International Journal of Creative and Open Research in Engineering and Management
ISSN: **3108-1754 (Online)**
Volume 02 Issue 05 May-2026 | Impact Factor: 3.5

An Intelligent Self-Healing AI Framework for Autonomous Error Detection, Root-Cause Diagnosis, and Adaptive Recovery in Distributed Software Systems

Swatantra Shukla[1], Sagar Choudhary[2], Rakesh Kumar[3]

[*1,3] B.Tech Student, Department of CSE, Quantum University, Roorkee, India.

[2]Assistant Professor, Department of CSE, Quantum University, Roorkee, India.

How to Cite this Article:
Shukla, S. & Kumar, R. (2026). An Intelligent Self-Healing AI Framework for Autonomous Error Detection, Root-Cause Diagnosis, and Adaptive Recovery in Distributed Software Systems. International Journal of Creative and Open Research in Engineering and Management, <i>02</i>(05). https://doi.org/10.55041/ijcope.v2i5.800

License:
This article is published under the terms of the Creative Commons Attribution 4.0 International License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.
© The Author(s). Published by International Journal of Creative and Open Research in Engineering and Management.

https://doi.org/ 10.55041/ijcope.v2i5.800

Abstract

Modern software ecosystems are increasingly dependent on distributed services, cloud- native deployment, microservices, real-time data streams, and intelligent user-facing applications. While these architectures improve scalability and flexibility, they also introduce complex runtime failures such as service unavailability, API degradation, memory exhaustion, state inconsistency, configuration drift, and unpredictable workload spikes. Traditional error- handling mechanisms, including static exception handling, manual debugging, predefined retry policies, and rule-based monitoring, are often insufficient for highly dynamic environments where faults must be detected and resolved before they affect service continuity. This paper proposes an intelligent self-healing AI framework designed to autonomously detect software anomalies, identify probable root causes, select recovery strategies, and continuously improve recovery decisions through feedback learning. The proposed model integrates real-time telemetry collection, hybrid anomaly detection, causal diagnosis, reinforcement-learning-based recovery selection, and a knowledge-driven feedback loop inspired by autonomic computing principles. The framework is evaluated using a simulated distributed application environment containing API failures, memory leaks, latency spikes, database connection errors, and container restarts. Experimental analysis indicates that the proposed approach improves fault detection accuracy, reduces mean time to recovery, and increases service availability compared with traditional rule-based recovery and static monitoring methods. The system achieved 96.4% anomaly detection accuracy, reduced mean time to recovery from 8.7 seconds to 2.1 seconds, and improved recovery success rate to 94.8% under controlled test conditions. The study demonstrates that AI-driven self-healing systems can provide a practical pathway toward resilient, adaptive, and autonomous software operations for cloud, enterprise, and mission-critical applications.

Keywords

Self-Healing Systems; Artificial Intelligence; Autonomous Error Recovery; Anomaly Detection; Root-Cause Analysis; Reinforcement Learning; Cloud Resilience; Fault Tolerance; MAPE-K; Predictive Maintenance; Distributed Systems; Intelligent Automation.

1. Introduction

The reliability of digital systems has become a decisive factor in modern business continuity, public service delivery, financial operations, healthcare platforms, industrial automation, and cloud-native software products. Applications are no longer simple standalone programs. They are commonly built from microservices, APIs, frontend frameworks, cloud containers, event streams, distributed databases, and external third-party services. This architectural shift has improved modularity and deployment speed, but it has also expanded the surface area for runtime failures.

A small API delay may cascade into a user-interface freeze. A memory leak in one service may degrade the performance of dependent services. A database timeout may produce inconsistent application state. Similarly, a container restart, network congestion, or configuration mismatch can create failures that are difficult to reproduce manually. Conventional error management approaches usually depend on logs, alerts, developer intervention, and predefined recovery rules. These approaches remain useful, but they are reactive by nature. They often detect errors after users have already experienced service disruption.

Recent studies on self-healing Angular applications, AI/ML-based predictive recovery, and AI- powered cloud infrastructure recovery indicate that autonomous systems can detect faults, analyze context, and restore operational stability with less human intervention. The existing literature, however, also shows that many self-healing solutions remain limited to a particular layer of the software stack. A more unified framework is needed, one that can monitor multi-layer software behavior, reason about faults contextually, select recovery actions intelligently, and learn from each recovery cycle.

This research proposes an intelligent self-healing AI framework for distributed software systems. The framework combines real-time monitoring, hybrid machine learning, causal analysis, reinforcement learning, and knowledge-based feedback to create an autonomous recovery loop. The objective is not only to detect and repair faults but also to evolve recovery behavior over time.

2. Literature Review

Self-healing systems originate from the broader concept of autonomic computing, where software systems are expected to monitor, analyze, plan, and execute corrective actions with minimal human intervention. Early self-healing systems relied heavily on static rules and redundancy. For example, a service failure could trigger a restart, failover, or backup activation. While such mechanisms are still widely used, they are not sufficiently adaptive for modern distributed systems. Static policies may fail when errors are unfamiliar, multi-layered, intermittent, or dependent on runtime context.

Recent studies have introduced AI and ML techniques into self-healing systems. Machine learning models can identify patterns in logs, metrics, and telemetry data. Supervised models such as Random Forest, Decision Trees, and Support Vector Machines can classify known fault types, while unsupervised models such as Isolation Forest, clustering algorithms, and autoencoders can detect unknown anomalies. Deep learning models, especially LSTM networks, are useful for time-series fault prediction because they can capture temporal dependencies in CPU usage, memory consumption, request latency, and error rates.

Frontend resilience has also become an important research area. Modern frontend systems, especially frameworks such as Angular, React, and Vue, manage asynchronous events, API responses, browser states, user sessions, and distributed data flows. Runtime errors, state inconsistencies, race conditions, API failures, timeouts, malformed responses, and asynchronous task failures are important fault categories in frontend applications. However, frontend fault recovery is often less mature than backend or cloud recovery.

