



Journal of Interdisciplinary Science & Technology

A Self-Healing Distributed Cloud Control Plane for Autonomous Fault Detection and Recovery

Naresh C. Thoutam¹, A. Ankita Karale¹, Divyanshu Sachindra Padvi¹, Aditya Sudhakar Naikwade¹

Priyanka Rajendra Mahire¹, Vivek Ravindra Patil¹

¹Sandip Institute of Technology and Research Centre, Nashik, Maharashtra, India

naresh.thoutam@sitrc.org, ankita.karale@sitrc.org, divyanshu.padvi@gmail.com, naikwadeaditya7777@gmail.com,

priyankamahire1834@gmail.com, patilvivek8550@gmail.com

Corresponding author- Aditya Sudhakar Naikwade

DOI: [10.5281/zenodo.23299218](https://doi.org/10.5281/zenodo.23299218)

Abstract: Cloud-native services increasingly run on distributed container platforms, where even short-lived failures can disrupt user-facing functionality and backend workflows. In Kubernetes-based microservice deployments, issues such as pod crashes, performance degradation, network instability, and sudden workload surges must be detected and mitigated automatically to maintain service continuity. This paper introduces a Self-Healing Distributed Cloud Platform (SHDCP) built around a layered control plane that supervises Fast API-powered microservices and orchestrates recovery actions at the pod level. The platform employs an AI-driven anomaly detection pipeline in which Isolation Forest and complementary time-series models analyse metrics and request traces to identify abnormal behavior before it becomes critical. Detected incidents are translated into recovery policies that trigger automatic pod restart, rescheduling, or horizontal scaling through Kubernetes, while a persistent knowledge base records decisions and outcomes for future optimization. In addition to describing the architecture and its core components, the work explains the AI integration for online anomaly detection, discusses practical design challenges, and presents an evaluation plan based on controlled failure injection scenarios including overload, node loss, and network partition. The proposed framework aims to deliver high availability, autonomous fault handling, and operational resilience for modern distributed cloud applications.

Keywords: Self-healing systems, distributed cloud, Kubernetes, FastAPI, Isolation Forest, AI-based anomaly detection, automatic pod recovery, microservices, fault tolerance, autonomous control plane.

1 INTRODUCTION

Distributed cloud systems have become an essential foundation for modern digital services, supporting applications such as e-commerce platforms, financial systems, healthcare services, social networks, Internet of Things applications, and enterprise information systems. Unlike traditional centralized infrastructures, distributed cloud environments deploy applications, services, data, and computing resources across multiple nodes, regions, availability zones, or data centers. This architecture provides several important benefits, including elastic resource scaling, geographic distribution, workload balancing, service isolation, and improved accessibility. Organizations can use these capabilities to deliver services to users with greater flexibility and efficiency. However, the same distribution that improves scalability and availability also introduces considerable operational complexity.

A distributed cloud platform may contain hundreds or thousands of interconnected components, including physical servers, virtual machines, containers, microservices, databases, networks, storage systems, APIs, and orchestration tools [15],[24]. These components are often dependent on one another, meaning that the failure of a single element can affect several related services. Failures may occur at multiple levels, such as infrastructure failure caused by node crashes or resource exhaustion, network failure caused by communication loss or partitioning, application failure caused by software defects or service degradation, and orchestration failure caused by incorrect scheduling or configuration. In addition, sudden workload increases can overload services and reduce their response capacity. As the number of managed components increases, identifying the source of a failure and restoring normal operation becomes more difficult for human operators.

Traditional cloud management approaches generally depend on monitoring tools that generate alerts when abnormal behavior is detected. Although alert-based monitoring is useful, it often requires an administrator or site reliability engineer to analyze the problem and manually select an appropriate recovery action[6],[24]. This process can introduce significant delays, particularly when failures occur outside normal working hours or when multiple incidents occur simultaneously. Manual recovery is also vulnerable to human error, inconsistent decision-making, and incomplete knowledge of the relationships between services. In a highly distributed environment, even a short delay can allow a localized fault to propagate through dependent components, resulting in increased downtime, reduced service quality, and violation of service-level objectives.

