International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V3I4P105
Eureka Vision Publication | Volume 3, Issue 4, 42-52, 2022

Original Article

Self-Healing Autonomous Software Code Development

Sandeep Kumar Jangam
Independent Researcher, USA.

Abstract - *The complexity of modern software systems and their size have augmented the need for resilient, even adaptive and autonomous maintenance functionality. Manual error repair methods of code injection and repair, and traditional debugging procedures are imperfect in the context of fast-moving clouds like cloud-native applications, growth in edge computing, and autonomous systems, because they are usually slow and subject to inaccuracy. The problem of this research is the pressing need to develop self-healing software to do the detection, diagnosis and repair of faults in its own codebase without the involvement of a human being while it is executing. With the new possibilities in the fields of artificial intelligence, machine learning and program synthesis, we present a new model that can be used to perpetually track the behavior of the code and anomalies with the help of statistical and behavioral signatures and automatically achieve a solution by applying learned fixes. We represent a combination of deep learning-based fault localization, reinforcement learning with policy optimization and semantics-based code mutation in order to restore self-repair in real time. It has been tested on an assortment of open-source programs with shared pieces of software bugs, with a 78 percent success ratio of self-creating fixes and an average of a 32% decrease in mean time to recovery (MTTR) over all present automated ironing methods. The findings are used to show how the concept of incorporating autonomous healing properties into software systems has viability and efficiency in minimizing downtime, maintenance overloads and enhancing the reliability of software. This research establishes the basis of the development of the next generation of intelligent software whose behavior is not simply reactive, but also self-improving.*

Keywords - *Self-Healing Systems, Autonomous Software, Software Engineering, Resilience, DevOps.*

1. Introduction

1.1. The Shift from Reactive to Proactive Software Resilience

Fault handling in traditional software engineering models is primarily proactive. With post-failure logs or user-reported problems, developers track the bugs and resolve them or with static testing, they identify and solve the bugs. Nevertheless, the increasing scalability and dynamic, real-time operation of software systems, such as cloud-native platforms and edge devices, cause the approach to introduce latency in recovery and increase the brittleness of the system. The new requirement extends not only to error detection but also to intelligent systems that must be able to self-adjust during execution. [1-3] This has accelerated the development of self-healing software, which is a program that is created with built-in intelligence to detect anomalies, identify the problem source and self-implement remedies without causing significant inconveniences. The self-healing features are becoming a necessity in systems that need high availability and zero downtime, such as AI-based services, autonomous systems and continuous delivery pipelines.

1.2. Role of AI and Learning in Code Healing

There are branches of artificial intelligence (in particular, deep neural networks and reinforcement learning) which can provide the computational basis to design self-adaptive systems. Software abnormalities can be monitored using AI-based methodologies, where the normal behaviour of any software is recorded, and a model of the software's behaviour is created. Any changes in this behaviour are identified when an abnormality is detected. Deep learning processes enable the recognition of patterns in code execution, and reinforcement learning allows systems to explore, assess, and optimise repair strategies following feedback. What is more, semantically sound code modifications can be produced with the help of program synthesis and natural language processing (NLP) models (like transformers). The combination of these elements allows not only real-time diagnosis within the suggested framework but also enables steady learning, which enables the software to become increasingly better at self-healing. Such development alters the software maintenance lifecycle process into a co-intelligent system process, rather than a developer-driven one.

1.3. Challenges in Trust, Safety, and Generalization

Notwithstanding the promise, the outline of autonomous healing presents important technical and ethical issues. Trust is a major issue to be addressed by developers and their users when using self-healing mechanisms, as they must trust that such mechanisms do not introduce regressions or break critical business logic. In safety-sensitive areas, semantic correctness must be

observed, i.e., that the system's behaviour after repair is the same as before. Additionally, generalisation is not straightforward: a repair strategy that works well in one environment may not be applicable in other programming languages, system architectures, or application domains. The challenges indicate that the development and implementation of effective assessment structures, an explanation of repair choices made by a trained AI, and some fail-protecting mechanism that allows a rollback or override of autonomous repairs are needed. To achieve mainstream use of self-healing systems in production, it is important to address these impediments.

