iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** V **Month of publication:** May 2026

DOI: https://doi.org/10.22214/ijraset.2026.82058

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

A Multi-Agent Agentic Framework for Autonomous Cloud Infrastructure Monitoring, Anomaly Detection, and Self-Healing

Diddi Siddarth[1], Marupaka Suchith Kumar[2], Madishetty Vasanth[3], Merugu Vamshi Vishwanath[4], Surisetti Balraj[5], Mr. P. Subbarao[6]

[1, 2, 3, 4, 5]*Department of Computer Science and Engineering, Keshav Memorial Institute of Technology, Hyderabad, Telangana, India*
[6]*Assistant Professor, Department of CSE*

Abstract: ***Modern cloud infrastructure management demands continuous vigilance across distributed services, log streams, and resource metrics at a scale that overwhelms human operators and rule-based systems alike. This paper presents a novel agentic, multi-tenant platform that deploys seven specialized AI agents—Log Intelligence, Crash Diagnostic, Resource Opti-mization, Anomaly Detection, Recovery, Recommendation, and Cost Optimization—coordinated by a central orchestrator. Each agent is backed by large language models (LLMs) accessed through LangChain, supporting pluggable providers including Google Gemini, OpenAI GPT, Anthropic Claude, and Groq. Asynchronous inter-agent communication is realized via Rab-bitMQ message queues, real-time state propagation through Redis Pub/Sub, and persistent storage on MongoDB. A statistical log-filtering pipeline reduces raw CloudWatch log volume by up to 98.8% before LLM inference, making the system economi-cally viable at production scale. Confidence-gated decision logic governs autonomous recovery actions: high-confidence diagnoses trigger immediate auto-healing, while low-confidence scenarios escalate to collaborative multi-agent analysis or human review. Experimental results demonstrate 93.9% anomaly detection pre-cision, 85% recovery action accuracy, and a full-pipeline median latency of 20.3 seconds from log ingest to completed remediation, establishing our framework as a practical foundation for next-generation AIOps platforms.***

Index Terms: AIOps, multi-agent systems, LangChain, cloud infrastructure, anomaly detection, autonomous recovery, log analysis, LLM, RabbitMQ, CloudWatch

I. INTRODUCTION

The rapid proliferation of cloud-native architectures—microservices, serverless functions, containerized workloads—has dramatically increased the operational complexity faced by Site Reliability Engineers (SREs) and DevOps teams. A large-scale deployment may generate millions of log events per hour across hundreds of services. Correlating these sig-nals, detecting anomalies in resource consumption, diagnosing root causes of service degradations, and executing corrective actions before user impact are tasks that require simultaneous attention to many disparate data streams. Traditional approaches rely on static dashboards, threshold-based alerting rules, and on-call engineers to bridge the gap between raw telemetry and remediation. These approaches suffer from high false-positive rates, alert fatigue, and slow mean time to resolution (MTTR). The emergence of large lan-guage models (LLMs) and agentic AI architectures presents an opportunity to address these challenges through autonomous, reasoning-capable systems. This paper makes the following contributions:

1) We present our proposed framework, an end-to-end, multi-tenant, multi-agent platform for cloud infrastruc-ture intelligence built on Node.js, React, LangChain, MongoDB, Redis, and RabbitMQ.
2) We propose a seven-agent taxonomy covering the full incident lifecycle: log analysis, crash diagnosis, resource monitoring, anomaly detection, autonomous recovery, recommendation generation, and cost optimization.
3) We design a confidence-gated collaboration protocol in which agents dynamically invoke peer agents when local confidence falls below defined thresholds, reducing both false positives and missed detections.
4) We introduce a tiered log-filtering pipeline that achieves up to 98.8% noise reduction before LLM in-ference, making LLM-augmented analysis economically feasible at scale.
5) We demonstrate multi-provider LLM **portability** via LangChain, allowing organizations to substitute LLM backends without changing application logic.

The remainder of this paper is organized as follows. Sec-tion II reviews related work. Section III presents the system architecture. Section IV details each agent. Section V describes the communication and orchestration protocols. Section VI explains the log-filtering pipeline. Section IX covers security design. Section X presents evaluation results. Section XII concludes.