Self-healing systems provide an approach for addressing these limitations by enabling software platforms to detect abnormal conditions, diagnose probable causes, select suitable corrective actions, and verify whether recovery has been successful. The concept is closely related to autonomic computing, in which systems are designed to monitor and manage themselves with limited human intervention [7]. A self-healing system may perform actions such as restarting a failed service, replacing an unhealthy container, scaling resources, rerouting requests, isolating a faulty component, restoring a previous software version, or migrating workloads to another node. These actions allow the platform to respond more rapidly and consistently than a completely manual maintenance process.

The importance of self-healing is particularly evident in cloud-native applications based on microservices. Microservice architectures improve modularity and deployment flexibility by dividing an application into smaller, independently managed services [24],[7]. However, they also create complex dependency relationships and increase the number of possible failure points. A problem in one service may cause delayed responses, cascading errors, or unavailability in other services. Therefore, effective self-healing must do more than simply restart failed components. It must consider service health, workload conditions, dependency relationships, historical incidents, and the possible consequences of each recovery action. Excessive or inappropriate recovery actions may themselves create instability, such as repeated restarts, unnecessary scaling, or disruption of healthy components.

This paper presents a Self-Healing Distributed Cloud Platform, referred to as SHDCP, for autonomous fault detection and recovery in distributed cloud environments. The proposed platform is organized around a layered control plane that integrates continuous monitoring, failure detection,

decision-making, recovery execution, notification, and logging [9][6]. The monitoring module collects operational information such as response time, error rate, CPU utilization, memory pressure, and service availability. The failure detection engine evaluates these observations using predefined thresholds and heuristic rules to identify service degradation or abnormal behaviour. Once a failure is detected, the decision engine selects a suitable recovery procedure from a knowledge-driven catalog.

The recovery module executes the selected action, which may include service restart, resource scaling, rollback, workload isolation, or traffic rerouting. After the action is completed, the platform verifies the condition of the affected service through a feedback loop. If the service has not returned to an acceptable state, SHDCP can escalate the incident and attempt an alternative recovery action. At the same time, the logging and knowledge-base components preserve information about the incident, selected action, recovery outcome, and system response [15]. This historical information can support future diagnosis, policy refinement, and more informed recovery decisions.

The primary objective of SHDCP is to reduce mean time to recovery, improve fault containment, limit manual intervention, and preserve service availability under changing workload conditions [15],[24]. Unlike traditional reactive maintenance, the proposed closed-loop design treats recovery as a continuous process of detection, decision, action, and verification. By combining explainable rule-based control with persistent operational knowledge, SHDCP provides a practical foundation for developing resilient and autonomous distributed cloud systems. The remainder of this paper discusses the problem statement, conceptual background, related research, proposed architecture, implementation methodology, evaluation plan, limitations, and future enhancements of the platform.

II. LITERATURE SURVEY

Self-healing distributed cloud systems have developed from basic service-restart mechanisms into integrated platforms that combine monitoring, anomaly detection, failure diagnosis, recovery planning, and post-recovery verification. The primary objective of these systems is to reduce downtime, improve service availability, and minimize dependence on manual intervention. Recent research has focused on reinforcement learning, predictive analytics, Kubernetes-based recovery, AIOps, and intelligent anomaly detection [1], [2].

Nikolic et al. investigated the relationship between fault correction and fault tolerance in distributed systems. Their study explains that automatic fault correction is not always beneficial because an aggressive recovery action may create

additional disruption. For example, restarting a service that is experiencing only temporary degradation may interrupt active requests and increase the overall recovery time. This observation is important for self-healing platforms because the recovery controller must determine whether a fault should be corrected immediately or temporarily tolerated.

Kumar et al. proposed a self-healing cloud ecosystem based on reinforcement learning of failure patterns. Their approach enables the system to learn from previous incidents and improve the selection of future recovery actions. Reinforcement learning is useful because it can identify relationships between failure conditions, recovery actions, and recovery outcomes. However, the method requires an appropriate reward function and sufficient historical data. Unsafe exploration during learning may also cause undesirable actions in a production environment.

Vyawahare et al. examined self-healing mechanisms for distributed cloud systems and emphasized the importance of continuous monitoring, fault detection, and automated recovery. Their work supports the use of a distributed control mechanism that can supervise multiple services and infrastructure components. However, a general architecture must also address service dependencies, recovery verification, logging, and the possibility of repeated recovery failure.

