

AI Powered Predictive Self-Healing Software Systems: A Review

Mansi Suryakant Kyatankeri, Prof. Abhishek Popat Bangar, Prof. S. B. Bhosale, Dr. A. A. Khatri

Department of Computer Engineering
Jaihind College of Engineering, SPPU, Pune, India

Abstract- Modern software, networks, operating systems and factories have grown too complex to be repaired by hand. Self-healing systems try to notice a problem, find its cause and fix it without waiting for a person. This paper reviews four recent studies that attack this goal at four different levels: finding faulty lines of code with a bee-colony and chaotic genetic search, choosing repair actions for software-defined networks with reinforcement learning, repairing Android, Linux, macOS and Windows machines from system logs, and a large survey of fault diagnosis and self-healing in smart manufacturing. We explain each study in plain language, compare their methods, data and results side by side, and point out where their numbers or claims need careful reading. All four follow the same basic loop of monitor, detect, diagnose and repair, and all report clear gains over simple baselines. However, most results come from simulations, emulators or benchmarks rather than live systems, multiple simultaneous faults and slowly growing faults are handled weakly, and almost nothing acts before a failure happens. We close with research gaps and directions for predictive, safe and explainable self-healing software.

Keywords— Self-healing software, fault localization, fault recovery, reinforcement learning, anomaly detection, predictive maintenance, smart manufacturing

I. INTRODUCTION

Software is now part of almost everything we depend on: banking, hospitals, power grids, phones and factory machines. When a program crashes, a network link drops or a sensor on a production line drifts, the cost is lost time, lost money and sometimes safety risk. Waiting for an engineer to read logs and apply a fix is slow, and the number of components keeps growing.

A self-healing system is one that can notice that something is wrong and bring itself back to normal operation without outside help [8].

In practice this means a repeating loop: watch the system, detect a problem, work out where the problem is, apply a repair and check that it worked.

A predictive self-healing system goes one step further and tries to act before the failure actually happens.

Many papers study parts of this loop, but they are scattered across different fields. This review brings together four recent studies, each looking at a different layer of the loop: program code [1],

computer networks [2], operating systems [3] and industrial plants [4]. Our aim is to explain them in simple words, compare them fairly, and show what is still missing. The main contributions of this review are:

- A plain-language summary of the problem, method, data and results of each study.
- A side-by-side comparison of techniques, test settings and reported numbers.
- A critical look at weaknesses and inconsistencies that readers should know about.
- A list of research gaps, with a focus on predictive self-healing software.

The rest of the paper is organized as follows. Section II gives background. Section III reviews the four studies. Section IV compares them. Section V gives critical remarks, Section VI lists gaps and future directions, and Section VII concludes.

II. BACKGROUND

1. The Self-Healing Loop

Figure 1 shows the loop used throughout this review. The first step is monitoring: collecting logs, counters, sensor readings or test results. Next comes detection, which decides that something is abnormal. Diagnosis then finds where the fault is and what kind it is. Smart manufacturing work splits this into detection (is there a fault and when did it start), isolation (where is it) and identification (how big is it) [4]. Repair applies an action such as restarting a service, resetting a port or switching to a backup. Finally, the system verifies the result and, in learning-based systems, improves its future choices.

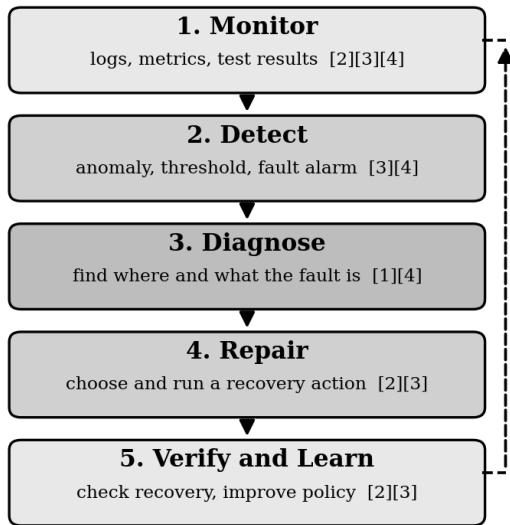


Fig. 1. The self-healing loop and the studies [1]–[4] that address each step.