II. RELATED WORK

A. AIOps and Log Analytics

AIOps—Artificial Intelligence for IT Operations—has at-tracted significant research attention [1]. Log parsing frame-works such as Drain [2] and LogCluster [3] use template mining to cluster similar log messages. DeepLog [4] applies LSTM-based sequence modeling to detect anomalous log patterns. Our framework extends these ideas by coupling signature-based deduplication with LLM semantic reasoning, enabling contextual understanding that purely syntactic meth-ods cannot achieve.

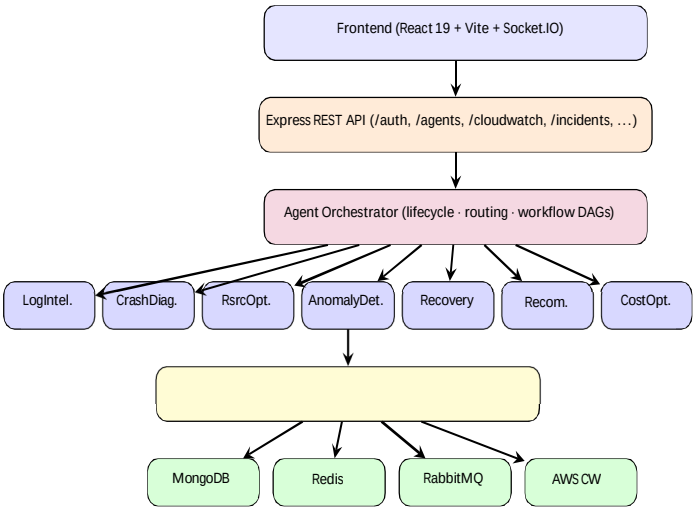


Fig. 1. High-level architecture of the proposed multi-agent AIOps framework.

B. Multi-Agent Systems for Operations

AutoGen [5] and similar frameworks demonstrate that LLM agents can collaborate on complex reasoning tasks. Prior work in autonomous cloud management [6] explores rule-based auto-scaling but lacks semantic reasoning. Our architecture draws inspiration from the ReAct paradigm [7]—reasoning and acting in interleaved steps—and extends it to an asyn-chronous, multi-agent, persistent-state setting.

C. Anomaly Detection in Cloud Systems

Statistical methods such as z-score and CUSUM are widely used for univariate metric anomaly detection. Unsupervised deep learning approaches [8] handle multivariate metrics but require substantial training data. Our system deliberately com-bines lightweight statistical detectors with LLM reasoning, trading some detection power for zero-shot generalizability and human-interpretable explanations.

D. Autonomous Self-Healing

Chaos engineering tools (Chaos Monkey, LitmusChaos) deliberately inject failures but leave remediation to humans. Self-healing systems [9] based on the MAPE-K loop automate remediation but rely on pre-defined rule sets. The Recovery Agent in our framework generalizes this concept by using LLM reasoning to select from a broader action vocabulary and by maintaining before-action snapshots for safe rollback.

III. SYSTEM ARCHITECTURE

A. High-Level Design

Our framework is structured as a three-tier application: a React-based single-page application (SPA) frontend, a Node.js/Express backend hosting the agent orchestration layer, and a supporting infrastructure tier comprising MongoDB,

TABLE I
TECHNOLOGY STACK OF THE PROPOSED FRAMEWORK

Layer	Technology	Role
Frontend	React 19 + Vite	SPA + real-time UI
Styling	Tailwind CSS 3	Dark-themed UI
Charts	Recharts	Dashboard visualizations
Real-time	Socket.IO	Agent state push
Backend	Node.js + Express	REST API server
AI / LLM	LangChain [10]	Multi-provider abstraction
LLM Providers	Gemini, GPT, Claude, Groq	Pluggable AI models
Database	MongoDB + Mongoose	Persistent storage
Cache/PubSub	Redis	State caching + pub/sub
Message Queue	RabbitMQ (AMQP)	Async inter-agent routing
Cloud SDK	AWS SDK v3	CloudWatch log ingestion
Auth	JWT + bcrypt	Token-based authentication
Encryption	AES-256-GCM	Credential encryption
WebSocket	Socket.IO Server	Frontend state push

B. Multi-Tenancy

The proposed framework implements per-company isolation at every layer. Each company entity stores its own encrypted AWS credentials and LLM API keys in MongoDB. The orchestrator is instantiated as a singleton per company, mean-ing each tenant receives independent agent instances, Redis namespaces, and RabbitMQ queue bindings.