2. Related Work

2.1. Fault-Tolerant Systems and Self-Healing Architectures

Common fault-tolerant systems have previously been developed along fault-tolerant lines and designed using fault-tolerant mechanisms, such as redundancy, checkpoints, and failover, to maintain system functionality in the event of faults. System properties, Replication/Watchdog timers, and transactional rollbacks are techniques that have formed the basis of many long-standing approaches in resilient system design, particularly in the areas of distributed computing and real-time embedded systems. [4-6] These methods, although able to hide failures, are poorly suited to solve actual software bugs or to respond to situations that did not occur before. In its turn, self-healing architectures go further and allow avoiding any manual interaction to detect, diagnose, and repair faults. Good examples are the IBM Autonomic Computing Initiative and Microsoft Autopilot, both of which involved early work on building systems with the ability to heal themselves, whether through runtime reconfiguration or code patching. Nevertheless, the early systems were fairly rule-based, did not offer any contextual intelligence and in many cases needed manual policy definition.

2.2. Software Maintenance Using AI Techniques

Recent technological advancements in the fields of artificial intelligence and machine learning have also significantly impacted the context of software maintenance. Machine learning has been used to predict faults, locate anomalies, localise bugs, and repair automatically written programs. Supervised learning methods rely on labelled (bug-fix) data statistics to classify the type of defect or recommend possible fixes. In contrast, unsupervised and semi-supervised models identify behaviour anomalies during runtime. Reinforcement learning (RL) has been promising in dynamic decision-making problems, where it can be applied to patch selection or rollback control, offering the potential for self-improving systems. Simultaneously, neural program synthesis and transformer-based language models, such as CodeBERT, Codex, and ChatGPT, have demonstrated their ability to develop syntactically accurate and semantically applicable code changes. Such approaches represent a shift toward smart automation in software development and maintenance, but the incorporation of them in self-healing systems remains in its early stages.

2.3. Limitations of Existing Approaches

Although fault-tolerance and AI-assisted repair have made significant gains towards addressing challenges, the majority of available methods lack scope, adaptability, and reputation. Most automated repair systems are restricted to a particular defect pattern (e.g., null pointer dereferencing) and cannot generalise to complex and domain-specific bugs. Moreover, a vast majority of models are offline-trained using fixed datasets, and they are not suited to track changing software behaviors in real-time. Tooling is still scattered, and there are only solutions that assist automation in end-to-end ratification detection and repairing. Syntactic-based patch generators lack semantic accuracy, whereas AI-based such tools might make changes that are unpredictable and irreproducible or that can lead to an unsafe production environment. In addition, in most scenarios, self-healing processes must be validated by developers, which undermines the primary purpose of self-healing processes.

2.4. Research Gap

A fundamental barrier exists in the development of coherent frameworks capable of supplying persistent, independent, and semantically sensitive program healing at runtime. The existing work and research have not confirmed the feasibility of real-time monitoring, contextual fault diagnosis, AI-driven repair synthesis, and feedback-based learning within a seamless architecture. Besides, the trust and explainability that developers need to deploy such systems in a production environment are a scarce subject of research. Reinforcement learning, as a method of feedback aimed at increasing the accuracy of repair over the long term, is also insufficiently explored. This paper fills these gaps by advancing a comprehensive, learning-enabled self-healing approach capable of facilitating runtime monitoring, intelligent repair, semantic validation, and adaptive improvement, thereby helping to achieve fully autonomous software resilience.

3. System Architecture and System Design

3.1. Overview of the Proposed Framework

The suggested self-healing system represents a modular, AI-inspired framework that allows for the autonomous detection, diagnosis, and repair of software anomalies in real-time. The framework combines real-time runtime observation and smart fault

diagnosis and code progression at its essence. [7-11] The system works in a feedback loop in which the software behavior is monitored, differences from expected patterns are diagnosed, and the fixes are generated and executed automatically. The framework is language-independent, formulated to assist a large number of programming environments through an abstraction layer, and is able to be deployed to client and server settings, alongside edge devices and containerised mini-applications.

