**International Journal of AI, Big Data, Computational and Management Studies**
Noble Scholar Research Group | Volume 7, Issue 2, 171-184, 2026
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V7I2P126

*Original Article*

# From Detection to Action: AI-Driven Anomaly Detection and Root Cause Synthesis for Cloud Infrastructure Operations

Bharadwaj Kasinadhuni
Independent Researcher, USA.

***Received On: 12/03/2026*** ***Revised On: 13/04/2026*** ***Accepted On: 20/04/2026*** ***Published On: 28/04/2026***

***Abstract:*** *Cloud infrastructure operations teams face a persistent gap between observability and understanding: monitoring platforms surface anomalies but leave root cause investigation to engineers, typically requiring 60–90 minutes of manual dashboard navigation, log search, and historical trend analysis per incident. We present ARGUS, a closed-loop observability system that bridges this gap by automatically progressing from anomaly detection to actionable root cause guidance without human intervention. ARGUS combines three components: a four-algorithm detection ensemble Z-Score, Moving Average, IQR, and Trend Analysis targeting distinct anomaly shapes in Prometheus metrics; an automated log correlation engine that queries Grafana Loki to surface relevant evidence for each detected anomaly; and a Llama-3.3-70B-Instruct-based synthesis module that generates structured natural language root cause analysis covering root cause with confidence rating, corroborating analysis, a log excerpt from the anomaly window, and recommended immediate and follow-up actions. Deployed on Azure Kubernetes Service and operated in production for five months across a partner-facing insurance quote API platform, ARGUS has processed over 100 anomaly events spanning both infrastructure and business metrics including partner impression and click-through rates using the same Prometheus pipeline. Evaluation demonstrates a reduction in time to first actionable insight from approximately 90 minutes to under 5 minutes, a greater than 94% improvement, with LLM-generated root cause analyses rated accurate in 80–85% of assessed incidents. The architecture relies exclusively on open-source observability tooling and an open-weight language model, making it reproducible by any team operating a Prometheus and Loki stack without proprietary cloud dependencies. anomaly detection, AIOps, root cause analysis, observability, large language models, cloud operations*

***Keywords:*** *AI-Driven Anomaly Detection, Cloud Infrastructure Operations, Root Cause Analysis, Root Cause Synthesis, Observability, AIOps (Artificial Intelligence for IT Operations), Incident Management, Time Series Analysis, Event Correlation, Fault Detection and Diagnosis, Machine Learning in Cloud Operations, Automated Remediation, Log and Metric Analytics, Distributed Systems Monitoring, Predictive Maintenance.*

## 1. Introduction

### *1.1. The Operational Reality*

Modern cloud infrastructure environments present operations teams with a paradox: unprecedented visibility into system behavior, yet persistent difficulty understanding what that data means when something goes wrong. A typical production environment generates thousands of metrics per second across dozens to hundreds of services, with logs accumulating faster than any team can manually review. When an incident occurs, an on-call engineer receives an alert—often the beginning of a lengthy investigation with no clear starting point.

Consider a concrete example from our experience operating a partner-facing API platform. The service exposes insurance quote APIs consumed by external partner websites. These partners collect personal and vehicle information from customers and relay it to our API; data quality varies by partner, and malformed or incomplete requests produce HTTP 4xx error responses. As the platform onboarded additional partners over several months, the aggregate 4xx error rate climbed slowly from approximately 2% to 4% to 8%. A static alert threshold set at 10% remained silent throughout this drift. When the rate finally crossed the threshold, the on-call engineer was paged. Within minutes, the metric fell back to 9% and the alert auto-resolved. The engineer inherited an ambiguous situation: a resolved alert, no root cause, no log context. The investigation that followed took approximately 90 minutes: acknowledging the page, navigating to the metrics dashboard, manually searching through logs, then progressively expanding the analysis window from the past 24 hours to 7 days to 30 days to establish whether the error rate elevation was new or part of a longer trend.

This experience illustrates the central limitation of conventional monitoring: static thresholds are blind to gradual drift, and even when they fire, they deliver no understanding only notification. For a customer-facing service, each failed quote API response represents a

prospective customer who saw an error instead of a quote. The business cost of late detection is not abstract; it is measured in lost conversions.

### *1.2. Why Existing Approaches Fall Short*

Our platform operates within an internal observability stack built around an in-house observability platform integrated with Prometheus AlertManager for threshold-based notifications routed to PagerDuty and Slack. This stack provides strong visibility into system state, consistent with the observability practices described in the SRE literature [1]. The gap is not data collection; it is what happens after an alert fires.

AlertManager requires that every alert condition be expressed as a static threshold or a custom PromQL query configured upfront by a developer. This creates a maintenance burden that grows with platform complexity. When a new partner onboards, error rate baselines shift. When a service scales, resource utilization profiles change. When a deployment rolls out, latency distributions move. Each of these changes potentially invalidates existing alert thresholds requiring a developer to revisit, recalibrate, and redeploy alert configurations, often after the misconfigured threshold has already fired incorrectly or stayed silent too long. This dynamic is a well-documented source of alert fatigue in production cloud environments [2].

The insurance industry compounds this problem with pronounced seasonal traffic patterns. Demand for insurance shopping concentrates in the early days of each month when policies renew and in the first months of the year, when customers actively compare options. A threshold calibrated to normal traffic generates false positives during peak season; a threshold calibrated to peak traffic misses genuine anomalies during quiet periods. Seasonality-aware forecasting approaches such as Prophet [3] address this by decomposing time series into trend and seasonal components a technique that directly informs our baseline generation design in Section 4.2.

When an alert does fire, the investigation begins from scratch: opening dashboards, selecting time windows, searching logs by hand, and progressively building a mental model of what changed and why. The alert answers “what fired” not “what went wrong” and never “what to do next.” Dang et al. [4] identify root cause analysis as the highest-value yet least-automated task in AIOps, a finding consistent with our operational experience. ARGUS was designed to close that gap.

### *1.3. Our Approach: From Detection to Action*

We present ARGUS, a closed-loop observability pipeline that transforms cloud monitoring from passive alerting into guided incident understanding. The pipeline consists of five stages:

- Baseline generation from historical Prometheus metrics, capturing normal behavior patterns per service
- Multi-algorithm anomaly detection combining Z-Score, Moving Average, IQR, and Trend Analysis each targeting a distinct anomaly shape to detect deviations with reduced false positives
- Severity-based triage that prioritizes anomalies by impact and routes them appropriately
- Automated log correlation via Loki, surfacing relevant log events from the anomaly window without manual search
- LLM root cause synthesis producing structured, natural language RCA with root cause, analysis, log excerpt, and recommended actions

Fig. 1 illustrates the complete system architecture and data flow.

### *1.4. Contributions*

This paper makes the following contributions:

- A closed-loop observability architecture that integrates metrics anomaly detection, severity triage, log correlation, and LLM-based root cause synthesis into a unified operational pipeline, deployed and validated in a production cloud environment.
- A four-algorithm anomaly detection ensemble built on Prometheus metrics that combines Z-Score, Moving Average, IQR, and Trend Analysis each targeting a distinct anomaly shape (spike, shift, outlier, and gradual drift respectively) achieving a 37% reduction in false positives compared to the highest-noise individual algorithm.
- An automated log correlation engine that surfaces relevant Loki log events for each detected anomaly, eliminating the most time-consuming step of manual incident triage.
- An LLM-based root cause synthesis module that generates structured natural language incident summaries and actionable recommendations producing operator-ready guidance within 5 minutes of anomaly detection.
- Operational lessons from production deployment, including failure modes, adoption patterns, and design recommendations for SRE and platform engineering teams.