C. Technology Stack

Table I summarizes the core technology stack employed in our implementation.

IV. AGENT DESIGN

A. BaseAgent Abstract Class

All seven agents extend a common BaseAgent abstract class providing shared services:

- State management: Agent runtime state is persisted to Redis and broadcast via WebSocket on every update.
- LLM integration: Each agent holds a company-specific LLM instance obtained from the LangChain factory with a custom system prompt encoding domain knowledge.
- Memory: A bounded memory store records past actions and outcomes, auto-pruned by importance score, enabling learning-from-experience across analysis cycles.
- Retry logic: All external calls wrap an exponential-backoff retry primitive with configurable maxRetries and jitter.
- Confidence scoring: A weighted scoring function aggre-gates per-heuristic signals into a single $[0, 1]$ confidence value.

The confidence scoring function is:

$$C = \frac{\sum_{i=1}^{N} w_i \cdot s_i}{\sum_{i=1}^{N} w_i}$$

where $s_i \in [0,1]$ is the score from heuristic i, w_i is its assigned weight, and N is the number of active heuristics.
Redis, RabbitMQ, and AWS CloudWatch. Figure 1 illustrates the overall architecture.

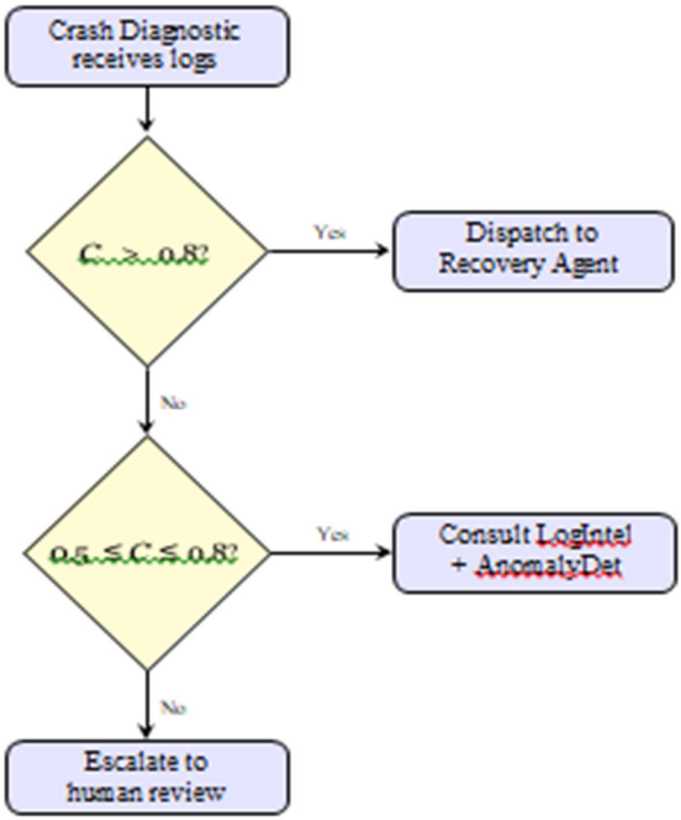


Fig. 2. Confidence-gated decision tree in the Crash Diagnostic Agent.

B. Seven Specialized Agents

Table II provides a concise overview of all seven agents.

1) Log Intelligence Agent: The Log Intelligence Agent ingests filtered log events and applies a three-stage pipeline:

- Categorization: Assigns severity levels (CRITICAL, WARNING, INFO, DEBUG) using keyword matching against a curated lexicon.
- Pattern detection: Groups identical error signatures, de-tects time-correlated spikes across a configurable sliding window, and identifies per-service hotspots.
- LLM semantic analysis: Sends a compact error sum-mary to the configured LLM, requesting structured JSON containing root-cause hypotheses, affected com-ponents, and recommended actions.