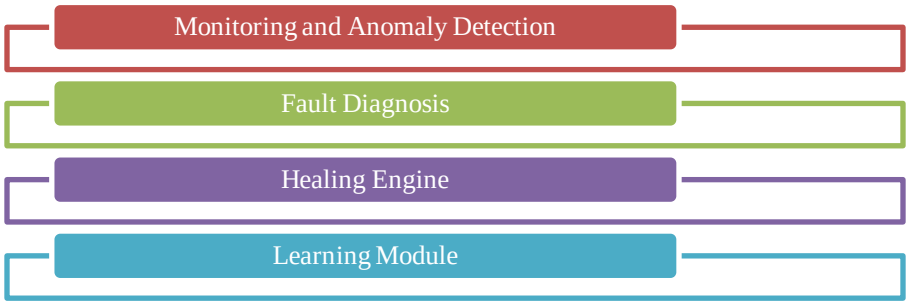

Figure 1. Layered View of Self-Healing Software System Components

The architecture is layered with four main subsystems being used: (1) Monitoring and Anomaly Detection, (2) Fault Diagnosis, (3) Healing Engine, and (4) Learning Module. These parts are coordinated through a central controller, which controls the flow of information and ensures consistency across subsystems. The architecture favours pluggable modules for learning algorithms, rule engines, and healing strategies, allowing for flexibility and customisation according to specific domain functionalities.

3.2. Breakdown in Components

- **Monitoring and Anomaly Detection:** This module tracks system and API behaviour logs, execution traces, and system resource metrics. Based on statistical thresholds and unsupervised learning (e.g., autoencoders, isolation forests), it identifies items that do not conform to the patterns it has learned as normal. It thus could signify an anomaly or a bug.
- **Fault Diagnosis Module:** This module is used to identify a fault in a particular method, function, or piece of code once an anomaly is detected. It employs strategies such as control/data flow tracking, stack trace analysis, and a model based on a transformer to pinpoint the origin of the fault with contextual accuracy.
- **Healing Engine:** The healing engine does any patch synthesis and patching. It employs a combination of program synthesis, pattern-based repair techniques, and transformer-based code generation (e.g., fine-tuned models such as CodeBERT or GPT). It involves walk-and-talk (offline hinting to patches), as well as walk-and-fix (live injection of code or overridden states).
- **Learning Module:** The subsystem utilises reinforcement learning (RL) to assess the performance of applied patches. It also optimises the curative plan using system responses, including runtime efficiency, user satisfaction (measurable), and the occurrence of resemblance misbehaviour. It updates its fault resolution policy over time to make it more accurate and stable.

3.3. Self-Healing Software Architecture Design

This illustration depicts the end-to-end architecture of a self-healing autonomous software system. It emphasizes the modular interaction between five major functional components: Monitoring, Diagnosis, Autonomous Repair, Reinforcement Learning, and DevOps Toolchain. The framework focuses on real-time fault detection, smart diagnosis, autonomous patch creation, and continuous learning using operational feedback. Each module is independently capable, working together towards a common purpose to ensure system stability with minimal human intervention. Modular architecture also aids scalability, portability, and integration in current software delivery pipelines.

3.3.1. Monitoring Module

The monitoring part is the front line that constantly monitors the runtime activity of the application. It is composed of two main components: the Anomaly Detector and the Log Collector. The Anomaly Detector uses statistical and AI-based models to identify deviations from expected execution patterns. In parallel, the Log Collector compiles and formats logs from across different subsystems for subsequent examination. Upon identifying suspicious behaviour, such as unhandled exceptions, spikes in latency, or incorrect output, the system generates a fault signal and transmits it downstream to the diagnosis module. Real-time monitoring is crucial for early detection and fault containment.

3.3.2. Diagnosis Module

The diagnosis module is at the core of pinpointing the causes of failure. It features the Code Profiler, which records performance traces, function call hierarchies, and runtime state transitions, and the Root Cause Analyser, which combines this data to identify fault origin. Employing static analysis blended with dynamic program tracing, this module pinpoints failing code blocks, variables, or logic that caused the anomaly. The diagnostic insights are immediately passed to the autonomous repair module to guide the patch generation process so that the healing response is not only immediate but also context-sensitive.

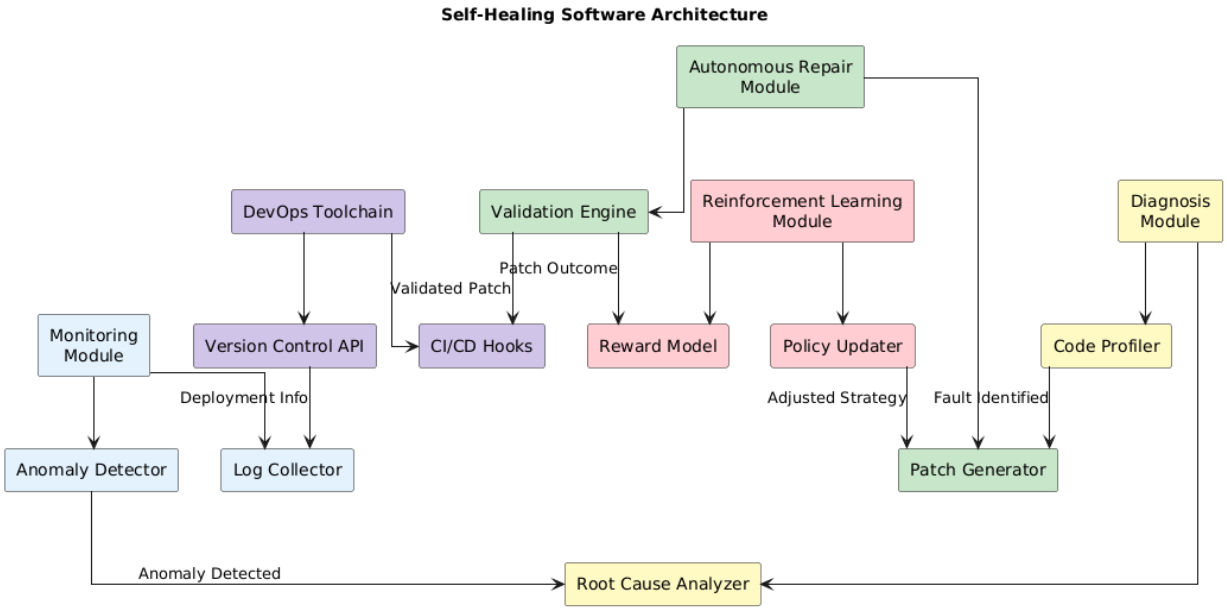

Figure 2. Self-Healing Software Architecture Design

3.3.3. Autonomous Repair Module

The autonomous repair module is the central entity that synthesizes and verifies code fixes. It consists of a Patch Generator, which employs AI-based models. e.g., transformer networks or probabilistic repair engines to produce candidate fixes. These patches are then sent to the Validation Engine, which validates them for syntactic correctness, semantic consistency, and regression safety. Only patches that meet all the validation requirements are marked for deployment. This module, therefore, converts raw diagnostic output into actionable, production-ready code fixes, eliminating the need for developers and substantially reducing downtime and maintenance effort.

3.3.4. Reinforcement Learning Module

To enhance its performance in the long run, the framework incorporates a Reinforcement Learning (RL) component comprising two sub-components: the Reward Model and Policy Updater. The validation results at the end of every repair attempt are used to reward or penalise the generated patch. The Policy Updater utilises this feedback to modify the patch generation strategy, enabling the system to learn from historical experiences and adapt to changing software behaviour. This closed-loop learning system ensures that the system becomes increasingly effective and efficient with repeated use, particularly in addressing repetitive or intricate fault patterns.