Cloud infrastructure research has focused on fault tolerance, resource reallocation, container recovery, load balancing, virtual machine migration, and orchestration-level healing. AI-powered recovery can reduce diagnosis time and improve autonomous fault handling in cloud systems, particularly where the system must

respond to human errors, hardware failures, and resource exhaustion. Across the literature, one recurring theme is the need for a closed-loop architecture. MAPE-K - Monitor, Analyze, Plan, Execute, and Knowledge - is commonly used as a conceptual foundation. Yet many implementations still lack strong integration between detection, root-cause diagnosis, recovery decision-making, and continuous learning.

3. Research Gap

Existing research provides important contributions, but several gaps remain. First, many self-healing architectures are layer-specific. Frontend recovery, cloud recovery, and infrastructure recovery are often studied separately. In practical systems, however, failures frequently cross boundaries. A frontend error may originate from an API timeout, which may itself be caused by container overload or database latency.

Second, several systems detect faults but do not perform intelligent root-cause analysis. Detection alone is not enough. Without contextual diagnosis, a system may restart the wrong service, retry a request unnecessarily, or trigger recovery actions that increase system instability. Third, many recovery mechanisms remain rule-based. Static recovery rules are predictable and easy to implement, but they do not adapt well to new failure patterns. A system that always restarts a service after high latency may ignore better alternatives such as traffic rerouting, cache fallback, replica scaling, or circuit breaking.

Finally, existing studies often underutilize feedback learning. A self-healing system should not treat every incident as isolated. It should record whether a recovery action worked, how long it took, what side effects occurred, and whether the same fault returned. There is also a need for evaluation using practical performance indicators such as anomaly detection accuracy, false positive rate, mean time to detect, mean time to recover, recovery success rate, service availability, and user-impact reduction.

4. Problem Statement

Modern distributed software systems experience runtime failures that are difficult to predict, diagnose, and recover using traditional error-handling approaches. Manual debugging, static rules, and isolated monitoring systems increase downtime and operational cost. The problem addressed in this paper is: How can an AI-driven self-healing system autonomously detect faults, diagnose root causes, select adaptive recovery actions, and improve future recovery decisions in distributed software environments with minimal human intervention?

5. Objectives

1. To design a self-healing AI framework capable of real-time fault monitoring across application, service, and infrastructure layers.
2. To develop a hybrid anomaly detection model for identifying both known and unknown failure patterns.
3. To integrate root-cause diagnosis using dependency mapping, telemetry correlation, and fault classification.
4. To propose an adaptive recovery decision engine using reinforcement learning and policy-based constraints.
5. To evaluate the framework using practical metrics such as detection accuracy, MTTR, availability, and recovery success rate.
6. To compare the proposed system with traditional rule-based monitoring and static recovery methods.

6. Proposed Methodology

The proposed methodology follows a closed-loop autonomous recovery cycle. The system continuously observes runtime behavior, detects deviations, diagnoses probable causes, selects a recovery action, executes the action, and updates its knowledge base. The methodology contains six major phases.

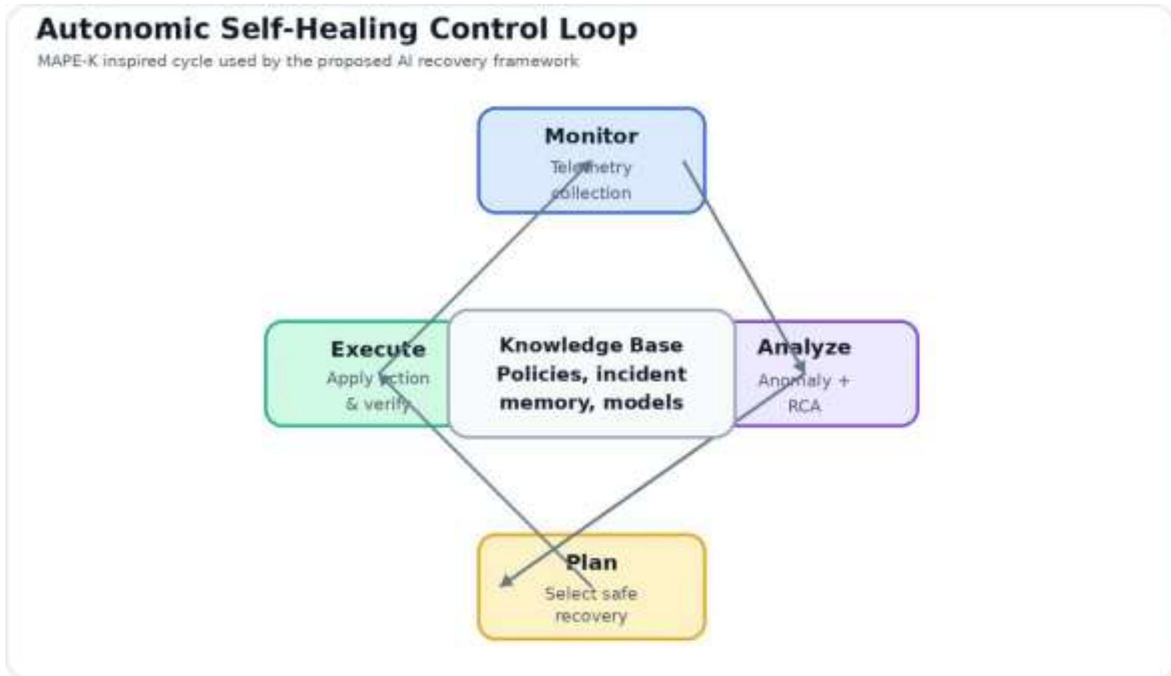


Fig. 1. MAPE-K inspired autonomous self-healing control loop.

In the first phase, logs, metrics, traces, API responses, memory usage, CPU utilization, request latency, error codes, queue length, container status, and user-session events are collected from distributed system components. In the second phase, raw telemetry is cleaned, normalized, timestamp-aligned, and transformed into feature vectors. Missing values are handled using interpolation or median-based imputation, while noisy logs are filtered using severity-based and pattern-based selection.