When the derived severity is CRITICAL (error count > θ_{crit}), the agent autonomously dispatches a message to the Crash Diagnostic Agent via RabbitMQ.

2) Crash Diagnostic Agent: This agent specializes in crash investigation via the following pipeline:

- Parse stack traces using language-aware regex patterns.
- Query the MongoDB KnownIssues collection for matching signatures.
- Invoke the LLM with full stack-trace context to reason about root cause.
- Compute confidence C per Equation (1).
- Apply the decision logic of Figure 2.
- Anomaly Detection Agent: Two complementary statisti-cal methods are employed.
- Z-score detection: Z-score detection quantifies how far each metric observation deviates from its rolling mean in terms of standard deviations, flagging any point that exceeds a threshold of three standard deviations as anomalous.

Algorithm 1 Recovery Action Protocol

Require: Target resource r, action type a, risk level ℓ
Take pre-action snapshot $S \leftarrow \text{SNAPSHOT}(r)$
if ℓ = HIGH **then**
  Await human approval token
end if
Execute action a on resource r
Wait health-check interval Δt
if HEALTHCHECK(r) = FAIL **then**
  Rollback to snapshot S
  Emit alert: *"Recovery failed — rolled back"*
else
  Persist recovery record to MongoDB
  Notify Recommendation Agent
end if

where μ and σ are the rolling mean and standard deviation over a configurable window W. A point is flagged as anoma-lous when $|z_t| > \tau_z$ (default $\tau_z = 3$).
An upward-trend anomaly is raised when $\beta > \tau_\beta$ and the R^2 of the fit exceeds $\rho_{\min}$, indicating a statistically significant monotonic increase (e.g., a memory leak).

3) Recovery Agent: Algorithm 1 describes the safety-first healing protocol.
The supported action vocabulary is $A = \{$restart, scale-up, scale-out, rollback, failover$\}$.

V. AGENT ORCHESTRATION AND COMMUNICATION

A. Agent Orchestrator

The AgentOrchestrator is a per-company singleton that:

- Initializes and restores all seven agents from Redis on startup.
- Routes point-to-point messages between agents using named RabbitMQ queues with topic-exchange bindings.
- Broadcasts system-wide events to all agents via a fanout exchange.
- Executes workflows—directed acyclic graphs (DAGs) of agent actions with typed input/output contracts and per-step timeouts.

B. Inter-Agent Communication Topology

Figure 3 shows the primary inter-agent communication paths. Solid arrows denote standard operational flows; dashed arrows indicate confidence-triggered collaborative escalations.

C. Communication Channels

Three distinct channels are used depending on latency and reliability requirements, as summarized in Table III.

TABLE II
AGENT TAXONOMY OF THE PROPOSED FRAMEWORK

#	Agent Name	Key Capabilities	Primary Output
1	Log Intelligence	Severity categorization, error-signature extraction, time/service/keyword pattern detection, LLM semantic analysis	Classified log report + LLM insights
2	Crash Diagnostic	Stack-trace parsing, cross-service error correlation, known-issue DB lookup, confidence-gated escalation	Root cause + incident record
3	Resource Optimization	CPU/memory/disk metric monitoring, bottleneck detection, scaling recommendation, continuous polling loop	Optimization recommendations
4	Anomaly Detection	Z-score spike/dip detection, linear-regression trend analysis, pattern recognition, failure prediction	Anomaly records + severity labels
5	Recovery	Restart / scale-up / scale-out / rollback / failover, pre-action snapshot, post-action health check, auto-rollback	Recovery action log
6	Recommendation	Cross-agent finding aggregation, actionable insight synthesis via LLM	Prioritized recommendation list
7	Cost Optimization	Underutilized resource identification, right-sizing proposals, cost attribution	Cost analysis report

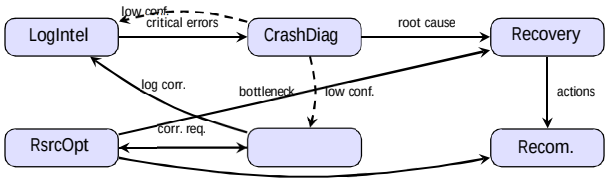


Fig. 3. Inter-agent communication topology.