Tutuncuoglu introduced a proactive self-healing architecture based on predictive artificial-intelligence agents [3]. Predictive recovery attempts to identify a failure before it becomes critical by analysing resource utilization, historical incidents, and system behavior. Such an approach can reduce reaction time compared with purely reactive monitoring. Nevertheless, predictive models may generate false alarms when workload patterns change, and their decisions may be difficult to interpret without suitable logging and explanation mechanisms.

Kubernetes provides built-in self-healing features through health probes, replica management, pod replacement, and workload rescheduling [4]. These features are effective for common failures, such as terminated containers and failed readiness checks. Prometheus can complement Kubernetes by collecting time-series metrics related to CPU usage, memory consumption, request latency, error rate, and service availability [5]. However, native orchestration mechanisms generally operate at the container or workload level. They may not understand application-level dependencies, incident history, or the relative importance of different services.

Cloud-native anomaly-detection architectures have also been proposed for Kubernetes and managed platforms such as Azure Kubernetes Service [8]. These systems generally use service metrics, time-series analysis, and machine-learning

models to detect abnormal behavior and initiate actions such as pod restart, scaling, and traffic redistribution. Their findings indicate that anomaly detection is more effective when multiple metrics are analysed together rather than when each metric is treated independently.

Yazdanparast classified self-healing software approaches into rule-based, model-based, and learning-based methods [9]. Rule-based approaches are comparatively easy to implement and audit because their decisions can be directly traced to predefined conditions. Model-based approaches use system representations and dependency relationships to diagnose faults. Learning-based approaches can adapt to changing system behavior but generally require larger datasets and more complex validation.

Automatic cloud-management research has emphasized that self-healing should be implemented as a closed-loop process [10]. A complete loop should contain monitoring, analysis, planning, execution, and verification. Without verification, the system may assume that an action such as restarting a pod has solved the problem even when the underlying fault continues. The AWS Well-Architected Framework similarly emphasizes reliability, monitoring, recovery planning, and controlled failure response in cloud systems [11].

Shah and Patel explored artificial intelligence for autonomous software recovery [16]. Their work supports the use of intelligent decision-making for restart, rollback, scaling, and other remediation operations. Moreshchini et al. discussed the cloud-continuum concept, in which applications may span cloud, edge, and intermediate computing resources [17]. This environment increases the need for distributed monitoring and location-aware recovery decisions.

The literature indicates that existing research generally emphasizes one particular capability, such as intelligent anomaly detection, Kubernetes recovery, predictive analysis, or reinforcement learning [1], [3]. SHDCP combines these concepts into a single layered control plane. The proposed design uses deterministic rules for transparent initial decisions, a persistent knowledge base for recording previous incidents, and a feedback mechanism for verifying recovery outcomes.

The main research gap is the absence of a lightweight and auditable framework that integrates detection, decision-making, recovery, logging, and verification for distributed microservices. Existing orchestration systems can restart failed workloads, but they may not select recovery actions according to incident history or service-level priorities [4], [9]. Similarly, AI-based methods can detect complex patterns, but they may introduce model overhead and explainability problems [6], [7], [12]. SHDCP addresses this gap through a hybrid design that can operate with rule-based logic initially

and support machine-learning enhancements in future versions.

Reference	Main approach	Detection method	Recovery method	Strength
Kumar et al.	Reinforcement-learning cloud ecosystem	Learned failure patterns	Policy-based recovery	Learns from previous incidents
Vyawahare	Distributed cloud self-healing	Service and infrastructure monitoring	Automated fault handling	Focuses on distributed environments
Kubernetes	Container orchestration	Health and readiness probes	Pod restart and rescheduling	Practical and widely available
Ji and Luo	LLM-assisted recovery	Logs, traces, and telemetry	Intelligent fault analysis	Handles unstructured operational data
SHDCP, proposed	Hybrid closed-loop control plane	Rules, heuristics, and optional AI	Restart, scale, rollback, isolate, escalate	Combines auditability and adaptability

Table no 1: Comparative Study of Existing Self-Healing Cloud Systems

III. PROBLEM STATEMENT

Modern distributed cloud systems experience recurring failures caused by node crashes, service overload, configuration errors, resource exhaustion, and network instability [7], [15]. Existing reactive approaches typically depend on alerts and operator response, which introduces delay and increases the chance of service interruption. In highly distributed environments, these delays can propagate failures across dependent services and reduce system reliability.