2. Kinds of Faults

Faults can be grouped in several ways. By time behaviour they are abrupt (sudden), intermittent (come and go) or incipient (grow slowly as equipment wears out). By location they affect a sensor, an actuator or the process itself. By how they enter the equations they are additive (an extra unknown signal) or multiplicative (a change in a system parameter) [4]. Faults can also be hardware, network or software faults. A system with multiple faults at once is much harder to diagnose, because their symptoms overlap [1], [4].

3. Common Techniques

Early self-healing relied on fixed rules and redundancy, such as duplicate hardware. Newer work uses models of the physical system, machine learning on collected data, search algorithms inspired by nature (metaheuristics), and reinforcement learning (RL), where a program learns by trial and reward. Metaheuristics try many candidate answers and keep improving the best ones. RL agents such as Proximal Policy Optimization (PPO) [5] learn which action to take in each situation.

4. Measures Used in the Studies

Fault localization papers use the **EXAM score**, the share of the program a developer must read before reaching the first faulty statement [1]. Lower is better:

$$EXAM = S_f / S_t \times 100 \quad (1)$$

where S_f is the number of statements checked before the fault is found and S_t is the total number of statements. **Top-N accuracy** asks whether the true fault appears in the first N suggestions. Recovery papers use **Mean Time To Recovery (MTTR)**, the time from detecting a fault to finishing the repair [3]:

$$t_r = t_{res} - t_{det} \quad (2)$$

Spectrum-based fault localization (SBFL) is the standard starting point. It counts, for each program statement, how often it runs in failing and passing tests. The popular Ochiai formula [7] is:

$$Susp(s) = a_{ef} / \sqrt{(a_{ef} + a_{nf})(a_{ef} + a_{ep})} \quad (3)$$

Here a_{ef} and a_{ep} are the numbers of failing and passing tests that run statement s , and a_{nf} is the number of failing tests that do not.

III. REVIEW OF SELECTED STUDIES

1. Fault Localization with ABC and Chaotic GA [1]

Problem. When a program has several bugs, SBFL scores get diluted and many statements tie. Developers then waste time reading harmless code. Idea. Ghosh and Singh combine two search methods. Artificial Bee Colony (ABC) sends “scout”,

"employed" and "onlooker" bees to explore widely. A Chaotic Genetic Algorithm (CGA) then refines the best candidates using crossover and mutation steered by a logistic chaotic map, which helps avoid getting stuck on a poor answer. A weighted score, $0.4 \times \text{SBFL} + 0.3 \times \text{ABC} + 0.3 \times \text{CGA}$, gives the final ranking of statements.

Setup. Tests used five Java projects from Defects4J v2.0 [6] (JFreeChart, Closure Compiler, Commons Math, Joda-Time, Mockito), with population 50, 100 iterations and 10 repeated runs, written in Python. Results. The average EXAM score fell from 57.3% (SBFL) to 35.7%, and Top-1, Top-5 and Top-10 accuracy reached 82.9%, 90.8% and 97.2%. Removing either half hurts: ABC-only gave 42.3% and CGA-only 44.1% EXAM. Wilcoxon tests give $p = 0.007$ against SBFL. The cost is time: 15.9 s per run versus 3.2 s for plain Ochiai. Limits. Only Java and one benchmark were used; the tool was not built into a developer workflow; and the method does not explain why a statement is ranked high.

2. Reinforcement Learning for Self-Healing SDN [2]

Problem. Software-defined networks (SDN) put control in a central controller, so link, port, flow and controller faults can disrupt many devices. Earlier RL work, such as FRRL [10], handled mainly link failures, or service chaining. Idea. Makiyah et al. build a test network with Mininet, the Faucet controller and Prometheus monitoring, inject faults and UDP congestion on purpose, and record eight measurements every 10 seconds (controller CPU and memory, OpenFlow counters, number of ports down). A PPO agent with a small neural network picks one of four actions: do nothing, reset all ports on a switch, restart the switch, or run a custom repair. The reward is designed by severity. Each fault type has a penalty and a larger "heal bonus" (for example $-10/+12$ for link faults and $-7/+10$ for port faults), a restored controller earns $+20$, and acting on a healthy network costs -1 . This pushes the agent to act decisively but not needlessly. Three versions were trained, adding reward parts step by step.

Results. Mean recovery time was 2.6 steps (about 26 s) against 4.8 steps (48 s) for a rule-based baseline,

and the proxy for packet loss fell from 18.4% (no healing) to 3.2%. In healthy states the agent chose "do nothing" 91.2% of the time, and in faulty states it chose a healing action 85–95% of the time. After 5,000 extra training steps it also worked on an 8-switch and a 16-switch fat-tree network, with recovery slowing to 3.4 and 4.2 steps. Limits. Everything ran in emulation, training used only about 10,000 steps, the rule-based baseline is a stand-in for earlier published methods, and no real switches were used.

3. CROSS: Cross-Platform Remediation from Logs [3]

Problem. Most log tools only detect anomalies; a person still has to fix them. Idea. Johnphill et al. extend their earlier LISH log-analysis system [9]. Logs from Android, Linux, macOS and Windows are turned into word counts (CountVectorizer) and sorted into normal, warning or error by a Multinomial Naive Bayes classifier. If the number of errors or warnings in a time window passes a threshold, an OS-specific handler runs: ADB commands on Android, package updates and service restarts on Linux, PowerShell scripts on Windows. If the real tool is missing, the system logs the failure and runs a simulated version, so the workflow does not stop. Prometheus and Grafana show counts of errors, fixes, scrape time and resource use. The code runs in Docker and Kubernetes.

Results. Averaged over 10 runs, MTTR was 34.1 s for CROSS, 54.2 s for static rules and 89.4 s for manual scripts. Precision and recall were 0.91 and 0.94, with about 9.2% CPU and 75 MB memory. Android logs held by far the most errors (154,232) and Linux the most warnings (4,198). Limits. The authors state that attack-style anomalies were not benchmarked, that real industrial controllers (PLCs, RTUs) were not tested, and that OS logs are only a rough stand-in for plant behaviour. Several fixes ran only as simulations.

4. Survey of Diagnosis and Self-Healing in Manufacturing [4]

Aldrini, Chihi and Sidhom reviewed about 256 papers from 2010 to 2022. They note that earlier surveys treat fault diagnosis and self-healing separately, so

they propose one conceptual model that links sensors, actuators and the process to fault detection, isolation, identification and then to self-healing and fault-tolerant control (SH-FT). They classify faults by time, place and equation (Section II-B). Diagnosis methods fall into physics-based (white box, for example parity space and state estimation), data-driven (black box, for example neural networks and PCA) and signal processing (for example wavelets). Of 163 diagnosis papers, about 55% were black box, 35% white box and 10% signal processing.

Self-healing work is split into passive fault-tolerant control, where a robust controller is designed once to cope with expected faults, and active control, which detects the fault and reconfigures the controller. Only about 30% of the reviewed papers concern self-healing. The authors conclude that (i) incipient and multiple faults are under-studied, (ii) combining physical and data-driven models is a promising direction, and (iii) new self-healing methods for resilient manufacturing are needed.

Table 1: Side-by-Side Comparison of the Four Studies

Study	Layer	Main technique	Test setting	Headline result
[1] Ghosh and Singh, 2026	Program code	Artificial bee colony plus chaotic genetic algorithm; weighted score	Defects4J, 5 Java projects, 10 runs	EXAM 57.3% to 35.7%; Top-5 90.8%
[2] Makiyah et al., 2026	Network (SDN)	PPO reinforcement learning with severity-based reward	Mininet, Faucet, Prometheus; 3 topology sizes	Recovery 2.6 vs 4.8 steps (rule-based)
[3] Johnphill et al., 2026	Operating systems	Naive Bayes log classifier with threshold-triggered fixes	Logs from 4 OS; Docker, Kubernetes; 10 runs	MTTR 34.1 s vs 54.2 s (rules), 89.4 s (manual)
[4] Aldrini et al., 2024	Factory (smart manufacturing)	Literature review and conceptual model	About 256 papers, 2010–2022	Gaps: incipient and multiple faults; hybrid models

IV. COMPARATIVE ANALYSIS

Table I shows that the four studies sit at different layers but share one structure. Studies [1]–[3] each

build a technique and test it against simpler baselines, while [4] steps back and maps the whole field. Table II collects the main numbers reported by the authors.