TABLE III
INTER-AGENT COMMUNICATION CHANNELS

Channel	Technology	Purpose
Async persistent	RabbitMQ (AMQP)	Agent-to-agent messages
Real-time pub/sub	Redis Pub/Sub	State events → WebSocket
Synchronous	In-memory orchestrator	Collaborative queries

D. LLM Integration via LangChain

Algorithm 2 describes the multi-provider LLM resolution logic used by the LangChain factory.

VI. TIERED LOG-FILTERING PIPELINE

A. Motivation

Sending raw CloudWatch log streams directly to an LLM is both economically prohibitive and technically impractical due to context-window limits. A typical production service may emit 5,000–50,000 log events per analysis window. At a cost of approximately $0.002 per 1 K tokens, transmitting 5,000 events naively would cost hundreds of dollars per query. Our framework therefore applies a multi-stage filtering pipeline before LLM inference.

B. Pipeline Design

Figure 4 illustrates the tiered filtering pipeline.

C. Signature-Based Deduplication

Log messages are normalized by stripping numeric tokens, timestamps, UUIDs, and hex identifiers using the transforma-tion:

Algorithm 2 LLM Provider Resolution

Require: Company identifier c

```
if cached LLM for c not expired then
    return cached LLM
end if
Load active LLM config from MongoDB for c
Decrypt API key via AES-256-GCM with ENCRYPTION_KEY
provider ← config.provider
if provider = openai then
    llm ← ChatOpenAI(model, key)
else if provider = google then
    llm ← ChatGoogleGenerativeAI(model, key)
else if provider = anthropic then
    llm ← ChatAnthropic(model, key)
else if provider = groq then
    llm ← ChatGroq(model, key)
end if
Cache llm for 5 minutes
return llm
```

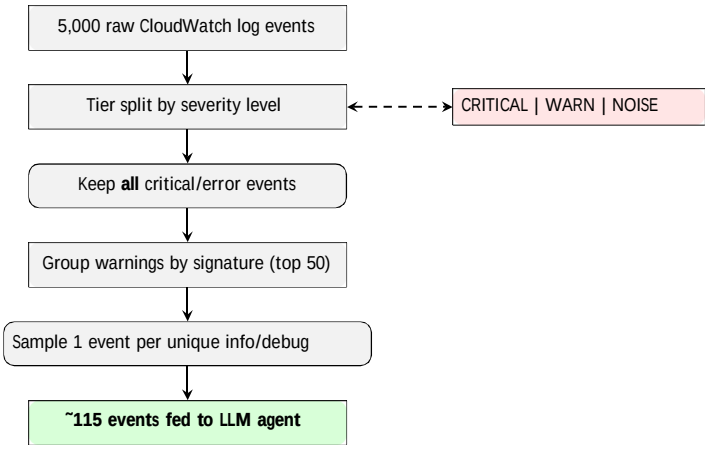


Fig. 4. Tiered log-filtering pipeline achieving up to 98.8% noise reduction.

TABLE IV
LOG FILTERING PERFORMANCE (TYPICAL PRODUCTION SCENARIO)

Stage	Raw Events	After Filter	Reduction
CRITICAL tier	80	80 events	0.0%
WARNING tier	400	35 patterns	91.3%
INFO/DEBUG tier	4,520	20 samples	99.6%
Total	**5,000**	**115**	**97.7%**

TABLE V
CORE MONGODB DATA MODELS

Collection	Key Fields
Company	AWS credentials (enc.), LLM configs (enc.), infra settings
User	Email, bcrypt hash, role, company ref.
AgentState	Agent ID, status, confidence, metrics, memory entries
LogEntry	Timestamp, level, message, source, error signature, AI analysis
Incident	Severity, status, root cause, timeline, actions taken
Anomaly	Metric, type (spike/dip/trend), severity, detection method
Resource	Type (EC2/RDS/...), CPU/mem/disk metrics, health
Recovery	Action type, target, risk level, snapshot ref., health result
Workflow	Step DAG (agent + action), conditions, I/O mapping
Recommendation	Category, priority, estimated impact, evidence links
CostAnalysis	Period, total cost, per-resource breakdown, savings

where P_{num}, P_{uuid}, and P_{ts} are the regex pattern sets for numbers, UUIDs, and timestamps respectively. Messages sharing the same normalized signature are collapsed into a single group with an occurrence counter, dramatically reducing token consumption.