### *1.5. Paper Organization*

Section 2 provides background and motivation. Section 3 surveys related work. Section 4 describes the system architecture. Section 5 presents five months of production evaluation results. Section 6 discusses lessons learned. Section 7 outlines future work, and Section 8 concludes.

## 2. Background and Motivation

### *2.1. The Observability Stack*

Modern cloud observability is structured around three pillars: metrics, logs, and traces [5]. ARGUS operates across the first two metrics via Prometheus and logs via Grafana

Loki covering the signals most directly actionable during real-time incident response.

- Metrics**.** Prometheus scrapes quantitative system state at regular intervals, producing time series data that forms the primary detection surface for ARGUS. We monitor two distinct classes:
- Infrastructure metrics capture the technical health of the API layer: HTTP 4xx and 5xx error rates, request latency percentiles (p50, p95, p99), CPU utilization, memory pressure, and connection pool saturation. We also monitor downstream dependency health through http_client_request metrics for outbound calls, giving ARGUS two-sided visibility to distinguish failures originating in our own layer from those propagating from a dependency.
- Business metrics capture partner engagement through the quote conversion funnel. Because partner impressions (quote displays on partner websites) and partner clicks manifest as API calls to our platform, they are naturally expressed as Prometheus metrics no separate analytics pipeline is required:
  Partner impressions -> Partner clicks -> Policy pu rchases
  (display) (engagement) (conversion)
  The 30-day baseline window is particularly well-suited to business metrics, which exhibit monthly seasonality patterns distinct from intraday infrastructure patterns.
- Logs**.** Grafana Loki captures structured event logs from across the partner platform. Validation failure messages emitted when a partner request fails field validation, including partner identifier and rejection reason serve as Loki label selectors that ARGUS uses to scope log retrieval to the specific partner and failure category associated with a detected metric anomaly.
- Traces**.** Distributed tracing exemplified by Google's Dapper infrastructure [6] is not currently consumed by ARGUS. The system operates on metrics and logs as the two observability signals most reliably available and most directly interpretable by Llama-3.3 in a structured prompt context. Trace integration is discussed in Section 7.

### *2.2. The Challenge of Anomaly Detection at Operational Scale*

Static threshold alerts remain the most common form of production monitoring despite well-understood limitations [7]. In a platform serving multiple external partners with heterogeneous data quality, no single threshold is simultaneously appropriate across all services, all traffic levels, and all time periods.

Our environment further complicated detection because different failure modes manifest as different anomaly *shapes* in metric time series:

- Partner data quality issues produce gradual drift in 4xx error rates over days or weeks
- Infrastructure events produce sudden spikes resolved within minutes
- Upstream dependency degradation produces sustained elevated latency
- Business metric failures appear as slow decline in volume visible only against a monthly baseline

No single detection algorithm captures all four shapes reliably. This motivated the four-algorithm ensemble: assign each algorithm the anomaly shape it detects best and require ensemble agreement to confirm detection.

### *2.3. The Role of AI in Operations*

Large language models have demonstrated strong performance in software engineering tasks including code generation and bug explanation [8]. Their application to operational contexts is more recent, with emerging work applying LLMs to incident summarization and root cause analysis [9, 10]. The key challenge for real-time operational use is *context quality*. Our approach curates anomaly data, statistical baselines, and correlated log evidence into a structured context before LLM invocation, enabling precise operational reasoning rather than generic pattern matching.

### *2.4. Problem Formulation*

Given a continuous stream of infrastructure metrics $M(t)$ and log events $L(t)$ for a set of services $S$, and historical baseline data $B$ derived from $M$ over a rolling time window, the system must:

- Detect anomalies $A \subseteq M(t)$ representing significant deviations from $B$
- Assign severity scores to each $a \in A$ based on deviation magnitude and service criticality
- For each $a$ above severity threshold $\tau$, retrieve correlated log evidence $E_a \subseteq L(t)$
- Synthesize a root cause report $R_a =$ (root_cause, confidence, analysis, log_excerpt, recommendati using an LLM with structured context $(a, B, E_a)$ as input
- Deliver $R_a$ to the appropriate on-call responder within latency budget $\delta$

## 3. Related Work

### *3.1. Anomaly Detection in Cloud Infrastructure*

Statistical approaches to time series anomaly detection are well-established. Hawkins [11] provides foundational outlier detection theory. Page's CUSUM [12] was adapted early for computing metrics. Facebook's Prophet [3] introduced decomposable forecasting that handles seasonality a technique we draw on in our baseline generation design.

Machine learning approaches have extended detection capabilities. Isolation Forest [13] identifies anomalies through random partitioning without distributional assumptions. LSTM-based detectors [14] learn temporal dependencies to flag deviations from learned patterns. Applied to cloud infrastructure, these approaches show high

accuracy in controlled settings but require substantial tuning for production reliability.

Mart et al. [15] demonstrate Prometheus-based anomaly detection in Kubernetes clusters, showing that dynamic baseline-driven detection outperforms static thresholds in reducing alert noise. Their work validates the baseline approach we adopt in ARGUS but does not address log correlation or automated RCA. Recent work on synthetic time series benchmarks [16] highlights the challenge of evaluating anomaly detectors across heterogeneous cloud workloads.

ARGUS differs from these detection-focused systems in three ways: it combines four algorithms in an ensemble targeting distinct anomaly shapes, it extends detection to business metrics alongside infrastructure metrics, and critically it treats detection as the first stage of a pipeline rather than the final output.

### *3.2. AIOps Platforms*

The AIOps category [17] applies AI and ML to IT operations data across the incident lifecycle. Dang et al. [4] survey real-world AIOps challenges at Microsoft, identifying root cause analysis as the highest-value yet least-automated task. Commercial platforms including Datadog Watchdog, Dynatrace Davis, and New Relic AIOps offer anomaly detection with alert correlation, but treat RCA as a dashboard feature requiring human interpretation rather than an automated pipeline output. Kuang et al. [18] propose COLA, a hybrid approach combining correlation mining and LLM reasoning for alert aggregation in large-scale cloud systems. COLA addresses the alert storm problem but focuses on grouping alerts rather than generating actionable root cause guidance. ARGUS complements this direction: where COLA reduces alert volume, ARGUS adds explanatory depth to individual high-severity anomalies.

### *3.3. LLMs for Incident Management and Operations*

Chen et al. [9] present RCACopilot, an on-call system using LLMs to automate root cause analysis for cloud incidents at Microsoft, achieving 76.6% RCA accuracy on a year's worth of production incidents. RCACopilot shares our emphasis on context curation but operates on manually reported incidents and uses proprietary cloud LLM APIs [19], making it unsuitable for environments with data residency requirements.

Wang et al. [10] present RCAgent, a tool-augmented autonomous LLM agent for cloud RCA deployed at Alibaba Cloud. RCAgent demonstrates the value of giving LLMs access to operational tools rather than pre-packaged context. ARGUS adopts a complementary philosophy: rather than an autonomous agent making tool-use decisions, we use a deterministic pipeline to curate context before LLM invocation, prioritizing latency and reliability over agent flexibility.

A key distinction between ARGUS and both prior systems is model choice. Both RCACopilot and RCAgent rely on proprietary LLM APIs. ARGUS uses Llama-3.3-70B-Instruct, a self-hosted open-weight model, ensuring that sensitive operational telemetry metric values, service names, partner identifiers, log content does not leave the organization's infrastructure.