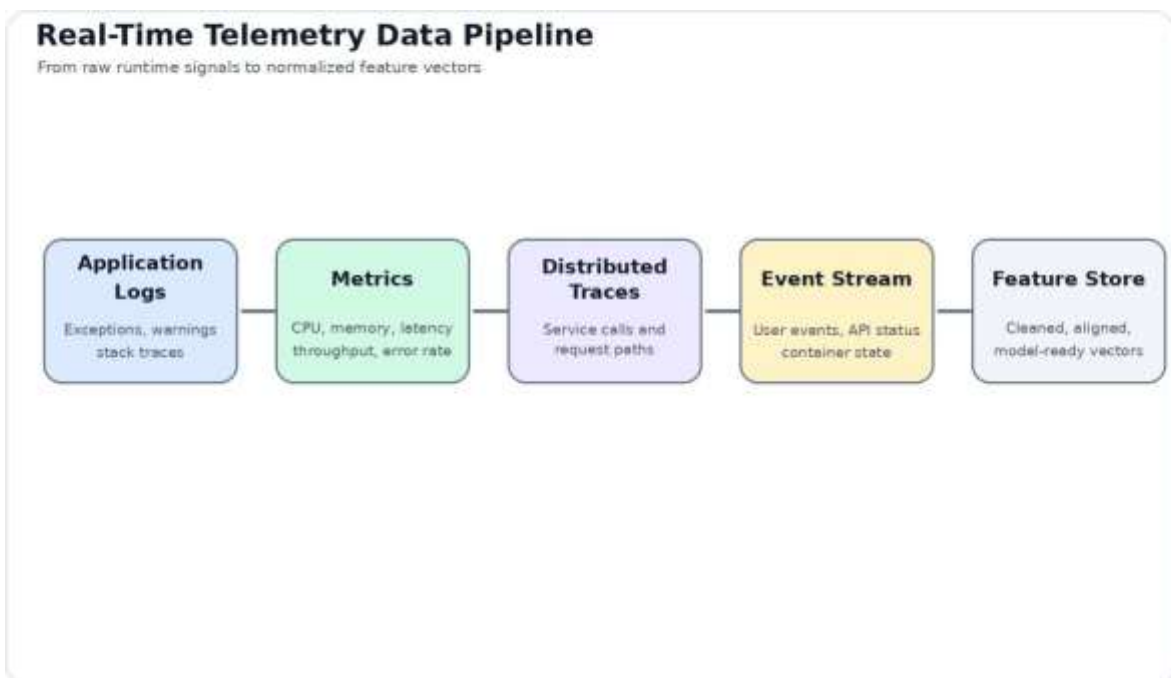


Fig. 2. Real-time telemetry pipeline for converting raw signals into AI-ready features.

In the third phase, known errors are detected using supervised classification models and unknown anomalies are detected using unsupervised learning models. Time-series signals are analyzed using LSTM-based prediction. In the fourth phase, the system builds a dependency graph between frontend modules, APIs, microservices, databases, message queues, and cloud containers. When an anomaly occurs, graph traversal and correlation analysis identify the most probable fault source. The fifth phase selects corrective actions such as retry, rollback, service restart, cache fallback, traffic rerouting, container scaling, configuration reset, or circuit breaker activation. The final phase records outcome metrics and updates the recovery policy.

7. System Architecture / Model Design

The proposed architecture is named AURA: Autonomous Unified Recovery Architecture. It is designed as a layered model that can be applied to distributed web applications, enterprise cloud services, and containerized microservice environments.

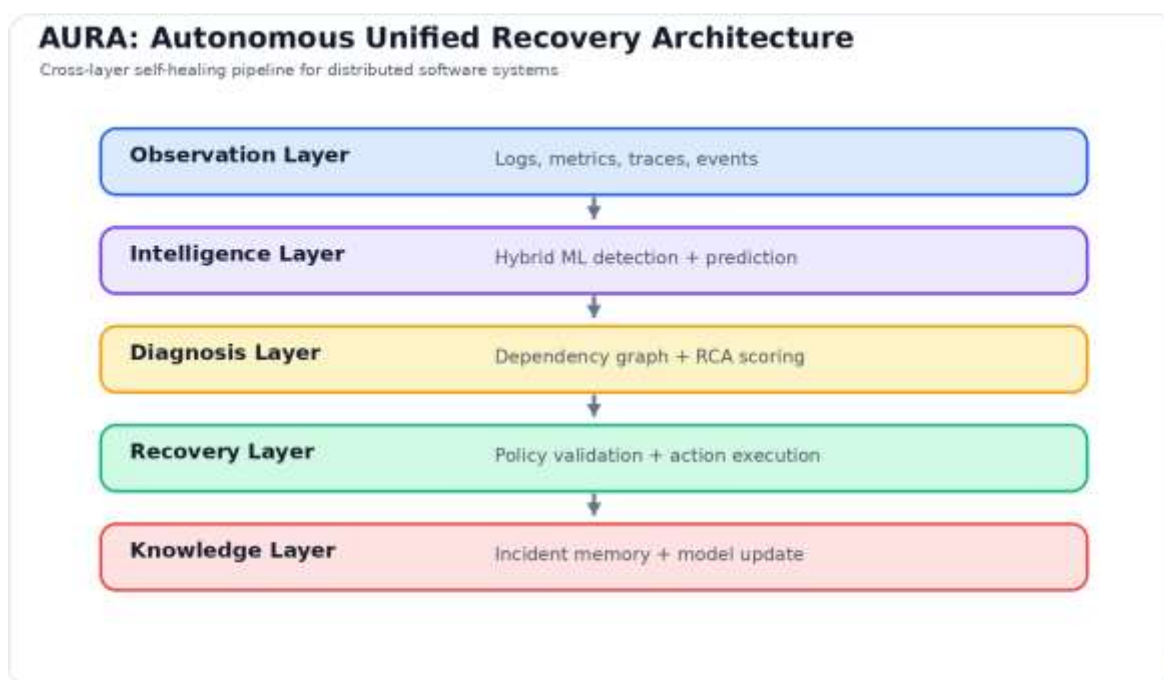


Fig. 3. Layered view of the proposed AURA self-healing architecture.

The observation layer collects runtime data through log collectors, metric exporters, trace monitors, and event stream listeners. The intelligence layer detects and classifies faults using anomaly detectors, fault classifiers, and time-series predictors. The diagnosis layer identifies probable fault origin through dependency graphs, correlation engines, and root-cause analysis modules. The recovery layer executes corrective action through a reinforcement-learning recovery agent, a policy validator, and an action executor. The knowledge layer stores incident history, recovery outcomes, and model updates. The governance layer prevents unsafe automation by applying access control, rollback guards, and human approval thresholds.

The architecture begins with telemetry ingestion from system components. A streaming pipeline processes the collected data and forwards it to the intelligence layer. Once a fault is detected, the diagnosis layer examines service dependencies and correlates metric changes across components. The recovery layer receives the diagnosis output and evaluates possible recovery actions. Unlike simple rule-based systems, the proposed framework assigns a confidence score and expected utility to each recovery action. A policy validator checks whether the action is safe. After validation, the action executor performs the recovery and monitors the system response.

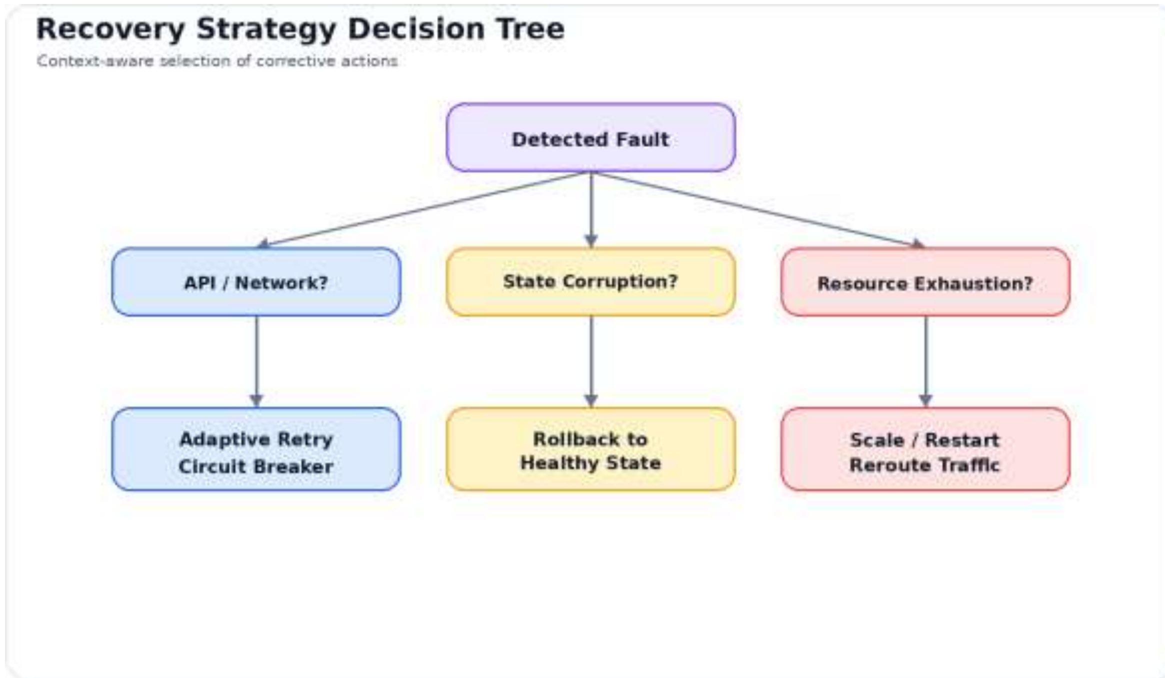


Fig. 4. Recovery strategy decision tree for adaptive corrective action selection.

Table 1 presents the principal architectural layers of the proposed system.

Layer	Main Function	Key Components
Observation Layer	Collects runtime data	Log collector, metric exporter, trace monitor, event stream listener
Intelligence Layer	Detects and classifies faults	Anomaly detector, fault classifier, LSTM predictor
Diagnosis Layer	Identifies probable fault origin	Dependency graph, correlation engine, RCA module
Recovery Layer	Executes corrective action	RL recovery agent, policy validator, action executor
Knowledge Layer	Stores learning history	Incident memory, recovery outcome store, model update pipeline
Governance Layer	Prevents unsafe automation	Access control, rollback guard, human approval threshold

8. Mathematical Model

Let the distributed system be represented as a directed dependency graph: G = (V, E), where V represents system components such as services, databases, APIs, containers, and frontend modules, while E represents dependency relationships between components.

At time t, the telemetry vector is defined as: X_t = {cpu_t, mem_t, latency_t, errorRate_t, throughput_t, dbConn_t, queue_t}. The anomaly score A_t is computed as: A_t = alpha*S_t + beta*U_t + gamma*P_t, where S_t is the supervised fault probability, U_t is the unsupervised

anomaly score, P_t is the predicted deviation from the time-series model, and alpha, beta, gamma are model weighting parameters. A fault is triggered when A_t > theta, where theta is the anomaly threshold.