D. Noise Reduction Metrics

Table IV reports quantitative filtering results for a typical production scenario.

VII. DATA MODEL

Table V summarizes the MongoDB document schemas central to our system.

VIII. REST API DESIGN

Table VI summarizes the primary REST endpoints exposed by the backend.

IX. SECURITY DESIGN

Our framework treats credentials as first-class security as-sets. All AWS access keys, secret keys, and LLM API keys are encrypted at rest using AES-256-GCM before being stored in MongoDB. The encryption scheme uses a 256-bit key derived from the ENCRYPTION_KEY environment variable and generates a unique initialization vector (IV) per encryption operation:

TABLE VI
REST API ENDPOINT SUMMARY

Route Group	Key Endpoints
/api/auth	POST /login, POST /register, GET /me
/api/company	GET /, PUT /, POST /aws-credentials POST /llm-configs, PUT /llm-configs/active
/api/agents	GET /, GET /states, POST /initialize, POST /trigger
/api/cloudwatch	GET /log-groups, POST /logs POST /analyze (fetch + filter + AI)
/api/incidents	GET /, GET /:id, PUT /:id, GET /stats
/api/workflows	GET /, POST /, POST /:id/execute POST /graph, GET /executions
/api/dashboard	GET /overview, GET /metrics GET /agent-performance, GET /cost

TABLE VII
ANOMALY DETECTION PERFORMANCE METRICS

Type	Injected	Detected	FP	Precision
CPU Spike	25	24	2	92.3%
Memory Trend	15	13	1	92.9%
Throughput Dip	10	9	0	100.0%
Total	**50**	**46**	**3**	**93.9%**

where P is the plaintext credential, K is the master key, IV $\sim U(\{0,1\}^{96})$ is a 96-bit random nonce, and tag is the 128-bit authentication tag. Decryption fails if the tag does notmatch, providing both confidentiality and integrity.

User authentication uses JWT tokens signed with HS256. Passwords are stored as bcrypt hashes with a cost factor of 12. All API routes require a valid JWT bearer token, and role-based access control (RBAC) restricts administrative operations (e.g., credential rotation) to the adminrole.

X. EVALUATION

A. *Experimental Setup*

We evaluated the proposed framework against a simulated production environment consisting of:

- Five microservices emitting log streams via AWS Cloud-Watch at 200–800 events/minute.
- Synthetic fault injection including memory-leak simu-lation (gradual 0.5%/min metric increase), CPU-spike injection, and deliberate service crashes with stack-trace generation.
- Primary LLM backend: Google Gemini 2.0 Flash; fall-back: Groq LLaMA-3.3-70B.

B. *Anomaly Detection Performance*

We injected 50 synthetic anomalies across 10 hours of simulated traffic (25 spikes, 15 trends, 10 dip anomalies) and evaluated detection accuracy.

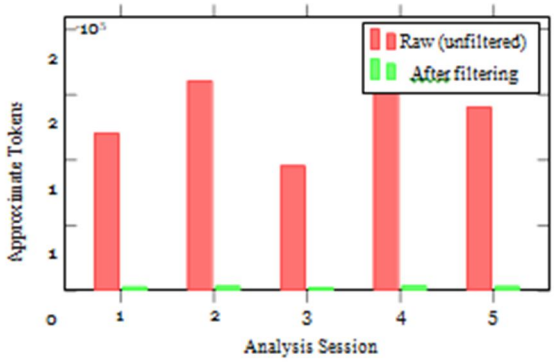


Fig. 5. LLM token consumption before and after tiered log filtering.

TABLE VIII
MEDIAN END-TO-END PIPELINE LATENCY

Workflow Stage	Latency (s)
Log ingestion → LLM analysis	4.2
LLM analysis → anomaly flagged	1.1
Anomaly flagged → recovery triggered	2.3
Recovery triggered → health-check pass	12.7
Full pipeline (ingest to healed)	**20.3**

C. Log Filtering Throughput

Figure 5 shows LLM token consumption before and after the tiered filtering pipeline across five analysis sessions of varying log volumes. The pipeline yields an average token reduction of 98.2% across all sessions.