Table 2: Main Reported Numbers

Study	Layer	Main technique	Test setting	Headline result
[1] Ghosh and Singh, 2026	Program code	Artificial bee colony plus chaotic genetic algorithm; weighted score	Defects4J, 5 Java projects, 10 runs	EXAM 57.3% to 35.7%; Top-5 90.8%
[2] Makiyah et al., 2026	Network (SDN)	PPO reinforcement learning with severity-based reward	Mininet, Faucet, Prometheus; 3 topology sizes	Recovery 2.6 vs 4.8 steps (rule-based)

Study	Layer	Main technique	Test setting	Headline result
[3] Johnphill et al., 2026	Operating systems	Naive Bayes log classifier with threshold-triggered fixes	Logs from 4 OS; Docker, Kubernetes; 10 runs	MTTR 34.1 s vs 54.2 s (rules), 89.4 s (manual)
[4] Aldrini et al., 2024	Factory (smart manufacturing)	Literature review and conceptual model	About 256 papers, 2010–2022	Gaps: incipient and multiple faults; hybrid models

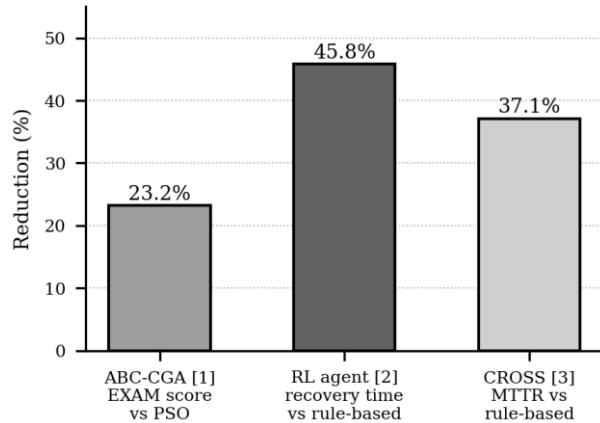


Fig. 2. Reduction against the closest compared baseline in each study. The three measures differ, so bars show size of gain only, not a ranking of methods.

The three method studies differ in how they decide. Study [1] ranks code statements with a search that mixes several scores. Study [2] learns a policy from rewards, so it can pick different actions for different faults. Study [3] uses a simple classifier plus fixed thresholds, which is easy to explain and cheap to run but not adaptive. Study [4] shows that the same trade-off exists in factories: model-based methods are explainable but need accurate equations, while data-driven methods need data but handle complex plants.

V. CRITICAL REMARKS

1. What the Studies Agree On

- Automatic recovery beats purely manual or purely rule-based recovery in every reported test.
- Good monitoring is a foundation: Prometheus telemetry appears in both [2] and [3].

- Several faults at once and slowly developing faults remain the hardest cases [1], [4].
- Simple, interpretable parts are kept inside the smarter systems, such as SBFL scores in [1] and thresholds in [3].

2. Points Readers Should Check

While reading the papers closely we noticed several details that affect how far the numbers can be trusted. We list them to encourage careful reading, not to dismiss the work.

- Study [1]. The abstract reports a 12.1% Top-5 gain over PSO, which matches Table 12 (88.4% vs 76.3%) but not Tables 7 and 8 (90.8% vs 89.4%). The text says Closure and Commons Math exceed 800,000 lines, while Table 3 lists about 90K and 85K. The p-values for GA and PSO differ between two tables.
- Study [2]. The abstract says fault-appropriate actions were chosen in over 85% of fault timesteps. The action table supports this only if any healing action counts; the single best action is chosen about 47–62% of the time. Training seems to use logged data, so repairs may not change the next state as they would on a live network. The fat-tree is described with both 16 and 64 hosts.
- Study [3]. The paper states a 37% MTTR reduction against manual scripts, but from its own Table 1 that figure matches the rule-based baseline (about 62% for manual). Placeholders “X% CPU and Y MB” remain in the text. Fixes on macOS and Windows, and some on Android and Linux, were simulated, so the MTTR for those paths may not reflect real repair time.
- Study [4]. As a survey it reports no experiments, and its model is conceptual rather than tested on a plant.