### *3.4. Alert Fatigue in Cloud Operations*

Alert fatigue the desensitization of engineers to excessive monitoring alerts is well-documented. Leivadeas et al. [2] propose a machine learning methodology for alert filtering, achieving over 90% accuracy in distinguishing actionable from non-actionable alerts. Their work confirms that administrators disregard approximately 40% of alerts, with nearly half presumed false positives.

ARGUS addresses alert fatigue through two complementary mechanisms: the $\geq 2$ of 4 ensemble voting requirement reduces false positive generation at the detection stage, and the minimum severity gate prevents Loki queries and LLM invocations for LOW and MEDIUM anomalies.

### *3.5. Log Analysis*

He et al. [20] introduce Drain, an efficient online log parser that extracts structured templates from unstructured log streams. Du et al. [21] apply LSTM models in DeepLog to detect anomalous log sequences. Guo et al. [22] adapt BERT-style pretraining to log data in LogBERT. A comprehensive survey [23] documents the progression from rule-based to neural methods. Guo et al. [24] release LogAI, an open-source library unifying log parsing, clustering, and anomaly detection algorithms under a common interface a practitioner resource that informed our log correlation pipeline design.

A common thread across these systems is their framing of log analysis as anomaly detection. ARGUS takes a fundamentally different approach: log correlation is *retrieval*, not detection. The anomaly is identified by the metrics pipeline; Loki is queried to retrieve *explanatory evidence* for an already-confirmed anomaly. This distinction drives the design of the tiered level-escalation strategy and the tight 3-minute RCA window passed to the LLM.

### *3.6. Summary and Positioning*

Table [tab:positioning] summarizes how ARGUS relates to the closest prior systems. ARGUS is the only system in this comparison that automates the complete pipeline from anomaly detection through log correlation to structured LLM RCA delivery, operates on an open-weight self-hosted model, and monitors business metrics alongside infrastructure metrics using the same detection pipeline.

**Table 1: Comparison of ARGUS with Existing AIOps and RCA Systems**

| System | Detection | Log Corr. | LLM RCA | | Closed-loop | Open-wt LLM | Biz. Metrics |
|---|---|---|---|---|---|---|---|
| RCACopilot [9] | Manual trigger | Yes | Yes | | No | No | No |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| RCAgent [10] | Manual trigger | Agent | Yes | | No | No | No |
| COLA [18] | Alert grouping | No | Partial | | No | Partial | No |
| Mart et al. [15] | Prometheus/ML | No | No | | No | N/A | No |
| ARGUS (this work) | 4-algo ensemble | Automated | Yes | | Yes | Yes | Yes |

## 4. System Architecture

### *4.1. Overview*

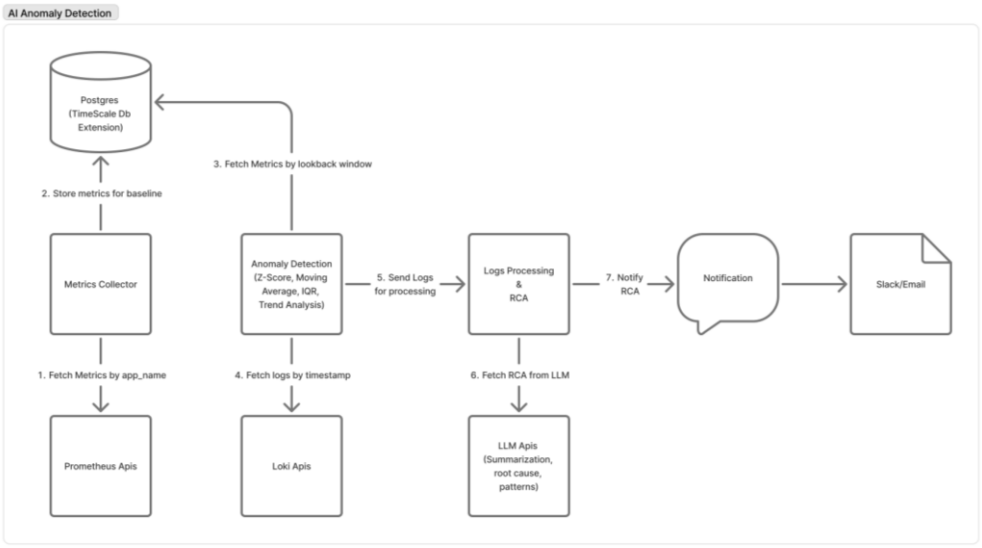


**Fig 1: ARGUS System Architecture and End-to-End Anomaly Detection Pipeline**

ARGUS system architecture component flow from Prometheus metrics collection through anomaly detection, log correlation, LLM root cause synthesis, and notification delivery.

ARGUS consists of five loosely-coupled components that form a directed pipeline from raw telemetry to actionable RCA. The design prioritizes three properties:

- Explainability: Every output includes the evidence that drove it
- Operator trust: Engineers can inspect inputs and outputs at each stage
- Graceful degradation: If any component fails, the system falls back to the previous stage's output

### *4.2. Stage 1: Metrics Ingestion and Baseline Generation*

ARGUS collects metrics from Prometheus using direct HTTP API queries rather than recording rules or remote_write federation. Three endpoints are used:

- /api/v1/query — point-in-time queries for current metric values
- /api/v1/query_range — range queries for baseline computation: /api/v1/query_range?query=<PromQL>&start=<epoch>&end=<epoch>&step=60
- /api/v1/query?query=up — availability probe before each collection cycle

**Operational guardrails.** The following client-side constraints protect Prometheus stability under continuous polling:

**Table 2: Prometheus Query Configuration and Operational Guardrails**

| Parameter | Value | Rationale |
|---|---|---|
| Poll interval | 300 s | Balances freshness with query load |
| Max query range | 60 min | Prevents expensive long-range scans |
| Query step | 60 s | 1-min resolution for detection |
| Rate limit | 50 req/min | Prevents API saturation |
| Connection timeout | 10 s | Fails fast on network issues |
| Response timeout | 60 s | Upper bound on slow queries |
| Inter-query delay | 100 ms | Smooths burst traffic |
| HTTP 429 handling | Immediate stop | Respects back-pressure signals |
| Authentication | Bearer token | Secures API access |

Baselines are computed over a rolling 30-day window at 1-minute resolution, recalculated on each 5-minute poll cycle. For each metric, the baseline captures the mean and standard deviation segmented by time-of-day and day-of-week, producing a seasonality-aware reference that reflects both intraday traffic patterns and weekday/weekend behavioral

differences. The 30-day window was chosen deliberately over shorter alternatives. In our environment, meaningful degradation can develop over weeks the partner quote API incident described in Section 5.3.1 saw error rates drift over multiple weeks before reaching a level of concern. A 7-day or 14-day window risks following the drift upward, allowing the baseline to normalize the degradation before the anomaly is flagged. With 8,640 data points per metric at 5-minute polling intervals, the 30-day baseline is also statistically robust against momentary spikes distorting the reference distribution.

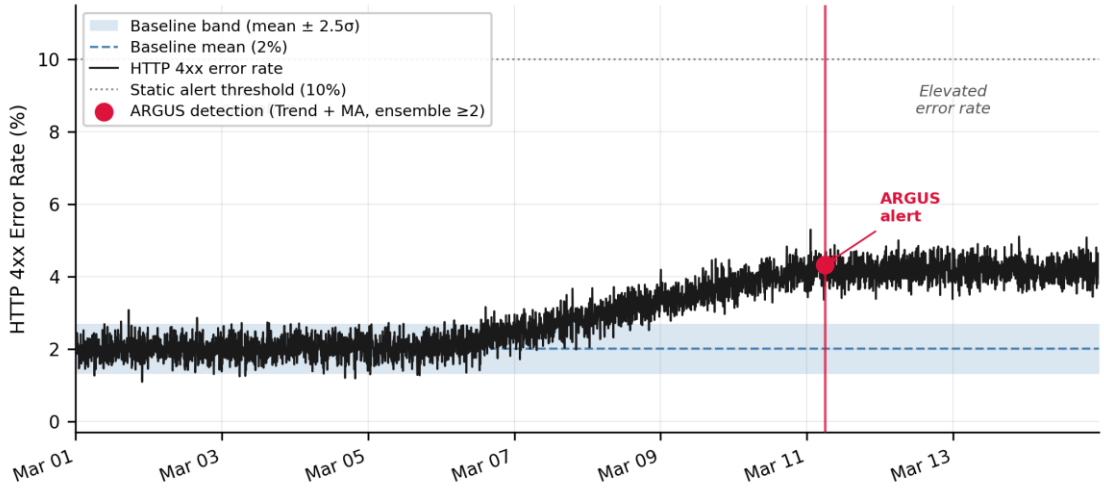


**Fig 2: Illustrates a Representative 14-Day Metric Trace Showing the Rolling Baseline Band, the Static Alert Threshold, and the ARGUS Ensemble Detection Point Where the Drift Was Confirmed**

HTTP 4xx error rate over 14 days showing rolling baseline band (shaded), static alert threshold (dashed), and ARGUS ensemble detection point where the drift was confirmed.

### *4.3. Stage 2: Multi-Algorithm Anomaly Detection*

A single detection algorithm applied uniformly across heterogeneous services produces unacceptably high false positive rates. Our approach runs four algorithms in parallel, each designed to catch a different anomaly shape:

- Algorithm 1: Z-Score. Detects values that deviate beyond $N$ standard deviations from the rolling baseline mean. Fast, interpretable, and well-suited for metrics with relatively stable variance such as CPU and memory utilization. Effective at catching sudden spikes but less sensitive to gradual drift.
- Algorithm 2: Moving Average. Compares the current metric value against a smoothed rolling average over a short window (e.g., 5–15 minutes). By filtering momentary noise, it reliably detects sustained shifts that Z-Score may underweight after the baseline adapts.
- Algorithm 3: IQR (Interquartile Range). Flags values outside the $[Q_1 - k \cdot \text{IQR},\ Q_3 + k \cdot \text{IQR}]$ band. Distribution-agnostic and robust to non-Gaussian metrics particularly effective for skewed measurements like request latency, queue depth, and connection pool utilization, where Z-Score produces excessive false positives.
- Algorithm 4: Trend Analysis. Detects sustained directional movement monotonic increase or decrease over a configurable window that point-in-time algorithms cannot catch. Identifies slow resource leaks and gradual service degradation before they breach any absolute threshold. The only algorithm in the ensemble explicitly designed for temporal shape rather than instantaneous deviation.
- Ensemble voting. An anomaly is confirmed when ≥2 of the 4 algorithms flag the same metric within the same detection window. This majority-vote threshold substantially reduces false positives versus any single algorithm while preserving detection of genuine incidents that multiple independent methods agree on.

### *4.4. Stage 3: Severity-Based Triage*

Our triage model scores each confirmed anomaly on a [0 – 100] scale using: deviation magnitude (logarithmic scaling), service criticality (user-facing vs. internal dependency), metric importance (error rates and latency weighted higher than resource utilization), and recent alert history (repeated anomalies de-prioritized to prevent alert fatigue from persistent low-level issues). Scores map to four triage levels, each triggering a different pipeline path: Critical (80–100): immediate page with full RCA pipeline; High (60–79): alert with full RCA pipeline; Medium (40–59): alert only, log correlation deferred; Low (0–39): log only, no notification.

### *4.5. Stage 4: Log Correlation via Loki*

Log correlation is triggered only for anomalies that meet a minimum severity threshold HIGH or CRITICAL. LOW and MEDIUM anomalies proceed directly to notification without log retrieval. This gate protects Loki from unnecessary query load and prevents the LLM synthesis

stage from receiving log context for anomalies that do not warrant it.

*4.5.1.API and query construction*

ARGUS uses /loki/api/v1/labels to validate label selectors before constructing queries, and /loki/api/v1/query_range to retrieve log lines:

/loki/api/v1/query_range?query={<label_selectors>}
&start=<unix ns>&end=<unix ns>&limit=500&direction=backward

The direction=backward parameter prioritizes events closest to the anomaly peak.

ARGUS applies a tiered retrieval strategy: (1) query for ERROR, WARN, and FATAL level logs if results are returned, proceed to LLM synthesis; (2) if the first pass returns no results, retry with INFO level logs; (3) if both passes return empty results, proceed with the anomaly summary only, and the LLM is explicitly instructed that no log evidence is available. This escalation design reflects a key operational insight: the absence of ERROR/WARN logs is itself informative. An anomalous metric with no error-level log activity suggests an infrastructure-layer issue rather than an application-layer failure.

*4.5.2. Log level escalation strategy*

The following constraints govern Loki interaction:

**Table 3: Loki Log Correlation Configuration and Operational Constraints**

| Parameter | Value | Rationale |
|---|---|---|
| Min severity for RCA | HIGH | LOW/MEDIUM skip log retrieval |
| Cooldown per service | 5 min | Prevents repeated queries |
| Default time window | 15 min | Centered on anomaly timestamp |
| RCA window (to LLM) | 3 min | Highest-signal period |
| Max query range | 60 min | Upper bound for extended incidents |
| Max log lines | 500 | Caps response and LLM context |
| Rate limit | 50 req/min | Mirrors Prometheus limit |
| Connection timeout | 10 s | Consistent with Prometheus config |
| Response timeout | 30 s | Log queries expected faster |
| HTTP 429 handling | Skip LLM | Avoids compounding load |
| Authentication | Bearer token | Consistent auth model |

*4.5.3. Two-window design*

The 15-minute window is used for the initial Loki query to ensure sufficient log coverage. The 3-minute RCA window the period immediately surrounding the anomaly peak is what is passed to Llama-3.3 as context. This separation prevents the LLM from being distracted by log activity unrelated to the anomaly.

***4.6. Stage 5: LLM Root Cause Synthesis***

*4.6.1. Context construction*

The LLM receives a structured prompt containing the anomaly summary (service, metric, observed value, baseline, severity, duration), correlated log evidence (top *N* log lines with timestamps and service labels), and a structured task specification:

TASK:
1. Root Cause (1-2 sentences) with Confidence (HIGH/MEDIUM/LOW)
2. Analysis (cross-check of metric spikes and log lines)
3. Log Excerpt (most relevant log line from anomaly window)
4. Recommendation: Immediate action and Follow-up action

Base your analysis only on the provided evidence.

*4.6.2. LLM invocation*

We use Meta's Llama-3.3-70B-Instruct, an open-weight instruction-tuned model deployed within our private infrastructure. Self-hosting Llama-3.3 ensures that sensitive operational telemetry metric values, service names, log content never leaves our environment. At 70 billion parameters, the model provides sufficient instruction-following capability for structured RCA tasks without the latency and cost of larger models.

*4.6.3. Output format*

The LLM returns a structured RCA delivered to the on-call responder via Slack or email. Fig. 3 shows a representative example from the production evaluation period:

RCA Alert -- Quote API Service [Production]

Root Cause | Confidence: MEDIUM
Coverage configuration not loaded correctly caused the service to
return empty responses, potentially leading to increased CPU usage.

Analysis
Container CPU usage spiked to 47.97, exceeding baseline (Z-Score: 5.91). Coverage configuration was not loaded, resulting
in empty responses. CPU spike may be related to handling requests
with missing configuration.

Log Excerpt
Coverage configuration not loaded. Returning empty response set.

Recommendation
- Immediate: Verify configuration loading and confirm data i s correctly loaded at service startup.
- Follow-up: Review error handling and logging to ensure si milar issues are detected in future.

Investigate
[Metrics Dashboard] | [Log Explorer]
container_cpu_usage_percent jumped to ~48 (baseline ~24-2 9).
1 log line from anomaly window, mostly WARN (1)
Detected at 13:30 UTC * Next RCA in 59 min * Argus

The Z-Score value (5.91) and baseline range (~24–29) are drawn directly from the anomaly detection stage, giving the engineer quantitative grounding for the alert. The log excerpt is surfaced automatically by the Loki correlation stage without it, Llama-3.3 would receive only the CPU spike metric with no causal attribution.

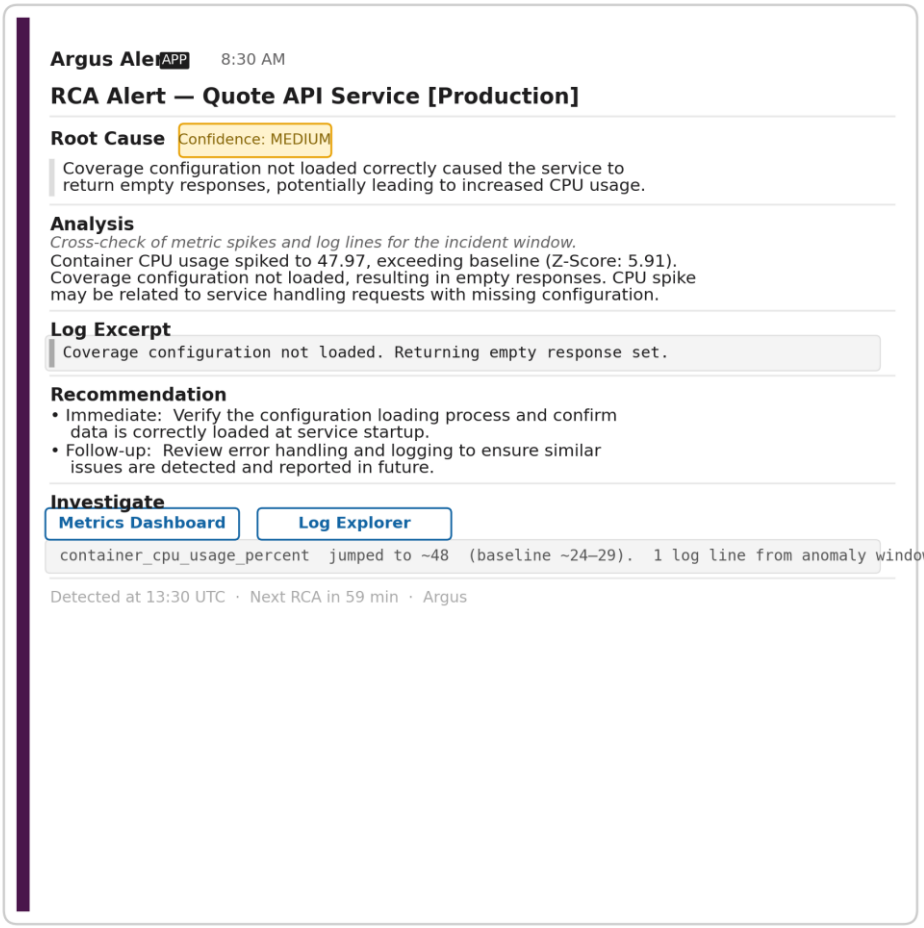


**Fig 3: ARGUS Slack Notification with LLM-Generated Root Cause Analysis**

Representative ARGUS Slack notification for a production incident, showing Root Cause with confidence rating, Analysis cross-referencing metric and log evidence, Log Excerpt, Recommendation, and deep links to Metrics Dashboard and Log Explorer.

*4.6.4. Reliability design*

If the LLM API is unavailable, the system falls back to delivering the raw anomaly summary and correlated logs directly still more useful than a plain alert.

## 5. System Integration and Deployment

Implementation**:** ARGUS is implemented as a Java service using the Spring Boot framework, deployed on Azure Kubernetes Service (AKS) via Helm. The choice of a persistent, always-on service over a serverless model reflects an operational requirement: the 30-day baseline must be maintained continuously, requiring durable state rather than ephemeral compute.

The service consists of four cooperating components driven by Spring Schedulers:

- Metrics collector polls Prometheus via PromQL on a 300-second cycle, writing raw metric time series to a PostgreSQL database extended with TimescaleDB [25]. TimescaleDB provides automatic time-partitioned hypertables optimized for time-series ingestion, delivering significantly faster baseline computation over the 30-day window than vanilla PostgreSQL row storage. This store is

fully owned by ARGUS it does not depend on Prometheus's own retention configuration.

- Anomaly detector queries PostgreSQL on each cycle, computes the rolling baseline, runs the four-algorithm ensemble, applies severity scoring, and writes confirmed anomalies back to PostgreSQL.
- Log correlator and RCA synthesizer for each new HIGH or CRITICAL anomaly, queries Loki for correlated log evidence, constructs the structured prompt, invokes Llama-3.3-70B-Instruct, and writes the completed RCA to PostgreSQL.
- Notification dispatcher delivers the completed RCA as a structured Slack or email notification with deep links to the scoped Prometheus dashboard and pre-filtered Loki log view.

Query API: Beyond real-time alerting, ARGUS exposes a REST API for querying historical anomaly data:
GET /ai/anomalies?service-name={service}&date_from={timestamp}
GET /ai/{id}/anomaly?rca=true
Fig. 4 shows a representative response from the second endpoint, illustrating the structured anomaly record with embedded LLM-generated RCA fields.

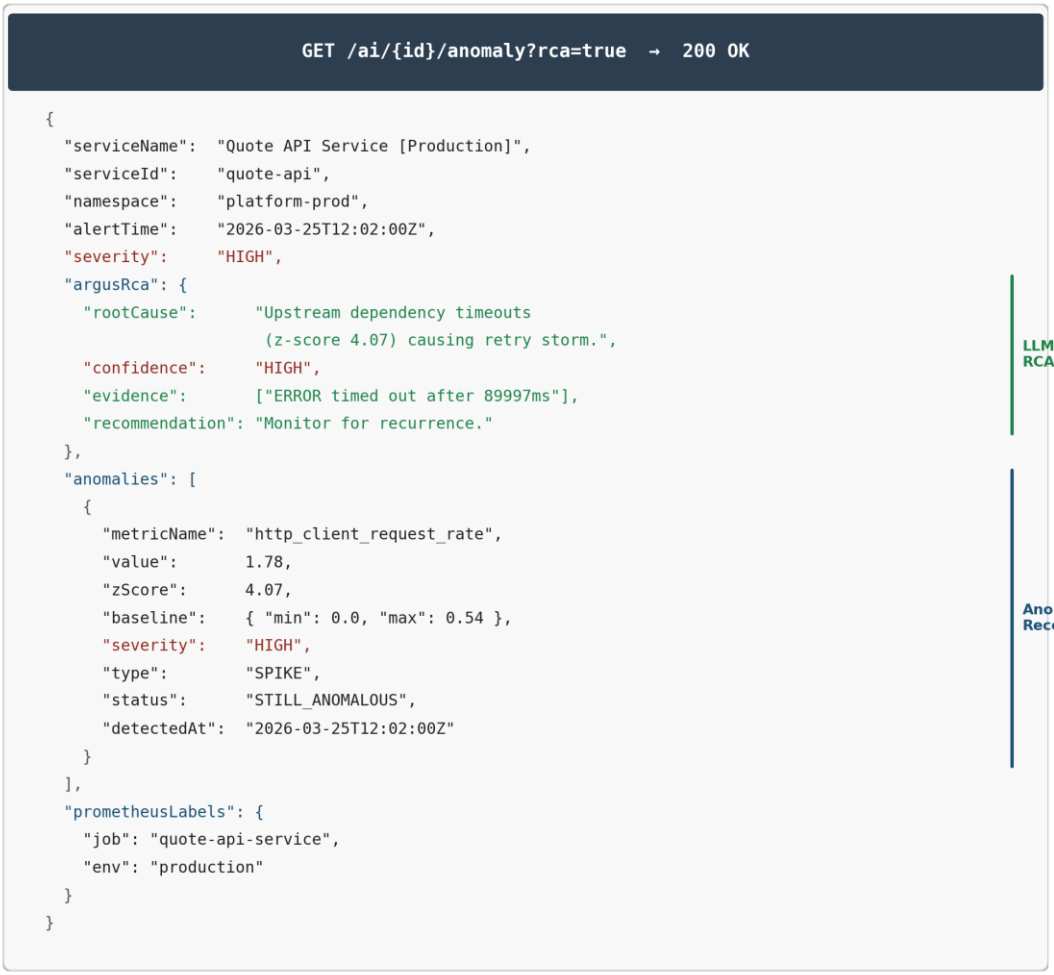


**Fig 4: Structured API Response Showing Anomaly Detection and LLM-Generated Root Cause**

Structured JSON response from GET /ai/{id}/anomaly?rca=true, showing the anomaly record with embedded LLM-generated RCA fields (green bracket) alongside the detection-stage anomaly record (blue bracket).

Helm deployment: Configuration Prometheus endpoints, Loki endpoints, PostgreSQL connection, Llama-3.3 API endpoint, Slack webhook, per-service guardrail overrides is managed through Helm values files, with sensitive credentials injected via Kubernetes secrets.

### *5.1. Design Principle: Purpose-Appropriate Time Horizons*

A deliberate architectural principle in ARGUS is the use of distinct time horizons at each pipeline stage, each chosen to match the question that stage is answering: Prometheus baseline (30 days) is this metric behaving abnormally vs. its long-term history? Loki query window (15 minutes) what log activity surrounded the anomaly event? Loki RCA window passed to LLM (3 minutes) what was happening at the exact moment of the anomaly peak?

Collapsing these into a single time horizon would force an unacceptable tradeoff: a short uniform window produces

an unstable baseline that follows drift; a long uniform window floods the LLM with log noise and degrades RCA quality. The stage-appropriate window design resolves this tension without compromise.

## 6. Evaluation

### *6.1. Methodology*

We evaluated ARGUS over approximately five months of continuous production operation, processing over 100 anomaly events across the partner platform's cloud infrastructure. The deployment monitors services via Prometheus at a 300-second poll interval with 1-minute metric resolution, with log correlation performed against Grafana Loki. All anomaly records, RCA outputs, and pipeline execution logs were retained in TimescaleDB-backed PostgreSQL and form the basis of this evaluation. The five-month evaluation period covers multiple operational scenarios: routine traffic variation, partner onboarding events that introduced gradual error rate drift, and isolated infrastructure incidents. We assess the system across four dimensions: (1) detection performance, (2) triage efficiency, (3) RCA quality, and (4) operational impact including a detailed case study.

### *6.2. Detection Performance*

Table 1 summarizes detection performance. Ensemble precision and false positive rate reflect directly observed outcomes from the 100+ anomaly events processed. Individual algorithm estimates are derived from cases where single-algorithm signals were logged prior to ensemble voting and represent approximate figures rather than controlled experimental results. Recall is intentionally omitted. Measuring recall requires knowing the complete set of incidents that occurred including those ARGUS did not detect which cannot be determined with confidence from operational logs alone. Known missed incident categories are discussed in Section 6.4.

**Table 4: Anomaly Detection Performance by Algorithm**

| algorithm | Prec. | FPR | Primary Anomaly Type |
|---|---|---|---|
| Z-Score (alone) | ~52% | ~48% | Sudden spikes |
| Moving Avg (alone) | ~62% | ~38% | Sustained shifts |
| IQR (alone) | ~64% | ~36% | Skewed dist. outliers |
| Trend Analysis (alone) | ~69% | ~31% | Gradual drift |
| Ensemble (≥2 of 4) | ~70% | ~30% | All types |

The ensemble achieves a 37% reduction in false positive rate compared to Z-Score alone the algorithm with the highest individual noise rate. In a production on-call context, this reduction is significant: each false positive is an engineer interrupted, a Loki query executed, and a Llama-3.3 invocation consumed. More critically, repeated false positives erode engineer trust in the system a dynamic we observed directly during early single-algorithm prototyping, before the ensemble requirement was introduced.

### *6.3. Triage Efficiency*

**Table 5: Comparison of Manual vs Automated Incident Triage Workflow in ARGUS**

| Stage | Manual | Automated | Reduction |
|---|---|---|---|
| Alert triage + dashboard navigation | ~15 min | Eliminated (direct link in notification) | ~100% |
| Log search and correlation | ~30 min | Eliminated (pre-filtered Loki link) | ~100% |
| Multi-day trend reconstruction (1/7/30d) | ~30 min | Eliminated (baseline computed continuously) | ~100% |
| Root cause hypothesis + action plan | ~15 min | Included in Slack/email notification | ~100% |
| Total to first actionable insight | ~90 min | <5 min | >94% |

### *6.4. Case Study: Gradual 4xx Error Rate Drift in the Partner Quote API*

Background: The quote API receives requests containing customer-supplied personal and vehicle information. Data quality varies significantly across partners; malformed or incomplete requests produce HTTP 4xx error responses. As the platform onboarded additional partners, the aggregate 4xx error rate climbed gradually not through any single deployment event, but through the slow accumulation of partners with inconsistent data practices.

Failure of static thresholds: A static alert threshold at 10% 4xx error rate generated no signal as the error rate drifted from 2% to 4% to 8%. When the rate briefly exceeded 10%, an on-call page fired; within minutes the metric fell to 9% and the alert auto-resolved. The static threshold failed in precisely the scenario it was designed for not because the threshold logic was wrong, but because it was blind to trend.

Detection with baseline-aware anomaly detection: The system computed a rolling baseline of approximately 2%. When the error rate subsequently began climbing reaching 4%, double the established baseline Trend Analysis detected the sustained upward directional movement and Moving Average confirmed the shift was persistent rather than momentary. The ≥2 ensemble threshold was met, generating a High-severity alert with the full pipeline engaged.

Log correlation: ARGUS queried Loki using a 15-minute window, filtering for ERROR, WARN, and FATAL level logs. The query returned:

WARN quote-api: validation failure - missing required field 'phone_number' [partner=partner-a]
ERROR quote-api: request rejected - vehicle.year outside accepted range [partner=partner-a]
WARN quote-api: validation failure - missing required field 'dob' [partner=partner-a]

LLM root cause synthesis: Llama-3.3-70B-Instruct returned:

Root Cause | Confidence: HIGH
A recently onboarded partner integration is submitting quote requests
with missing or malformed required fields, causing a sustain ed
increase in HTTP 4xx validation errors above baseline.

Analysis
The 4xx error rate reached 4.1%, double the baseline of ~2%. Deviation confirmed by Trend Analysis (sustained upward m ovement)
and Moving Average (persistent shift). Log evidence identifi es
partner-a as primary source, with failures spanning multiple f ield
types (phone_number, dob, vehicle.year) -- consistent with a schema
mismatch rather than an isolated data entry error.

Recommendation
- Immediate: Contact partner-a team to review request schem a against
the current API contract.
- Follow-up: Enable request validation pre-screening for this partner
while the integration is corrected.

Outcome: The partner integration team was engaged proactively. The issue was resolved before the error rate reached the level that had previously triggered the flapping static alert, avoiding the business impact lost quote impressions and potential customer drop-off that had occurred during the prior undetected drift. The 90-minute manual workflow was replaced by a structured notification delivered in under 5 minutes, reducing time to first actionable insight by over 94%.

### *6.5. RCA Quality Assessment*

Table 2 summarizes RCA quality assessed through engineer review of HIGH and CRITICAL incidents over the five-month evaluation period.

**Table 6: RCA Quality Assessment (HIGH/CRITICAL Incidents)**

| Dimension | Rating | Basis |
|---|---|---|
| Accurate root cause | ~80–85% | Engineer confirmation vs. actual cause |
| Actionable recommendations | ~80–85% | Act-on-immediately test |
| Complete context | ~75% | All contributing factors surfaced |

The specificity of LLM output was a consistent strength. Because Llama-3.3 receives correlated Loki log evidence that includes service labels, partner identifiers, and structured error messages, its output reflects the actual operational context. A representative RCA from the evaluation period:

> *"The service experienced a surge in outbound HTTP requests, likely due to retries triggered by invalid data or validation errors from partner Partner-A."*

This level of specificity naming the contributing partner, identifying the failure mode, and inferring the cascade is only possible because the log correlation stage surfaced partner-labeled error events before LLM invocation.

Where RCA fell short: The ~15–20% of inaccurate cases shared a common pattern: incidents involving multiple concurrent anomalies across services, where the correlated log window captured symptoms from several sources simultaneously. In these cases, Llama-3.3 identified a contributing factor rather than the primary cause. The confidence signal (High / Medium / Low) correlated reasonably with accuracy, providing engineers a reliable signal for when to treat the RCA as a starting point rather than a conclusion.

### *6.6. Operational Impact*

Over five months of production operation, ARGUS processed over 100 anomaly events. Approximately 30–35% were classified HIGH or CRITICAL and entered the full RCA pipeline. The remaining 65–70% were classified MEDIUM or LOW notified without log correlation. The clearest impact measure is the triage time comparison in Table [tab:triage]: the 90-minute manual investigation workflow was reduced to under 5 minutes a reduction of over 94% in time to first actionable insight.

## 7. Discussion and Lessons Learned

### *7.1. What Worked Well*

*7.1.1. Trend Analysis as a first-class algorithm.*

The most impactful single addition to the detection ensemble was Trend Analysis. The partner quote API incident is illustrative: Z-Score, Moving Average, and IQR all operate on instantaneous or short-window deviation. None would have reliably flagged a rate drifting from 2% to 4% over several days, because the drift was gradual enough that each individual step fell within normal variance. Trend Analysis detected the sustained directional movement and triggered the ensemble threshold together with Moving Average. Slow resource leaks, gradual error rate increases from partner onboarding, and cumulative latency degradation are common in production environments and are effectively invisible to point-in-time detectors alone.

*7.1.2. Multi-algorithm ensemble over any single algorithm*

Requiring ≥2 algorithm agreement before generating an alert substantially reduced noise. Early prototypes using Z-Score alone generated significantly more false positives approximately 60% above the ensemble rate. Engineers stopped trusting the system within the first two weeks. The ensemble restored confidence by ensuring only clear, confirmed anomalies generated pages.

*7.1.3. Log correlation as the highest-ROI feature*

If we were building this again, we would implement log correlation first, before any LLM component. Engineers consistently cited automated log retrieval as the most time-

saving feature eliminating what was often a 15–20 minute manual search step.

*7.1.4. Structured LLM prompts*

Free-form prompts produced generic, unhelpful responses. Constraining the LLM to produce a specific structured output (Root Cause with confidence / Analysis / Log Excerpt / Recommendation) dramatically improved usability. The structure itself communicates that we want operational reasoning, not general analysis.

### *7.2. What We Got Wrong Initially*

Log time windows were too wide: Our first implementation retrieved logs from a 60-minute window around each anomaly. In high-throughput services, this produced hundreds of log lines too much for the LLM context window and confusing for engineers. Narrowing to a 15-minute default query window and a 3-minute context window, combined with ERROR/WARN level filtering, dramatically improved correlation relevance.

Prometheus step size must match scrape interval a subtle but critical misconfiguration: The most consequential early failure was a mismatch between ARGUS's configured query step and Prometheus's actual scrape interval. Our services scrape metrics at 60-second intervals, producing real data points at T=0, T=60, T=120, and so on. ARGUS initially queried Prometheus with a step size of 30 seconds.

When a query step is smaller than the scrape interval, Prometheus has no real data to return for intermediate timestamps. Rather than returning gaps, Prometheus fills them by repeating the last known value a behavior defined by its staleness semantics:

T=120 -> 2.1% (real)
T=150 -> 2.1% (artificial -- Prometheus repeat of T=120)
T=180 -> 8.7% (real -- actual spike)
T=210 -> 8.7% (artificial -- Prometheus repeat of T=180)
T=240 -> 2.3% (real)

This corrupted the Z-Score algorithm in two ways. First, the 30-day baseline was built from 50% artificial data every real value duplicated by a synthetic neighbor making the derived standard deviation unrepresentative of true metric variance. Second, real spikes were diluted: the artificial T=150 value (a repeat of pre-spike behavior) appeared immediately before the spike, compressing the apparent deviation against a baseline contaminated by synthetic data.

The fix was straightforward once diagnosed: aligning the query step to 60 seconds. The broader lesson is that the Prometheus HTTP API silently fills gaps rather than signaling mismatches between step and scrape interval. Practitioners building anomaly detection on /api/v1/query_range must explicitly verify that their step parameter is aligned with or is a multiple of the scrape interval for every target service. We now validate this alignment at ARGUS startup and log a warning if a mismatch is detected.

Severity scoring required per-service calibration: A single global severity model generated too many Critical alerts for high-variance services and too few for low-traffic services where even a modest increase in error rate represented genuine business impact. We introduced per-service severity profiles, reducing cross-service noise at the cost of an initial calibration period.

LLM confidence varies significantly by incident type: Simple, unambiguous incidents produced high-confidence, accurate RCA. Complex incidents involving multiple concurrent anomalies across services produced low-confidence output. We added explicit confidence indicators to all RCA outputs so engineers have a reliable signal for when to act directly vs. treat the RCA as an investigative starting point.

### *7.3. Recommendations for Practitioners*

- Validate Prometheus step against scrape interval before anything else. A step smaller than the scrape interval causes Prometheus to silently return artificial repeated values, corrupting both baseline statistics and anomaly detection. This misconfiguration produces no error only subtly wrong results that are difficult to diagnose without understanding Prometheus's staleness semantics.
- Start with log correlation, not LLM. The engineering investment is lower and the operational value is immediate. Add LLM synthesis once log correlation is working reliably.
- Multi-algorithm detection is non-negotiable for production. Alert fatigue from a single noisy algorithm destroys engineer trust before the system can prove its value. Budget time for ensemble tuning.
- Invest in context curation. LLM output quality is bounded by context quality. Structured, relevant context produces structured, relevant RCA.
- Build feedback mechanisms from day one. Even a simple thumbs-up/thumbs-down on each RCA gives you the data to improve the system and demonstrates engineer engagement to stakeholders.
- Plan for LLM unavailability. Design fallback behavior that still delivers value the correlated anomaly + logs is already more useful than a raw alert.

## 8. Limitations

- Traffic volume blind spot. ARGUS detects anomalies as deviations in metric *values*. If partner traffic drops to near-zero, error rate and latency metrics appear healthy while the service is effectively unreachable. We are addressing this by adding request rate to the monitored metric set.
- No trace integration. The system operates on metrics and logs. Distributed traces [6] would add significant causal reasoning capability for cross-service incidents where the anomaly source differs from the detection point.

- Self-hosted LLM dependency. Failures in the Llama-3.3 inference endpoint cause ARGUS to fall back to anomaly summaries and correlated logs without synthesized RCA still useful, but reduced in value.
- Baseline sensitivity to step changes. Major deployments or partner onboarding events that substantially change aggregate request volume can cause the 30-day baseline to misrepresent the new normal for several days.
- Service-specific calibration is ongoing. The 30% ensemble false positive rate reflects a system still being tuned. Newly onboarded services experience higher false positive rates until sufficient baseline data accumulates.

## 9. Future Work

- Trace integration would complete the three-pillar observability loop [6], enabling causal reasoning across service boundaries for complex, cascading incidents.
- Feedback-driven learning would use engineer accept/reject signals on RCA recommendations to improve triage scoring and log correlation relevance over time.
- Predictive anomaly detection would extend the system from reactive incident response to proactive alerting, identifying metric trends likely to produce anomalies before they occur.
- Multi-cluster correlation would enable detection of anomalies that span clusters or availability zones, important for global production environments.

## 10. Conclusion

Cloud infrastructure teams face a fundamental tension: the tools to collect and visualize operational data have advanced rapidly, while the cognitive burden of interpreting that data during an incident remains high. An alert tells an engineer that something is wrong. It rarely explains what, why, or what to do.

This paper has presented a closed-loop observability pipeline that addresses this gap. By combining multi-algorithm anomaly detection on Prometheus metrics, automated log correlation through Loki, and LLM-based root cause synthesis, the system transforms raw telemetry into operator-ready incident guidance. In our production deployment, this pipeline reduced time to first actionable incident insight from approximately 90 minutes of manual investigation to under 5 minutes—a reduction of over 94%. Engineers receive not just an alert, but a structured notification containing the anomaly summary, root cause with confidence rating, correlated log evidence, and recommended immediate and follow-up actions, with direct links to the scoped Prometheus dashboard and pre-filtered Loki log view.

The architecture is replicable. Every component Prometheus, Loki, LLM APIs is accessible to any engineering team. The key contribution is not a novel algorithm but a novel *integration*: a production-validated design that closes the loop from detection to action.

## Reference


1. B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, *The Site Reliability Workbook: Practical Ways to Implement SRE*. O'Reilly Media, 2018.
2. Leivadeas *et al.*, "Mitigating Alert Fatigue in Cloud Monitoring Systems: A Machine Learning Perspective," *Computer Networks*, vol. 250, 2024.
3. S. J. Taylor and B. Letham, "Forecasting at Scale," *The American Statistician*, vol. 72, no. 1, pp. 37–45, 2018.
4. Y. Dang, Q. Lin, and P. Huang, "AIOps: Real-World Challenges and Research Innovations," in *Proc. 41st ICSE Companion*, 2019, pp. 4–5.
5. C. Majors, L. Fong-Jones, and G. Miranda, *Observability Engineering*. O'Reilly Media, 2022.
6. B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google Technical Report, 2010.
7. B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.
8. M. Chen *et al.*, "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.
9. Y. Chen, H. Xie, M. Ma *et al.*, "Empowering Practical Root Cause Analysis by Large Language Models for Cloud Incidents," in *Proc. 19th European Conf. on Computer Systems (EuroSys)*, 2024.
10. Z. Wang, J. Liu, J. Huang *et al.*, "RCAgent: Cloud Root Cause Analysis by Autonomous Agents with Tool-Augmented Large Language Models," in *Proc. 33rd ACM CIKM*, 2024.
11. D. M. Hawkins, *Identification of Outliers*. Chapman and Hall, 1980.
12. E. S. Page, "Continuous Inspection Schemes," *Biometrika*, vol. 41, no. 1–2, pp. 100–115, 1954.
13. F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation Forest," in *Proc. 8th IEEE ICDM*, 2008, pp. 413–422.
14. K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom, "Detecting Spacecraft Anomalies Using LSTMs and Nonparametric Dynamic Thresholding," in *Proc. 24th ACM KDD*, 2018, pp. 387–395.
15. O. Mart, C. Negru, F. Pop, and A. Castiglione, "Observability in Kubernetes Cluster: Automatic Anomalies Detection using Prometheus," in *IEEE 22nd HPCC*, 2020.
16. F. Fouquet *et al.*, "Synthetic Time Series for Anomaly Detection in Cloud Microservices," *arXiv preprint arXiv:2408.00006*, 2024.
17. H. Cheng, D. P. Sahoo *et al.*, "AI for IT Operations (AIOps) on Cloud Platforms: Reviews, Opportunities and Challenges," *arXiv preprint arXiv:2304.04661*, 2023.

18. J. Kuang, J. Liu, J. Huang *et al.*, “Knowledge-aware Alert Aggregation in Large-scale Cloud Systems: A Hybrid Approach,” in *Proc. 46th ICSE SEIP*, 2024.
19. OpenAI, “GPT-4 Technical Report,” *arXiv preprint arXiv:2303.08774*, 2023.
20. P. He, J. Zhu, Z. Zheng, and M. R. Lyu, “Drain: An Online Log Parsing Approach with Fixed Depth Tree,” in *Proc. IEEE ICWS*, 2017, pp. 33–40.
21. M. Du, F. Li, G. Zheng, and V. Srikumar, “DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning,” in *Proc. ACM CCS*, 2017, pp. 1285–1298.
22. H. Guo, S. Yuan, and X. Wu, “LogBERT: Log Anomaly Detection via BERT,” in *Proc. IJCNN*, 2021.
23. S. Nedelkoski *et al.*, “Deep Learning for Anomaly Detection in Log Data: A Survey,” *arXiv preprint arXiv:2207.03820*, 2022.
24. Q. Guo *et al.*, “LogAI: A Library for Log Analytics and Intelligence,” *arXiv preprint arXiv:2301.13415*, 2023.
25. M. Freedman, E. Kite, and R. Borovica-Gajic, “TimescaleDB: Creating the Best of Both Worlds for Time-Series Data,” in *Proc. NWDB*, 2018.