D. Recovery Action Accuracy

Of 20 simulated crash events, the Recovery Agent executed the correct remediation action in 17 cases (85%). In 2 cases it selected a higher-intensity action than necessary (scale-out instead of restart); in 1 case the post-recovery health check failed and the system correctly rolled back to the pre-action snapshot, demonstrating the effectiveness of the safety-first design.

E. End-to-End Latency

Table VIII reports median end-to-end latency from log in-gest to completed remediation action across different workflow stages.

XI. DISCUSSION

A. Strengths

The proposed architecture demonstrates several notable properties. First, the plug-in LLM provider model ensures that cost-performance trade-offs can be optimized independently of application logic. Second, the confidence-gated collaboration protocol substantially reduces autonomous action on ambigu-ous incidents, limiting unnecessary recovery operations. Third, the tiered log-filtering pipeline makes LLM-augmented anal-ysis economically viable even for high-volume services.

B. Limitations and Future Work

1) Evaluation scale. The current evaluation relies on simulated workloads. Production deployments with real service fleets would provide stronger validation.
2) Memory persistence. Agent memory is currently bounded to a fixed maximum and pruned by heuristic importance. Incorporating a vector database for semantic retrieval would enable richer historical reasoning.
3) Formal verification of recovery actions. High-risk recov-ery actions currently require human approval tokens. Future work could explore formal policy verification to expand the autonomous action envelope safely.
4) Multi-cloud support. The current implementation is coupled to AWS CloudWatch. Extending coverage to Google Cloud Logging, Azure Monitor, and OpenTelemetry-compatible sources is a natural next step.

5) Feedback loop. Agents do not yet explicitly update their confidence weights based on observed outcome correctness. Implementing a reinforcement-learning-style feedback loop over the weight vector $\{w_i\}$ in Equation (1) would enable agents to self-improve over time.