3. Common Weaknesses

All three technical studies were validated in controlled settings: a benchmark, an emulator, or recorded logs. None shows a self-healing loop running for a long time on a live production system. Baselines are also limited, since [2] could not reimplement earlier RL methods and [3] built its rule baseline from the same logs it later tested on.

VI. RESEARCH GAPS AND FUTURE DIRECTIONS

- Predictive healing. The four studies react after a fault. Using trends in logs, metrics and sensors to predict failures and repair early is largely open, although predictive maintenance is a strong theme in [4].
- Real-world validation. Tests on physical switches, real controllers, production services and attack scenarios are needed [2], [3].
- Multiple and slow faults. Methods must handle overlapping faults and gradual degradation [1], [4].
- Hybrid methods. Combining physical models, learning and search, as suggested in [4], could balance accuracy and explainability.
- Safe automatic repair. Self-healing actions can themselves cause harm. Sandboxing, limits on actions and human override are needed [3].
- Explainability. Developers and operators need to know why a statement or action was chosen [1].
- Shared benchmarks and metrics. Common datasets and measures would make results from different layers comparable.
- Cost and scale. Better speed through parallel evaluation and lightweight models is needed for large systems [1].

VII. CONCLUSION

This review compared four recent studies on self-healing at the level of code, networks, operating systems and manufacturing plants. Together they show that a monitor-detect-diagnose-repair loop can reduce debugging effort and recovery time compared with manual or fixed-rule approaches. At the same time, the evidence comes mostly from

benchmarks and simulations, some reported figures need rechecking, and prediction before failure is rarely attempted. Future AI-powered predictive self-healing software should combine accurate early warning, explainable decisions, safe and tested repair actions, and validation on real systems.

Acknowledgment

The Authors thank the Department of Computer Engineering, Jaihind College of Engineering, SPPU, Pune, for its support. This review is based on published work; all results quoted belong to the original authors.

REFERENCES

1. D. Ghosh and J. P. Singh, "Hybrid artificial intelligence for multi fault diagnosis in software systems using chaotic genetic algorithm and artificial bee colony optimization," *Discover Applied Sciences*, vol. 8, art. 450, 2026, doi: 10.1007/s42452-026-08374-x.
2. E. H. Makiyah, M. N. Rasool and F. A. Rawdhan, "A reinforcement learning framework for self-healing fault recovery in intent-based SDNs," *Discover Networks*, vol. 2, art. 14, 2026, doi: 10.1007/s44354-026-00028-z.
3. O. Johnphill, A. S. Sadiq, O. Kaiwartya and M. A. Taheir, "CROSS: a cloud-native approach to automated remediation and self-healing in cyber-physical systems," *Cybersecurity*, vol. 9, art. 124, 2026, doi: 10.1186/s42400-026-00549-8.
4. J. Aldrini, I. Chihi and L. Sidhom, "Fault diagnosis and self-healing for smart manufacturing: a review," *J. Intelligent Manufacturing*, vol. 35, pp. 2441–2473, 2024, doi: 10.1007/s10845-023-02165-6.
5. J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal policy optimization algorithms," *arXiv:1707.06347*, 2017.
6. R. Just, D. Jalali and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Software Testing and Analysis*, 2014, pp. 437–440.
7. R. Abreu, P. Zoetewij and A. J. C. van Gemund, "On the accuracy of spectrum-based fault

- localization," in Proc. TAIC PART-MUTATION, 2007, pp. 89–98.
8. D. Ghosh, R. Sharman, H. R. Rao and S. Upadhyaya, "Self-healing systems: survey and synthesis," *Decision Support Systems*, vol. 42, no. 4, pp. 2164–2185, 2007.
 9. O. Johnphill, A. S. Sadiq, O. Kaiwartya and M. Aljaidi, "An intelligent approach to automated operating systems log analysis for enhanced security," *Information*, vol. 15, art. 657, 2024.
 10. Y. Ma, Y. Guo, R. Yang and H. Luo, "FRRL: a reinforcement learning approach for link failure recovery in a hybrid SDN," *J. Network and Computer Applications*, vol. 234, art. 104054, 2025.