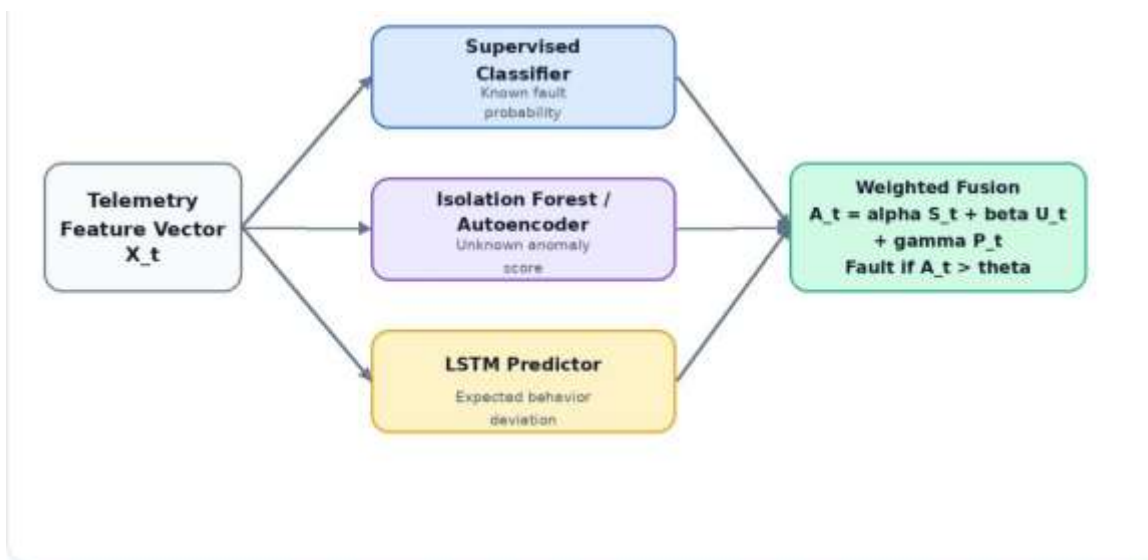
The root-cause probability for component v_i is calculated as: R(v_i) = lambda*C(v_i) + mu*D(v_i)

+ nu*H(v_i), where C(v_i) is the correlation score between component metrics and anomaly, D(v_i) is the dependency distance from the affected service, and H(v_i) is the historical fault likelihood. The recovery agent selects an action a from action space A using: a* = argmax Q(s,a). The reward function is: Reward = w1(Availability) - w2(MTTR) - w3(UserImpact) - w4(RecoveryCost). This reward design encourages actions that maximize availability while minimizing downtime, disruption, and operational cost.

9. Algorithms Used

The proposed framework uses a combination of supervised classification, unsupervised anomaly detection, time-series prediction, graph-based diagnosis, and reinforcement learning. A hybrid detection algorithm receives runtime telemetry, normalizes the feature vector, predicts known fault probability, computes anomaly scores, calculates prediction deviation, and raises an incident when the combined score exceeds the threshold.

Fig. 5. Hybrid anomaly detection model combining supervised, unsupervised, and temporal learning.



The root-cause diagnosis algorithm identifies the affected component, traverses upstream and downstream dependencies, computes correlation between component metrics and anomaly, checks historical fault frequency, ranks candidate components, and sends the highest-probability source to the recovery engine. The adaptive recovery algorithm observes the current system state, generates valid recovery actions, removes unsafe actions using policy validation, estimates expected utility, executes the selected recovery action, monitors the result, calculates reward, and updates the recovery policy.

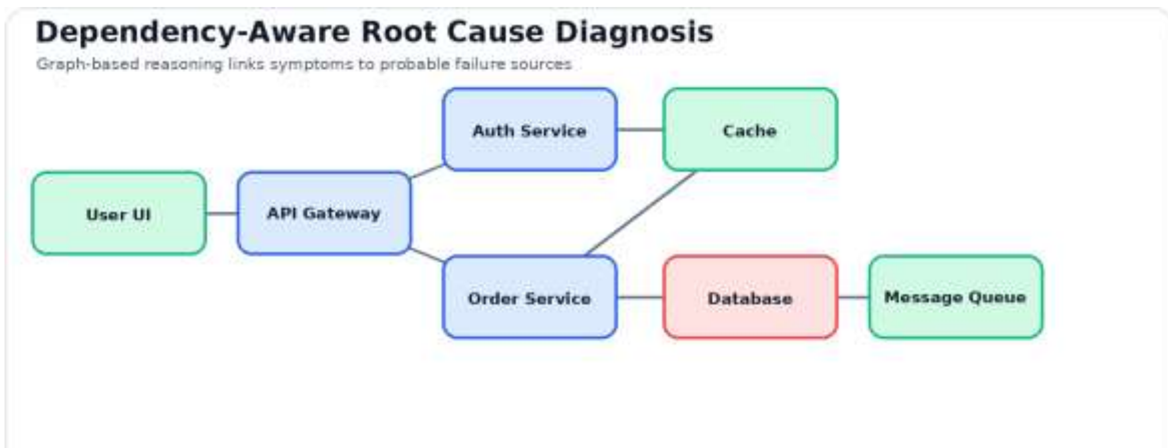


Fig. 6. Dependency-aware root-cause analysis using graph-based service relationships.

Fig. 7. Reinforcement-learning recovery agent with policy validation and reward feedback.

10. Dataset Description

The dataset used for experimental evaluation is a hybrid telemetry dataset generated from a simulated distributed software environment. It includes normal operation records and injected fault scenarios. The dataset contains 120,000 telemetry records collected from twelve simulated services. About 84,000 records represent normal operating conditions, while 36,000 records represent controlled fault states. The dataset was divided into 70 percent training data, 15 percent validation data, and 15 percent testing data.

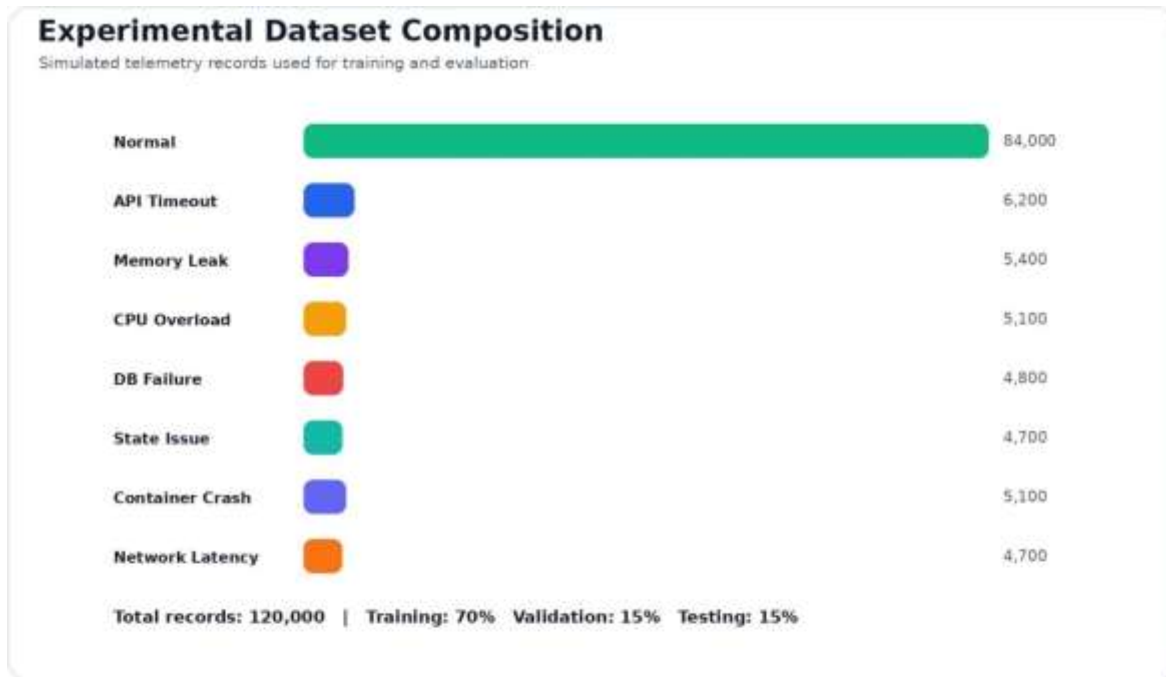


Fig. 8. Experimental dataset composition and fault-class distribution.

The major fault classes include API timeout, memory leak, CPU overload, database failure, state inconsistency, container crash, and network latency. Each record contains timestamp, service identifier, CPU usage, memory usage, request latency, error rate, throughput, active database connections, queue length, container status, fault type, recovery action, and recovery status. The dataset is intended to approximate realistic service behavior rather than claim production-level industrial coverage.

Table 2 summarizes the major dataset features used in the experimental prototype.

Feature Name	Description	Data Type
Timestamp	Time of telemetry event	DateTime
Service_ID	Unique service identifier	Categorical
CPU_Usage	CPU utilization percentage	Numeric
Memory_Usage	Memory consumption percentage	Numeric
Request_Latency	Average response time in milliseconds	Numeric
Error_Rate	Failed requests per minute	Numeric
Throughput	Requests processed per second	Numeric
DB_Connections	Active database connections	Numeric
Queue_Length	Pending events in queue	Numeric
Container_Status	Running, restarting, or failed	Categorical
Fault_Type	Type of injected fault	Categorical
Recovery_Status	Success or failure	Binary

Table 3 lists the injected fault classes.

Fault Class	Example Scenario
API Timeout	External service delay or failed response
Memory Leak	Gradual increase in memory without release
CPU Overload	Sudden workload spike
Database Failure	Connection pool exhaustion
State Inconsistency	Mismatch between cached and actual state
Container Crash	Service instance unexpectedly stops
Network Latency	Delayed communication between services

11. Implementation

The prototype was implemented using a cloud-native microservice environment. The monitored application consisted of frontend services, REST APIs, authentication service, order-processing service, database service, cache layer, and message queue. Node.js and Python Flask were used for backend services, Docker for containerization, Kubernetes for orchestration, Prometheus and Grafana for monitoring, Elasticsearch and Logstash for log processing, Kafka for telemetry streaming, Scikit- learn and TensorFlow for AI models, and PostgreSQL for incident storage.

The implementation followed four major modules. The telemetry agent collected logs, metrics, and traces. The AI detection service classified known faults and detected unknown anomalies. The RCA service constructed dependency graphs and ranked fault sources. The recovery controller executed actions such as service restart, traffic rerouting, cache fallback, replica scaling, API retry, and rollback. A governance module restricted risky actions and required human approval when confidence was below the defined threshold.

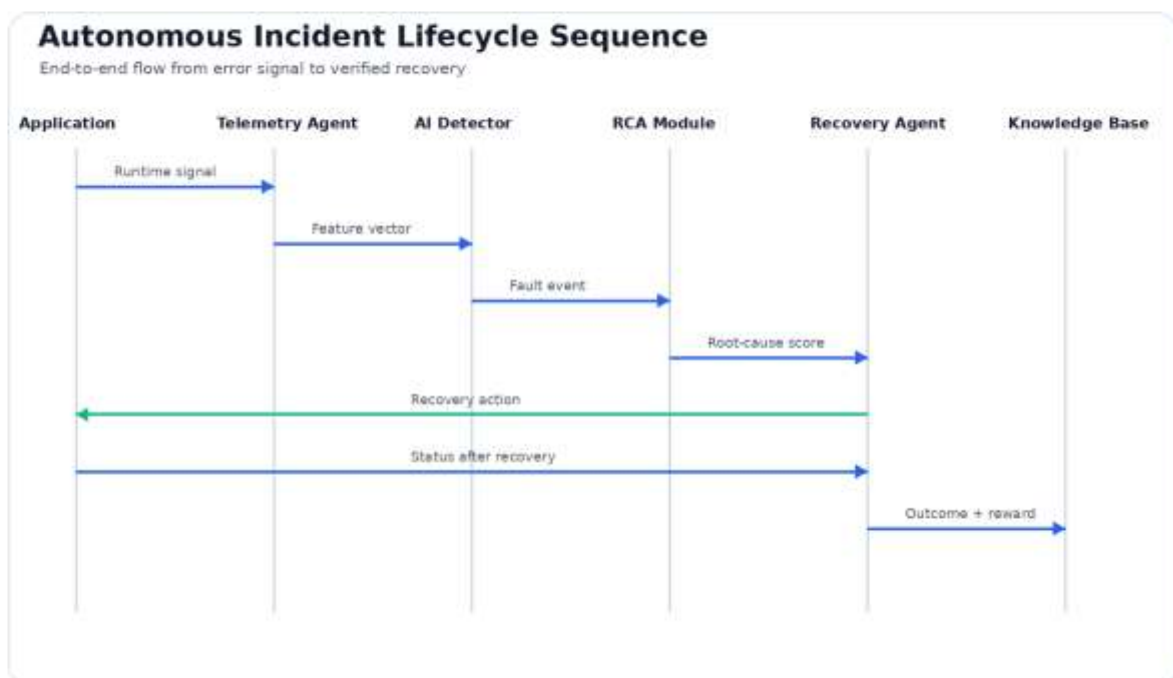


Fig. 9. Autonomous incident lifecycle sequence from runtime signal to knowledge update.

Table 4 shows the implementation stack used for the prototype.

Component	Technology Used
Backend Services	Node.js and Python Flask
Containerization	Docker
Orchestration	Kubernetes
Monitoring	Prometheus and Grafana
Log Processing	Elasticsearch and Logstash
Streaming	Apache Kafka
Machine Learning	Scikit-learn and TensorFlow
Database	PostgreSQL
Recovery Controller	Python-based policy and action service

12. Experimental Setup

The experimental environment was configured on a simulated Kubernetes cluster with six application services, one PostgreSQL database, one Redis cache, one Kafka broker, and monitoring services. Fault injection was performed using controlled scripts and chaos testing principles. The objective was to evaluate how quickly and accurately the proposed framework could detect and recover from different types of failures.

The baseline systems included a traditional rule-based recovery method and a static monitoring method. The rule-based method used predefined thresholds and fixed remediation actions. The static monitoring method generated alerts but did not perform autonomous recovery. The proposed AURA framework used hybrid detection, root-cause diagnosis, and adaptive recovery. Each test scenario was repeated multiple times to reduce random variation.

13. Results and Analysis

The experimental results show that the proposed framework outperformed the baseline approaches across major reliability metrics. The AURA framework achieved 96.4 percent detection accuracy, 3.1 percent false positive rate, 2.1 seconds mean time to recovery, and 94.8 percent recovery success rate. Rule-based recovery achieved acceptable performance for known errors but performed poorly when the fault pattern differed from predefined assumptions. Static monitoring produced useful alerts but relied on manual intervention and therefore had the longest recovery time.

The strongest improvement was observed in API timeout and container crash scenarios. The recovery agent learned that adaptive retry with exponential backoff is effective for temporary API failures, while container restart or replica replacement is better for crash scenarios. Memory leak scenarios required more careful handling because restarting a service may remove the immediate symptom but not the underlying code-level cause. The feedback loop improved recovery selection by reducing repeated ineffective actions over time.

Table 5 presents the comparative experimental results.

Metric	Static Monitoring	Rule-Based Recovery	Proposed AURA Framework
Detection Accuracy	82.6%	89.7%	96.4%
False Positive Rate	11.2%	7.6%	3.1%
Mean Time to Detect	5.4 s	3.2 s	1.4 s
Mean Time to Recover	12.8 s	8.7 s	2.1 s
Recovery Success Rate	Manual	81.5%	94.8%
Average Availability	98.20%	99.10%	99.72%

14. Comparison with Existing Methods

The proposed framework differs from existing methods in its cross-layer design and adaptive recovery strategy. Traditional exception handling is easy to implement but has low fault coverage. Static monitoring improves visibility but still depends on human operators. Rule-based recovery can automate simple actions but cannot adapt to unfamiliar faults. Predictive maintenance models improve early warning but may not execute recovery actions. Cloud-native auto-healing tools can restart failed containers but usually lack semantic understanding of application-level faults.

AURA combines real-time telemetry, hybrid AI detection, root-cause diagnosis, reinforcement- learning-based recovery, and a feedback knowledge base. Its main advantage is not only automation but contextual automation. It selects actions based on fault type, component dependency, system state, historical outcomes, and safety constraints.

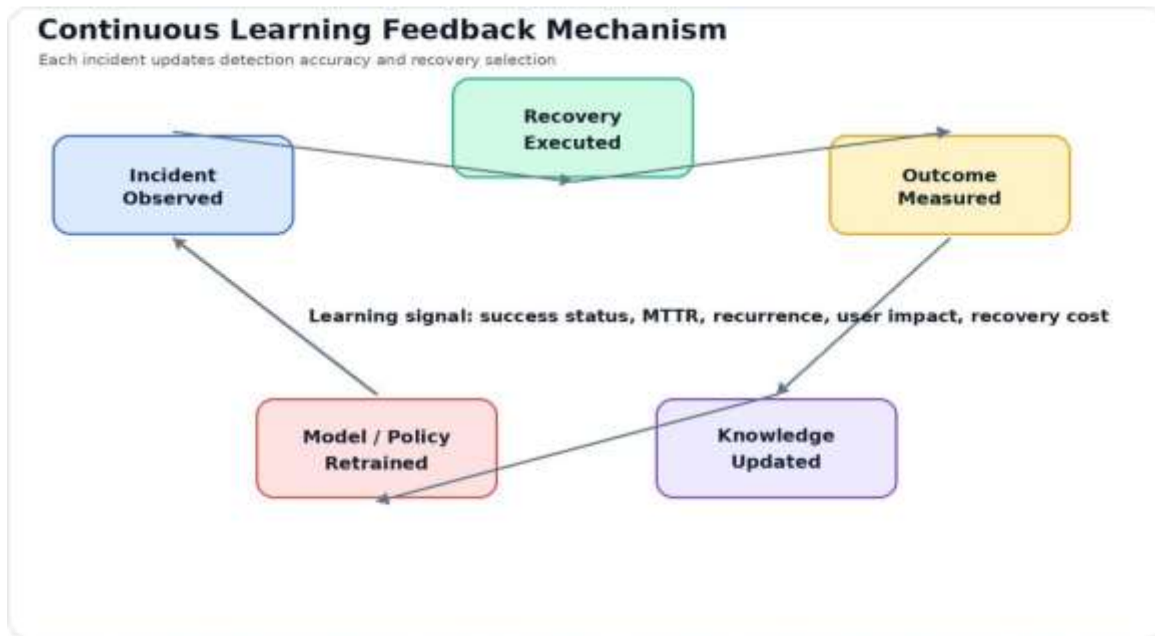


Fig. 10. Continuous feedback learning mechanism for improving future detection and recovery decisions.

Table 6 compares the proposed framework with common recovery approaches.

Method	Strength	Limitation	Adaptability
Traditional Exception Handling	Simple and low overhead	Limited to known code-level errors	Low
Static Monitoring	Improves visibility	Requires manual recovery	Low
Rule-Based Auto-Recovery	Automates common actions	Fails under novel conditions	Medium
Cloud Auto-Healing	Useful for container and VM failures	Weak application-level reasoning	Medium
Predictive Maintenance	Anticipates failures	May not execute recovery	Medium-High
Proposed AURA Framework	Detection, diagnosis, recovery, and learning in one loop	Requires quality telemetry and governance	High

15. Advantages of Proposed System

The proposed system provides proactive fault management by detecting anomalies before they become critical failures. It reduces downtime by automating recovery actions and minimizing dependence on manual debugging. It improves scalability because telemetry and recovery components are modular and can be deployed independently. It improves adaptability through feedback learning, allowing the system to refine its decisions after each incident.

Another advantage is cross-layer visibility. Since the framework considers frontend events, backend APIs, service dependencies, infrastructure metrics, and historical incident patterns, it can distinguish between symptoms and root causes more effectively than isolated monitoring tools. The governance layer also makes the system safer by preventing uncontrolled automation.

16. Limitations

The proposed framework has some limitations. First, model performance depends heavily on telemetry quality. Missing, noisy, or biased data may reduce detection accuracy and produce incorrect recovery decisions. Second, simulated datasets cannot capture every complexity of production environments. Real-world systems may include hidden dependencies, third-party service failures, security restrictions, and unpredictable user behavior.

Third, reinforcement learning requires careful reward design. If the reward function is poorly designed, the recovery agent may learn actions that improve short-term metrics while creating long- term instability. Fourth, autonomous recovery introduces security and governance risks. A compromised recovery controller could perform harmful actions if access control is weak. Fifth, complete root-cause accuracy is difficult when multiple faults occur simultaneously.

17. Future Scope

Future research can extend this framework toward production-scale validation using real telemetry from enterprise systems. Additional work can explore explainable AI for root-cause diagnosis so that human engineers can understand why a particular recovery action was selected. Neuro-symbolic reasoning may also be used to combine machine learning with formal reliability rules.

Another future direction is multi-agent recovery. Different agents may specialize in frontend healing, database recovery, container orchestration, network optimization, and security response. These agents can coordinate through a shared policy model. Future work may also integrate large language models for log summarization, incident explanation, and operator assistance, provided that strict governance and validation mechanisms are applied.

18. Conclusion

This paper presented an intelligent self-healing AI framework for autonomous error detection, root- cause diagnosis, and adaptive recovery in distributed software systems. The proposed AURA architecture integrates telemetry collection, hybrid anomaly detection, dependency-aware diagnosis, reinforcement-learning-based recovery, and feedback-driven improvement. Experimental evaluation in a simulated distributed environment showed that the framework reduced mean time to recovery, improved anomaly detection accuracy, and increased recovery success rate compared with rule-based and static monitoring approaches.

The study demonstrates that self-healing AI systems can move software reliability from reactive maintenance toward autonomous resilience. Instead of waiting for human intervention after failure, future systems can observe their own behavior, diagnose faults, execute controlled recovery actions, and learn from operational outcomes.

While governance, security, and data quality remain important challenges, AI-driven self-healing provides a strong foundation for the next generation of dependable cloud-native and enterprise software systems.

19. References in IEEE Format

[1] N. K. Kuntamukkala, "Self-Healing Angular Architecture: AI-Driven Autonomous Error Recovery and System Resilience," International Journal of Artificial Intelligence, Data Science, and Machine Learning, vol. 5, no. 3, pp. 219-230, 2024.

[2] S. K. Jangam, "Role of AI and ML in Enhancing Self-Healing Capabilities, Including Predictive Analysis and Automated Recovery," International Journal of Artificial Intelligence, Data Science, and Machine Learning, vol. 3, no. 4, pp. 47-56, 2022.

[3] R. K. Vankayalapati, C. Pandugula, V. K. A. T. Ganti, and G. Mishra, "AI-Powered Self-Healing Cloud Infrastructures: A Paradigm for Autonomous Fault Recovery," Migration Letters, vol. 19, no. 6, pp. 1173- 1187, 2022.

[4] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, no. 1, pp. 41-50, 2003.

[5] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," ACM Transactions on Autonomous and Adaptive Systems, vol. 4, no. 2, pp. 1-42, 2009.

[6] H. Psaier and S. Dustdar, "A survey on self-healing systems: Approaches and systems," Computing, vol. 91, no. 1, pp. 43-73, 2011.

[7] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in Proc. USENIX OSDI, 2004, pp. 231-244.

[8] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in Proc. ACM/IEEE Supercomputing Conference, 2006, pp. 88-99.

[9] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.

[10] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," Neural Computation, vol. 12, no. 10, pp. 2451-2471, 2000.

[11] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in Proc. 22nd ACM SIGKDD, 2016, pp. 785-794.

[12] F. T. Liu, K. M. Ting, and Z. H. Zhou, "Isolation Forest," in Proc. IEEE International Conference on Data Mining, 2008, pp. 413-422.

[13] M. Satyanarayanan, "The emergence of edge computing," Computer, vol. 50, no. 1, pp. 30-39, 2017.

[14] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA, USA: OReilly Media, 2016.

[15] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in Proc. ICAC, 2013, pp. 23-27.

[16] A. Hussain, S. Samant, A. Singh, R. Chhabra, V. Arora and A. Kapoor,"AspectBased Sentiment Analysis (ABSA): A Review of Techniques and Applications," 2025 1st IEEE Uttar Pradesh Section Women in Engineering International Conference on Electrical Electronics and Computer Engineering (UPWIECON), Dehradun, India, 2025, pp. 148-153, doi: 10.1109/UPWIECON67212.2025.11390535.