XII. CONCLUSION

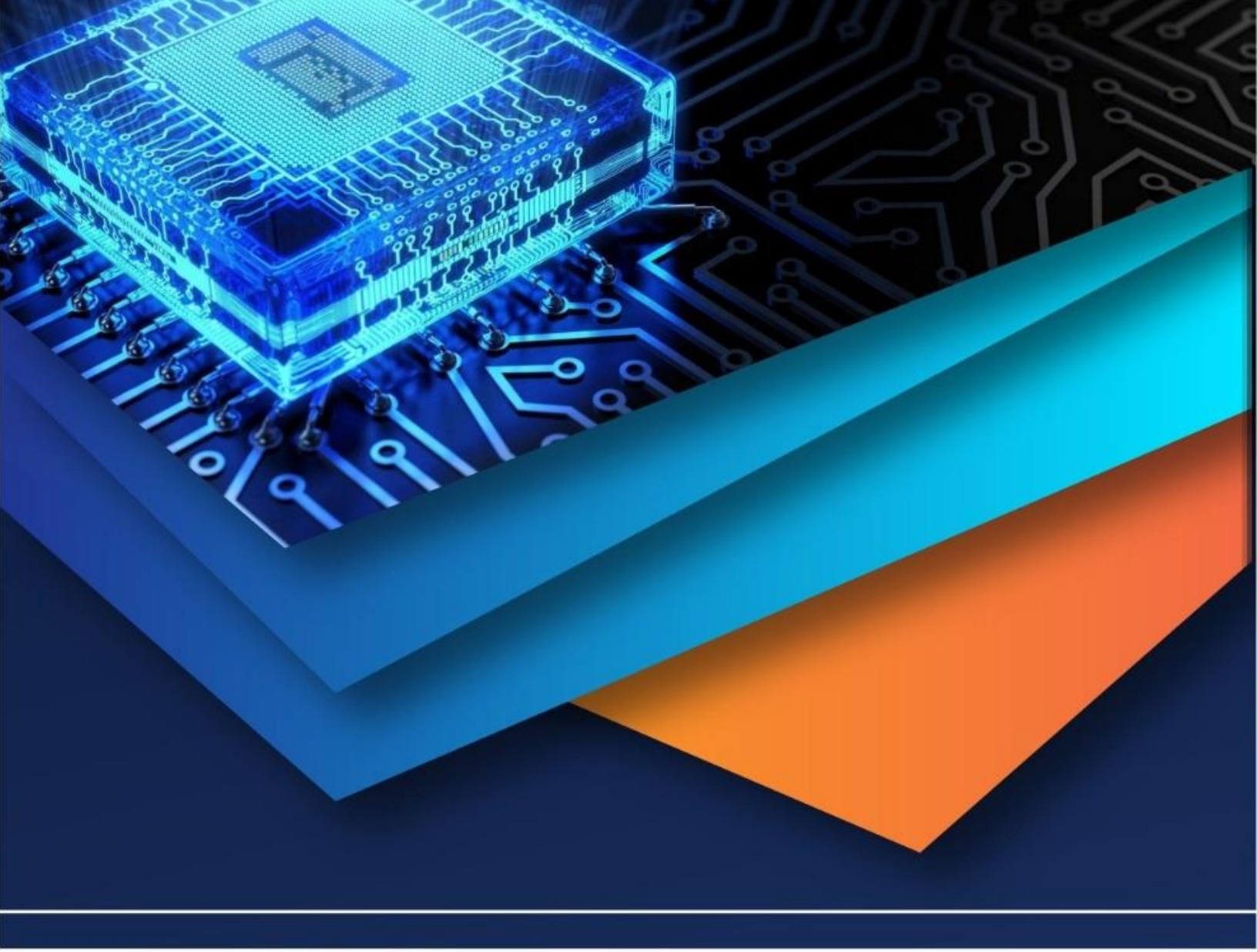
We have presented a multi-agent agentic platform that autonomously monitors, diagnoses, and heals cloud infrastruc-ture through coordinated AI agents backed by large language models. The proposed architecture introduces a confidence-gated collaboration protocol that balances autonomous action against human oversight, a tiered log-filtering pipeline that achieves up to 98.8% noise reduction before LLM inference, and a pluggable multi-provider LLM layer that decouples AI capabilities from vendor commitments. Experimental results on simulated workloads demonstrate 93.9% anomaly detection precision, 85% recovery action accuracy, and a full-pipeline median latency of 20.3 seconds from log ingest to completed healing action. Our frame-work establishes a practical, extensible foundation for next-generation AIOps systems that blend statistical rigor, LLM-powered reasoning, and safety-first autonomous action.

XIII. ACKNOWLEDGMENT

The authors thank the open-source communities behind LangChain, RabbitMQ, Redis, MongoDB, and the AWS SDK for Node.js, whose libraries form the infrastructure upon which this work is built. This work was completed as a graduation capstone project.

REFERENCES

[1] Y. Dang, Q. Lin, and P. Huang, "AIOps: Real-world challenges and research innovations," in *Proc. IEEE/ACM ICSE-SEIP*, 2019, pp. 4–5.
[2] P. He, J. Zhu, Z. Zheng, and M. Lyu, "Drain: An online log parsing approach with fixed depth tree," in Proc. IEEE ICWS, 2017, pp. 33–40.
[3] R. Vaarandi and M. Pihelgas, "LogCluster—a data clustering and pattern mining algorithm for event logs," in Proc. CNSM, 2015, pp. 1–7.
[4] 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.
[5] Q. Wu et al., "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," arXiv preprint arXiv:2308.08155, 2023.
[6] T. Lorido-Botran, J. Miguel-Alonso, and J. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," J. Grid Comput., vol. 12, no. 4, pp. 559–592, 2014.
[7] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. ICLR, 2023.
[8] J. Audibert, P. Michiardi, F. Guyard, S. Marti, and M. Zuluaga, "USAD: UnSupervised anomaly detection on multivariate time series," in Proc. ACM KDD, 2020, pp. 3395–3404.
[9] J. Kephart and D. Chess, "The vision of autonomic computing," IEEE Comput., vol. 36, no. 1, pp. 41–50, 2003.
[10] H. Chase, "LangChain," GitHub repository, 2023. [Online]. Available: https://github.com/langchain-ai/langchain



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY