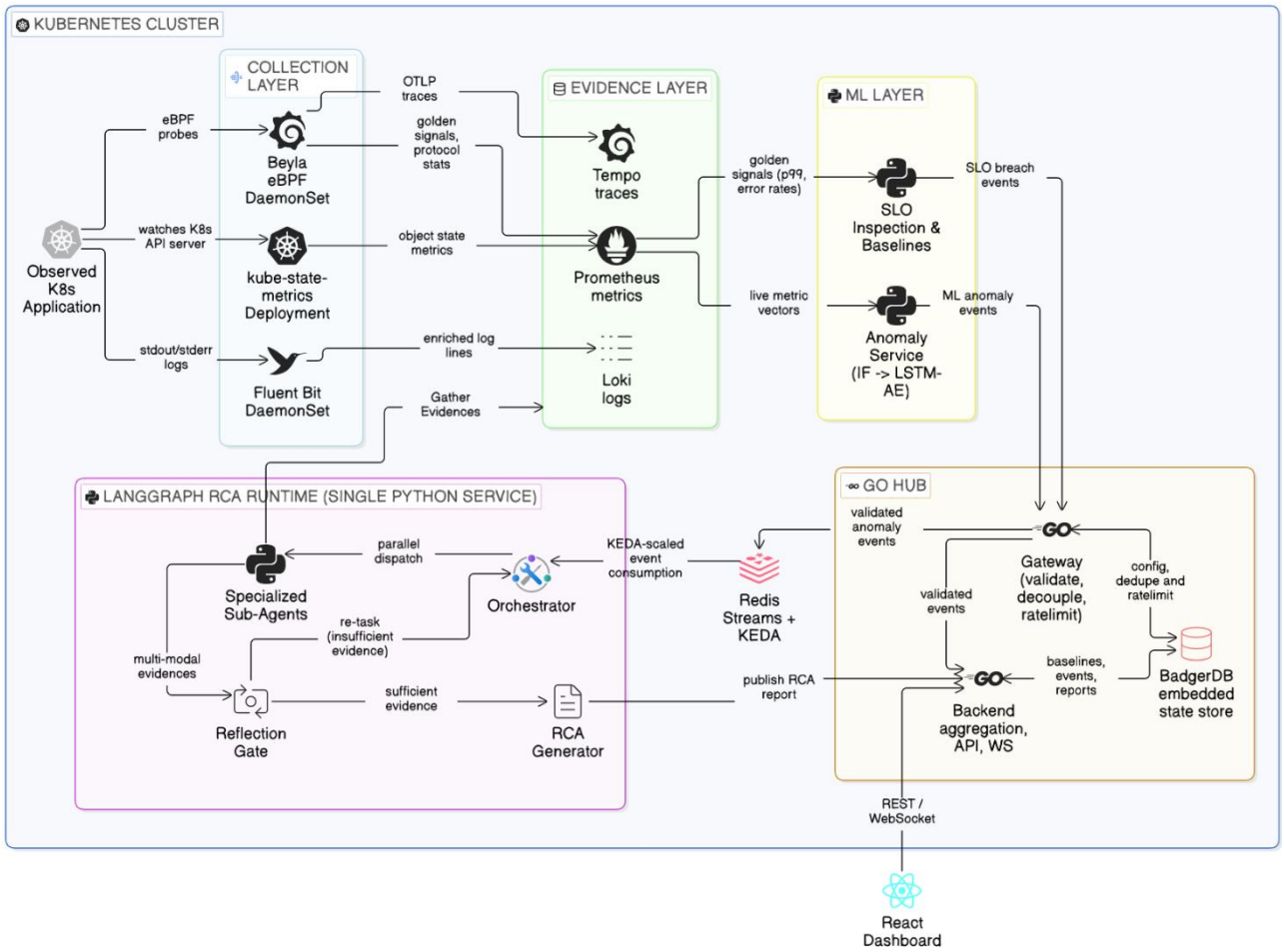


Fig. 1 System Architecture and Telemetry Dataflow of the Proposed KubeVision AI Platform.

B. *Zero-Intrusion Collection and Pointer Referencing*

KubeVision AI uses Grafana Beyla deployed as a node-level DaemonSet with host PID and eBPF capabilities to extract Layer 7 protocol metrics (HTTP, gRPC, Redis, Kafka, SQL) directly from kernel socket calls. Golden signals (latency percentiles, error rates, throughput) are exported to Prometheus, while distributed trace spans are forwarded to Tempo. Fluent Bit captures container logs from host log directories, appends Kubernetes pod metadata, and forwards them to Loki. To prevent telemetry duplication, incident records in BadgerDB store lightweight query pointers (trace IDs, timestamps, pod selectors) rather than copying full metric series or log files.

C. *Two-Stage ML Anomaly Filtering*

To prevent false alarms from benign traffic fluctuations, KubeVision AI employs a two-stage anomaly detection pipeline:

- **Stage 1 (Statistical Screening):** An Isolation Forest model rapidly scores live Prometheus metric vectors. Scores $s \leq \tau_{\text{screen}}$ are filtered out as routine variance, while extreme scores $s > \tau_{\text{high}}$ bypass sequence modeling and route directly to the alert gateway.
- **Stage 2 (Temporal Verification):** Borderline candidates ($\tau_{\text{screen}} < s \leq \tau_{\text{high}}$) are evaluated by an LSTM Autoencoder. An anomaly is confirmed only if reconstruction errors exceed the z-score threshold across $w \geq 3$ consecutive time steps, filtering out transient spikes such as garbage collection pauses.

D. *Parallel SLO Evaluation and Fast-Path Bypass*

Alongside the machine learning (ML) anomaly detection pipeline, KubeVision AI incorporates an independent Service Level Objective (SLO) evaluation layer. While the ML pipeline identifies statistical and temporal deviations in telemetry, the SLO layer evaluates service-level indicators (SLIs) against objectives configured by the user. This parallel design enables the system to respond to user-visible service degradation without requiring every incident to pass through both ML stages.

The SLO evaluator consumes service-level metrics exported to Prometheus, including request latency percentiles, HTTP error ratios, request success rates, and availability indicators. Users can configure the monitored SLIs, target values,

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940



International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 15, Issue 10, October 2026

DOI: 10.17148/IJARCCE.2026.1510145

evaluation windows, breach thresholds, and persistence requirements according to individual service requirements. These settings are maintained independently of the Isolation Forest and LSTM Autoencoder thresholds.

The evaluation and routing process consists of three steps:

1. **Continuous SLO Evaluation:** The evaluator periodically queries the configured SLIs and compares their observed values with the corresponding objectives. For example, a user may configure an upper bound on p95 request latency or a maximum permissible HTTP error ratio. The direction of comparison depends on the indicator: latency and error ratios typically have upper bounds, whereas availability and success ratios typically have lower bounds.
2. **Configurable Breach Confirmation:** A violation is confirmed when a configured SLI exceeds its permitted boundary for the specified evaluation window or persistence duration. Optional hysteresis and cooldown settings can reduce repeated triggers caused by short-lived fluctuations. All thresholds, evaluation windows, and confirmation rules are user-configurable rather than hard-coded into the detection logic.
3. **Direct Gateway Routing:** When a breach satisfies the configured trigger policy, the SLO layer emits a high-priority incident event directly to the Go Hub Gateway, bypassing the Isolation Forest and LSTM Autoencoder stages for that event. The gateway continues to apply deduplication, rate limiting, prioritization, and asynchronous queueing before initiating root cause analysis. Events that do not meet the SLO trigger conditions continue through the existing ML pipeline unchanged.

The SLO layer complements rather than replaces ML-based anomaly detection. It provides a deterministic route for detecting configured service-level violations, while the ML pipeline can identify previously unspecified or less obvious patterns in multivariate telemetry. Both paths converge at the same gateway, preserving a unified incident-handling and diagnostic workflow.

E. *Go Hub Gateway: Alert-Storm Mitigation*

The Go Hub incorporates a dedicated gateway module to protect downstream diagnostic agents from alert storms. The gateway accepts incident events from both the ML anomaly pipeline and the parallel SLO evaluator. Each event carries its source, affected service, triggering indicator, observed value, configured threshold, evaluation window, and severity. SLO-triggered events are marked as high-priority candidates according to the configured policy; they are not exempt from gateway deduplication, rate limiting, or queue management. This convergence preserves a single downstream interface for the diagnostic agents while allowing confirmed SLO violations to avoid ML inference latency.

1. **Deduplication:** Anomaly events are hashed into deterministic fingerprint keys and cached in Redis with a configurable TTL, collapsing redundant alerts from the same service.
2. **Rate Limiting and Prioritization:** A token-bucket rate limiter throttles incoming events, prioritizing incidents based on their anomaly confidence score and estimated blast radius (the number of downstream dependent services derived from Beyla's live dependency graph).
3. **Asynchronous Queueing:** Deduplicated, prioritized events are published to a Redis Stream, decoupling alert ingestion from agent execution and providing backpressure protection.

F. *Multi-Agent RCA Engine with Reflection Verification*

The diagnostic layer uses LangGraph to coordinate four specialized investigation agents under a supervisor pattern:

1. **Parallel Evidence Gathering:** An Orchestrator agent reads an incident event from the Redis Stream and issues targeted diagnostic subtasks to four specialized subagents in parallel:
   - *Metrics Agent:* Queries Prometheus via PromQL to evaluate golden signal degradation around the incident window.
   - *Log Agent:* Queries Loki via LogQL to identify error traces and exceptions in affected pods.
   - *Network Agent:* Queries Tempo via TraceQL to measure critical-path latency and identify failing spans.
   - *K8s Event Agent:* Queries the Kubernetes API server for pod lifecycle signals (OOMKilled, probe failures, eviction events).
2. **Reflection Gate:** The collected evidence is evaluated by a programmatic reflection gate requiring corroboration from at least three of the four telemetry modalities. If evidence is inconclusive, the orchestrator adjusts query parameters and directs the subagents to re-examine the incident.
3. **Synthesis and Self-Validation:** Once sufficient evidence is assembled, an RCA Generator drafts a structured incident report containing the diagnosed root cause, supporting telemetry pointers, affected blast radius, and suggested remediation steps. A Self-Validation Critic verifies each finding against the collected raw evidence, removing any unsupported statements to prevent LLM hallucinations.

 This work is licensed under a Creative Commons Attribution 4.0 International License

DOI: 10.17148/IJARCCE.2026.1510145

G. *Operational Efficiency and Scale-to-Zero*

To keep resource usage minimal on small or single-node clusters, KubeVision AI compiles its core Backend, Gateway, and BadgerDB storage into a single Go binary. In addition, the LangGraph Python agent workers are scaled dynamically using Kubernetes Event-driven Autoscaling (KEDA) based on the Redis Stream queue length, scaling down to zero replicas during normal operations when no incidents are being investigated.

OPEN CHALLENGES, RESEARCH GAPS, AND FUTURE SCOPE

While the proposed KubeVision AI architecture provides a foundation for non-invasive observability and evidence-grounded RCA, several open challenges remain to be addressed:

A. *Future Scope: Proactive Capacity Forecasting via Prophet*

While real-time anomaly detection and RCA focus on diagnosing active incidents, preventing resource exhaustion requires forward-looking predictive modeling. In the current project roadmap, capacity forecasting is designated as a **planned future extension**:

- **Continuous Forecasting Service:** Facebook Prophet [8] uses an additive regression model:

$$y(t)=g(t)+s(t)+h(t)+\varepsilon_t$$

capturing non-linear growth trends $g(t)$, periodic seasonalities $s(t)$, holiday effects $h(t)$, and error terms ε_t. Because Prophet is designed for batch processing, future work will explore wrapping it in an asynchronous background service. Computational model fitting can run on a low-frequency schedule (such as hourly intervals), while forward inference runs every 5 to 15 minutes using cached parameters.

- **Early Saturation Warnings:** Integrating forward projections into the Go Hub will enable proactive alerts—for example, warning operators when a Persistent Volume Claim (PVC) or node memory utilization is projected to exceed 95% capacity within a 30- to 60-minute window, allowing teams to take preventative action before an outage occurs.

B. *Local LLM Deployment in Air-Gapped Clusters*

Enterprise and regulated environments often prohibit transmitting operational logs and telemetry to third-party cloud LLM endpoints due to data privacy policies. Future work will investigate hosting quantized open-source models (such as Mistral-7B or LLaMA-3-8B) directly within the cluster using vLLM or Ollama, evaluating the tradeoff between inference latency, hardware resource consumption, and diagnostic accuracy.

C. *Automated Remediation and Safety Guardrails*

KubeVision AI intentionally follows a human-in-the-loop model, generating actionable remediation advice for SRE approval. Moving toward automated remediation (such as automated pod restarts, traffic throttling, or horizontal pod autoscaling overrides) introduces risks of unintended cascading failures. Establishing formal verification guardrails, rollback policies, and safety constraints represents an important area for future investigation.

CONCLUSION

As distributed applications on Kubernetes grow in scale and complexity, traditional monitoring workflows and manual incident triage become increasingly difficult to maintain. This survey paper reviewed recent developments across non-invasive kernel-level observability, two-stage machine learning anomaly detection, and tool-augmented multi-agent LLM systems for root cause analysis. Our comparative analysis identified several key challenges in current systems, including significant instrumentation overhead, vulnerability to cascading alert storms, and hallucination risks in unconstrained LLM outputs.

To address these challenges, we formulated the architecture of **KubeVision AI**. By combining zero-instrumentation eBPF telemetry via Grafana Beyla, a two-stage anomaly filtering pipeline with confidence gating, an alert-storm mitigation gateway, and a multi-agent LangGraph RCA runtime with programmatic reflection gates, KubeVision AI presents an evidence-grounded framework for cloud-native reliability. The proposed design provides multi-modal diagnostic insights while maintaining resource-efficient scale-to-zero operational economics, establishing a practical foundation for autonomous incident management in Kubernetes environments.

 This work is licensed under a Creative Commons Attribution 4.0 International License

DOI: 10.17148/IJARCCE.2026.1510145

REFERENCES

[1] T. Wang and G. Qi, "A comprehensive survey on root cause analysis in (micro) services: Methodologies, challenges, and trends," *arXiv preprint arXiv:2408.00803*, 2024. [arXiv:2408.00803]

[2] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, and others, "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 (CIKM)*, 2024, pp. 1–13. [arXiv:2310.16340]

[3] P. Chen, Y. Qi, P. Zheng, and D. Hou, "CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *Proceedings of IEEE INFOCOM*, 2014, pp. 1456–1464. [DOI: 10.1109/INFOCOM.2014.6848128]

[4] Y. Meng, S. Zhang, Y. Sun, et al., "Localizing failure root causes in a microservice through causality inference," in *IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, 2020, pp. 1–10. [DOI: 10.1109/IWQoS49365.2020.9213058]

[5] P. Wang, J. Xu, M. Ma, and others, "CloudRanger: Root cause identification for cloud native systems," in *IEEE/ACM 18th International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 492–502. [DOI: 10.1109/CCGRID.2018.00076]

[6] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation Forest," in *IEEE International Conference on Data Mining (ICDM)*, 2008, pp. 413–422. [DOI: 10.1109/ICDM.2008.17]

[7] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "LSTM-based encoder-decoder for multi-sensor anomaly detection," in *ICML Anomaly Detection Workshop*, *arXiv preprint arXiv:1607.00148*, 2016. [arXiv:1607.00148]

[8] S. J. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, vol. 72, no. 1, pp. 37–45, 2018. [DOI: 10.1080/00031305.2017.1380080]

[9] D. Soldani, P. Nahi, H. Bour, and others, "eBPF: A new approach to cloud-native observability, networking and security for current (5G) and future mobile networks (6G and beyond)," *IEEE Access*, vol. 11, pp. 57174–57202, 2023. [DOI: 10.1109/ACCESS.2023.3281480]

[10] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of eBPF for non-intrusive performance monitoring," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2020, pp. 1–7. [DOI: 10.1109/NOMS47738.2020.9110434]

[11] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations (ICLR)*, 2023. [arXiv:2210.03629]

[12] R. Patel, S. K. Mohanty, and J. Chaudhary, "TriFleetRCA: On-premise LLM root cause analysis for Kubernetes," *arXiv preprint arXiv:2609.23766*, 2026. [arXiv:2609.23766]

[13] K8sGPT Authors, "K8sGPT: Giving Kubernetes superpowers via SRE-grade AI analyzers," *CNCF Sandbox Project*, 2024. [Online: https://k8sgpt.ai]

[14] Pixie Authors, "Pixie: Scriptable eBPF observability for Kubernetes workloads," *CNCF Sandbox Project*, 2022. [Online: https://px.dev]

[15] Y. Elkhatib and J. P. Povedano Poyato, "An evaluation of service mesh frameworks for edge systems," in *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '23)*, ACM, 2023, pp. 25–30. [DOI: 10.1145/3578354.3592867]

[16] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 60–70. [DOI: 10.1145/3236024.3236083]

[17] Grafana Labs, "Grafana Beyla: Zero-code eBPF-based auto-instrumentation for application observability," *Grafana Labs Engineering*, 2024. [Online: https://grafana.com/oss/beyla/]

 This work is licensed under a Creative Commons Attribution 4.0 International License