The core problem is the lack of a lightweight, auditable, and adaptive recovery mechanism that can monitor services continuously, detect degradation early, choose an appropriate remediation action, and restore service without manual intervention. The challenge becomes more difficult when the system must manage multiple microservices while preserving low overhead and maintaining historical knowledge of past incidents. SHDCP is proposed to address this gap through a layered closed-loop control plane.

IV. BACKGROUND AND DEFINITION

A distributed cloud system refers to a cloud environment where workloads, services, and resources are deployed across multiple nodes or sites, often with geographic distribution and heterogeneous infrastructure [15]. Such systems may include microservices, containers, virtual machines, and cross-zone workloads.

A self-healing system is one that can recognize incorrect behavior and initiate repair actions automatically or semi-automatically [6], [24]. The essential capabilities are self-diagnosis, self-repair, and self-adaptation. In practice, these capabilities may involve restarting services, moving workloads, scaling resources, rerouting requests, or rolling back unstable deployments [7], [13].

Autonomic computing provides the conceptual basis for self-healing because it emphasizes systems that can self-configure, self-optimize, self-protect, and self-heal [24]. In distributed cloud platforms, self-healing contributes directly to resilience engineering by reducing the impact of faults and improving continuity of service [15].

V. RELATED WORK

Early research on self-healing systems established the idea that systems should monitor themselves and recover automatically from faults [24]. Ghosh et al. provided one of the foundational surveys, describing self-healing as a combination of fault detection, recovery, and adaptive feedback mechanisms [24]. This work remains relevant because it introduces the conceptual structure used by later studies.

More recent surveys have focused on software-centric self-healing methods, including rule-based, model-based, and learning-based approaches [6]. These studies show that modern self-healing systems increasingly rely on runtime monitoring, anomaly detection, and recovery policies that respond to changing system states [6]. In cloud environments, this trend becomes more important because service dependencies and workload patterns are more dynamic.

Cloud-focused studies emphasize that self-healing must integrate redundancy, health checks, automated recovery, and dynamic configuration to remain effective at scale [7]. Other works describe AI-assisted healing mechanisms that use prediction and reinforcement learning to choose recovery actions more intelligently [3], [13]. Research on distributed systems also highlights an important dilemma: deciding whether to correct a fault aggressively or tolerate it temporarily to avoid unnecessary disruption [15]. These insights motivate the SHDCP design, which combines deterministic rules with operational traceability and historical learning.

VI. PROPOSED WORK

1. User and Client Layer

This is the entry layer through which users and external applications interact with the cloud platform. Requests enter the system through standardized APIs, making integration easier for web services, mobile applications, and third-party clients. This layer is intentionally kept simple so that user-facing communication remains stable even when internal recovery actions are taking place.

2. Cloud Services Layer

The cloud layer hosts multiple managed microservices, such as Service A, Service B, and Service C. Each microservice operates independently but is supervised by the SHDCP control plane. This modular structure improves isolation because a failure in one service does not automatically disrupt the others. It also supports scalable deployment because additional services can be added without redesigning the entire platform.

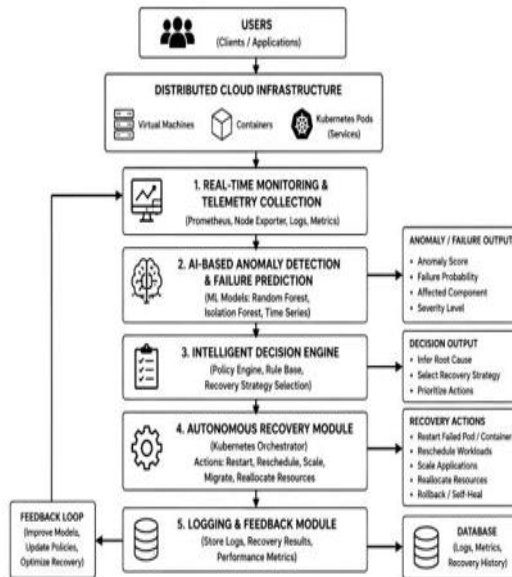


Figure 1: System Architecture

3. SHDCP Control Plane

The control plane is the intelligence core of the system. It contains five functional modules that operate as a closed-loop recovery engine:

Monitoring Module: Performs continuous health polling and collects runtime metrics.

Failure Detection Engine: Applies threshold-based and heuristic rules to identify anomalies or degradation.

Decision Engine: Selects a recovery action from a predefined knowledge catalog.

Recovery Module: Executes automated recovery playbooks such as restart, scaling, or rollback.

Logging and Notification System: Records all decisions in an audit trail and sends alerts through email or webhook channels.

This layered design makes the system explainable and operationally manageable. It is also suitable for practical deployment because the decision logic is explicit and easy to audit.

4. Knowledge Base

The knowledge base is implemented as a persistent repository, such as MongoDB, to store logs, prior recovery actions, and performance metrics. This historical store helps the control plane make informed decisions based on previous incidents. Over time, the repository can also support pattern analysis and refinement of recovery policies.

5. Feedback Loop

A restoration feedback loop connects the recovery module back to the cloud services layer. After recovery, the system verifies whether service health has returned to an acceptable level. If the issue persists, the control plane can escalate to an alternative action. This closed-loop approach is the key to continuous resilience because it ensures that healing is not a one-time event but an ongoing process [7], [15].

VII. METHODOLOGY

The SHDCP methodology follows a detect-decide-act-verify cycle. The monitoring module collects service health data at regular intervals, including response time, error rate, CPU usage, memory pressure, and service availability. These metrics are passed to the detection engine, which evaluates them against predefined thresholds and heuristic indicators.

When abnormal behavior is detected, the decision engine maps the incident to a recovery policy. For example, a temporary overload may trigger scaling, while a stuck or failed service may trigger a restart. If the failure is more severe, rollback or isolation may be selected. The recovery module then executes the corresponding playbook and updates the knowledge base with the outcome.

The methodology is intentionally practical rather than overly complex. Instead of depending entirely on opaque machine learning models, it uses rule-supported control logic that can be validated in real deployments. This makes the system more suitable for student projects, prototype deployment, and incremental future enhancement.

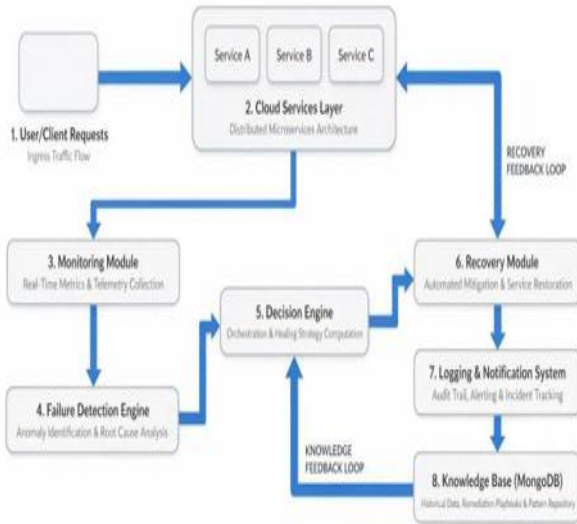


Figure 2. Closed-loop self-healing workflow for distributed cloud systems in SHDCP methodology.

1. Methodological Overview

The proposed Self-Healing Distributed Cloud Platform follows a closed-loop detect– decide– act– verify process. The platform continuously observes services, identifies abnormal behavior, selects an appropriate recovery policy, executes the policy, and verifies whether the service has returned to a stable condition. This design follows established principles of automated cloud management and self-healing software systems [9], [10].

The complete workflow is expressed as:

$$O_t \rightarrow D_t \rightarrow P_t \rightarrow A_t \rightarrow V_t$$

where O_t denotes system observations at time t , D_t denotes the detection result, P_t denotes the selected recovery policy, A_t denotes the recovery action, and V_t denotes the verification result.

2. Monitoring and Data Collection

The monitoring module collects operational data from service endpoints, containers, nodes, and orchestration APIs. The main metrics include:

- CPU utilization.
- Memory utilization.
- Request latency.
- Error rate.
- Throughput.
- Pod restart count.
- Service availability.

Network communication failures.

Queue length.

Number of active replicas.

The monitored data at time (t) can be represented as:

$$\mathbf{x}_t = [x_t^{cpu}, x_t^{mem}, x_t^{lat}, x_t^{err}, x_t^{req}, x_t^{rep}, x_t^{net}]$$

where each variable represents a normalized operational metric. Normalization is useful because different metrics have different units and value ranges.

Prometheus can store the time-series data, while the SHDCP control plane can periodically retrieve the latest values for analysis. Historical data can also be stored in a knowledge base such as MongoDB, allowing the system to compare current incidents with previous events [5], [12].

3. Failure Detection

The failure-detection engine uses threshold rules, persistence conditions, and optional anomaly models. A basic threshold rule is represented as:

$$F_t = \begin{cases} 1, & \text{if } x_t^{cpu} > \tau_{cpu} \vee x_t^{mem} > \tau_{mem} \vee x_t^{lat} > \tau_{lat} \vee x_t^{err} > \tau_{err} \\ 0, & \text{otherwise} \end{cases}$$

where ($F_t=1$) indicates abnormal behaviour and (τ) represents the acceptable threshold for a metric. Threshold-based detection is simple and explainable, which is useful for safety-critical recovery decisions [9].

To prevent a single temporary spike from triggering recovery, SHDCP applies a persistence rule:

$$F_t^* = 1 \quad \text{if} \quad \sum_{i=t-k+1}^t F_i \geq m$$

where (k) is the monitoring window and (m) is the minimum number of abnormal observations required to confirm the incident.

This hybrid arrangement permits rule-based detection for known faults and model-based detection for unfamiliar patterns [9], [26].

4. Decision-Making and Recovery Selection

After a fault is detected, the decision engine determines the most suitable recovery action. The decision process considers fault type, severity, service dependency, previous recovery results, and possible disruption. The use of historical recovery outcomes is consistent with research on learning-based self-healing and AIOps-based operational automation [1], [12].

Let the possible recovery actions be:

$$A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$$

where the actions may represent restart, rescheduling, scaling, rollback, isolation, and traffic rerouting. For example, a

temporary increase in CPU utilization may trigger horizontal scaling, while a failed health check may trigger a pod restart. A repeated restart failure may cause the system to select rescheduling or rollback. Such recovery operations are supported by Kubernetes-based orchestration mechanisms [4], [19].

5. Recovery Execution

The recovery executor communicates with the orchestration layer and executes the selected playbook. The recovery actions include:

Restarting an unhealthy pod.

Replacing a failed container.

Rescheduling a workload to another node.

Increasing the number of replicas.

Applying horizontal scaling.

Rolling back an unstable deployment.

Rerouting traffic away from an unhealthy instance.

Isolating a failed service.

Escalating the incident to a human operator.

These operations are aligned with established container orchestration and site-reliability practices [4], [19].

All recovery actions should be logged with a timestamp, service identifier, fault type, action type, and execution result. The system should also apply a retry limit to avoid repeated restart loops. Controlled recovery is necessary because an unsuitable automated action can worsen service instability [18], [20].

6. Knowledge Base and Feedback Loop

The knowledge base stores the history of incidents and recovery outcomes. An incident record can be represented as:

$$I_i = (T_i, F_i, S_i, A_i, \Delta t_i, O_i)$$

where T_i is the incident time, F_i is the fault type, S_i is the severity, A_i is the recovery action, Δt_i is the recovery duration, and O_i is the final outcome.

After the recovery action is executed, the verification module checks service health using readiness probes, response latency, error rate, and availability. Kubernetes health mechanisms provide a practical basis for this verification [4], while closed-loop cloud-management approaches emphasize that recovery should be confirmed through feedback rather than assumed after command execution [10].

The action is considered successful if:

$$V_i = \begin{cases} 1, & \text{if } x^{lat} \leq \tau_{lat} \wedge x^{err} \leq \tau_{err} \wedge Availability \geq \tau_A \\ 0, & \text{otherwise} \end{cases}$$

If ($V_i=0$), the platform may select another action or escalate the incident. This feedback process ensures that recovery is measured by restored service health rather than merely by completion of a command [10], [11].

7. Evaluation Metrics

The proposed evaluation uses Mean Time to Detection (MTTD), Mean Time to Recovery (MTTR), availability, automatic recovery success, manual intervention rate, precision, recall, and F1-score. These metrics allow the proposed platform to be evaluated in terms of detection quality, recovery speed, service availability, and operational intervention [12], [20].

8. Expected Contribution

The proposed SHDCP methodology contributes a structured and measurable framework for self-healing distributed cloud systems. Its main contribution is the integration of monitoring, fault detection, recovery selection, automated execution, knowledge storage, and post-recovery verification into one closed-loop control plane. The use of explicit rules improves transparency, while the knowledge base supports future learning from previous incidents.

VIII. IMPLEMENTATION PLAN AND EVALUATION

The implementation can be prototyped using containerized microservices deployed on a Kubernetes or Docker-based environment. The monitoring module can collect health data from service endpoints, while the control plane processes that information and applies recovery logic. MongoDB can be used to store logs, incident records, and action outcomes. Notification services can be integrated using SMTP email or webhook-based alerting.

A. Failure Injection Scenarios

The evaluation should include at least the following fault cases:

- Service crash: terminate one service instance unexpectedly
- Service overload: generate traffic beyond normal operating limits.
- Node failure: simulate loss of a compute node.
- Network partition: block communication between services.
- Slow degradation: reduce response capacity gradually to test early detection.

2. Metrics

- The evaluation should measure:
- Mean Time to Recovery.
- Service downtime.

- Number of successful auto-recoveries.
- Number of manual interventions.
- False positives and false negatives in detection.
- Resource overhead caused by monitoring and recovery.
- SLA compliance before and after healing.

3. Baseline

The baseline should be a traditional reactive recovery model where failures are handled manually or by simple restart scripts. SHDCP should be compared against this baseline to show whether the closed-loop design improves response time and reduces downtime. Even a small experimental setup can demonstrate whether automated healing reduces operational burden.

4. Expected Evaluation Outcome

The expected outcome is that SHDCP will detect failures earlier than manual methods and reduce the time needed to restore services. The system should also show better consistency in handling repeated fault scenarios because the knowledge base retains incident history. The recovery module should improve availability while keeping overhead at an acceptable level.

IX. DISCUSSION

The proposed architecture offers practical benefits, but it also introduces trade-offs. Continuous monitoring improves visibility, yet it consumes resources and may create overhead if metrics are too frequent. Rule-based decision logic is easy to understand, but it may not adapt well to completely new failure patterns.

Another limitation is that automated recovery must be carefully controlled to avoid repeated restarts or unnecessary scaling. A poorly tuned recovery rule can worsen service instability instead of improving it. For this reason, the logging and knowledge-base layer is important because it enables post-incident analysis and future refinement.

The SHDCP model is most effective when used in environments where service behavior is sufficiently observable and where common fault patterns can be mapped to clear recovery actions. In future versions, the decision engine can be extended with statistical prediction or machine learning for improved adaptability [3], [13].

X. CONCLUSION

Self-healing distributed cloud systems are essential for modern cloud infrastructure because they reduce the dependence on manual intervention and improve resilience under failure. The SHDCP architecture presented in this paper provides a structured closed-loop framework for

monitoring, detecting, deciding, recovering, and learning from incidents. Its layered design supports service isolation, transparent logging, and automated restoration.

By combining a control plane, persistent knowledge base, and feedback loop, the proposed model offers a practical foundation for building resilient cloud-native systems. Future work can extend the framework with predictive analytics, multi-cloud support, and more advanced recovery policies. The result is a more adaptive and dependable cloud platform suited for real-world distributed deployments [15], [24].

XI. ACKNOWLEDGEMENT

The authors thank the Department of Computer Engineering, Sandip Institute of Technology and Research Centre, Nashik, Maharashtra, India, and the guide Vijay Karwande for support and guidance during this study.

XII. REFERENCES

- [1] S. Kumar B, J. Kumari M, A. Krishna Veni, R. Subapriya, R. Ramya, and R. Deepalakshmi, "Self-Healing Cloud Ecosystems via Reinforcement Learning of Failure Patterns," in Proc. 9th Int. Conf. on I-SMAC (IoT in Social, Mobile, Analytics and Cloud), IEEE, 2025, pp. 267–273, doi: 10.1109/ISMAL65734.2025.11597603.
- [2] S. Vyawahare, A. Thete, S. Shirsat, P. Mandlik, and P. R. Dholi, "Self-Healing Distributed Cloud Systems," International Research Journal of Modernization in Engineering Technology and Science (IRJMETS), vol. 7, no. 8, Aug. 2025.
- [3] B. T. Tutuncuoglu, "AuroraShield: Proactive Self-Healing Architecture for Cloud Infrastructure Using Predictive AI Agents," 2025.
- [4] Cloud Native Computing Foundation, "Kubernetes Documentation: Self-Healing Applications," 2025.
- [5] Prometheus Authors, Prometheus Documentation. Cloud Native Computing Foundation, 2025.
- [6] C. Ji and H. Luo, "Cloud-Based AI Systems: Leveraging Large Language Models for Intelligent Fault Detection and Autonomous Self-Healing," arXiv, 2025.
- [7] V. Nunavath, "AI-Enhanced Self-Healing Kubernetes for Scalable Cloud Operations," World Journal of Advanced Engineering Technology and Sciences, vol. 16, no. 2, pp. 21–29, 2025.
- [8] Scalable Cloud-Native Architecture for Automated Anomaly Detection and Intelligent Response in Kubernetes and AKS Platforms," International Journal of Engineering Research and Science & Technology, vol. 21, no. 2, pp. 3088–3095, 2025.

- [9] Z. Yazdanparast, "A survey on self-healing software system," arXiv, 2024.
- [10] "Cloud Management with Automatic Self-Healing Services: A Comprehensive Study," 2024.
- [11] Amazon Web Services, AWS Well-Architected Framework, 2024.
- [12] R. Krishnan, "Artificial Intelligence for IT Operations (AIOps): A survey," IEEE Cloud Computing, vol. 10, no. 2, pp. 45–56, 2023.
- [13] L. Kong et al., "Edge computing-driven Internet of Things: A survey," ACM Computing Surveys, vol. 55, no. 8, pp. 1–41, 2023.
- [14] A. Ullah et al., "Orchestration in the cloud-to-things compute continuum: Taxonomy, survey and future directions," Journal of Cloud Computing, vol. 12, no. 1, pp. 135–164, 2023.
- [15] Y. Chiang et al., "Management and orchestration of edge computing for IoT: A comprehensive survey," IEEE Internet of Things Journal, vol. 10, no. 16, pp. 14307–14331, 2023.
- [16] H. Shah and J. Patel, "Self-Healing AI: Leveraging Cloud Computing for Autonomous Software Recovery," International Journal of Intelligent Systems and Applications in Engineering, vol. 10, no. 3s, 2022.
- [17] S. Moreschini et al., "Cloud continuum: The definition," IEEE Access, vol. 10, pp. 131876–131886, 2022.
- [18] J. Nikolic, N. Jubatyrov, and E. Pournaras, "Self-healing dilemmas in distributed systems: Fault correction vs. fault tolerance," 2020.
- [19] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," ACM Queue, vol. 14, no. 1, pp. 70–93, 2016.
- [20] B. Beyer, C. Jones, J. Petoff, and N. Murphy, Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media, 2016.
- [21] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in Proc. ACM SIGKDD, 2016, pp. 785–794.
- [22] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016.
- [23] A. Verma et al., "Large-scale cluster management at Google with Borg," in Proc. EuroSys, 2015, pp. 1–17.
- [24] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," IEEE Cloud Computing, vol. 1, no. 3, pp. 81–84, 2014.
- [25] P. Mell and T. Grance, The NIST Definition of Cloud Computing, NIST Special Publication 800-145, 2011.
- [26] F. T. Liu and K. M. Ting, "Isolation Forest," in Proc. IEEE Int. Conf. Data Mining (ICDM), 2008, pp. 413–422.
- [27] D. Ghosh, R. Sharman, H. R. Rao, and S. Upadhyaya, "Self-healing systems—survey and synthesis," Decision Support Systems, 2006.