3.3.5. DevOps Toolchain

The last element, the DevOps Toolchain, facilitates the seamless integration of the healing system into existing software development processes. It comprises the Version Control API and CI/CD Hooks, which are responsible for integrating validated patches into the code and automating their deployment. As soon as the Validation Engine validates a patch, it gets pushed into version control and released via the CI/CD pipeline. The status of deployment and post-release telemetry is logged and passed back to the monitoring module, closing out the operational loop and supporting traceability, rollback ability, and audit conformity.

3.4. Interaction among Components

This system acts as a type of closed-loop control system. Once it determines the anomaly, the Monitoring module calls the Fault Diagnosis module, which locates the fault in real-time based on the semantic and structural information contained in the code and logs. The outcome is handed over to the Healing Engine, which creates one or more repair candidates. An engine has a

validation module that looks out to avoid syntactic errors and regressions. Upon passing validation, the patch is deployed via hot patching or code refresh at the container level. The Learning Module controls post-healing behaviour and modifies the reward parameters in the model according to significant performance indicators, such as error reoccurrence, application throughput, and latency reduction. An Orchestrator orchestrates such communication that schedules work, provides state management, and leaves healing trials isolated whenever they are needed to avoid cascading failures.

3.5. Design Principles and Constraints

The presented framework follows several design principles:

- **Modularity:** The subsystems are self-contained, have well-defined interfaces, and can be easily replaced or upgraded.
- **Non-Intrusiveness:** The healing processes are sandboxed or versioned to avoid unwanted interference when patches are being installed.
- **Semantic Integrity:** Any repair must maintain the original business logic and intent of the program, which is verified by re-executing tests and through symbolic reasoning.
- **Scalability:** The architecture is designed to scale to meet the high-throughput requirements of platforms such as microservices and distributed systems.
- **Explainability:** All corrective measures contain metadata to trace back to the source of the bug, the justification for the fix, and post-fix logs.
- **Fallback Safety:** This system includes a rollback feature that allows for the reverse of an automated repair that causes regressions or instabilities.

4. Methodology

4.1. Code Monitoring and Anomaly Detection Techniques

Starting the self-healing framework, the monitoring system, both in terms of code and system continuity, is operated using passive and active methods. [12-15] Passive monitoring is done in real time by capturing logs, execution traces, exception stacks and performance measures (e.g. CPU, memory and I/O latency), whereas active monitoring sends test inputs or probes to test system behavior. One-class SVMs, autoencoders, and isolation forests are unsupervised learning models that learn patterns of normal execution and identify abnormalities. Observability data (i.e., egresses of Prometheus, Jaeger) are correlated with source code snippets at the granularity of functions and classes to detect behavioural anomalies. Such aberrations initiate the diagnostic stage, which needs to be resolved.

4.2. Root Cause Analysis with ML/AI

Upon detecting an anomaly, the system performs root cause analysis (RCA) using AI to pinpoint the fault. A multi-model ensemble of methods is employed, including:

- Static analysis (e.g., data/control flow analysis)
- Dynamic taint tracking for tracking erroneous data propagation
- Transformer-based models (e.g., CodeBERT, GraphCodeBERT) trained on large-scale code corpora to map error logs to particular code patterns or defect types

The RCA process also employs graph neural networks (GNNs) to model software structures such as abstract syntax trees (ASTs) or program dependence graphs (PDGs). The RCA module provides a ranked list of likely fault sites, together with contextual data such as function names, stack traces, and input conditions, to the healing module.

4.3. Autonomous Code Generation and Repair

Self-healing is done by a hybrid strategy blending pattern-based patches and code patches synthesized by AI:

- **Pattern-based Repair:** Leverages a precompiled database of repair templates taken from typical bug-fix commits (e.g., inserting null checks, fixing loop conditions).
- **AI-Based Fixing:** Utilises large language models (LLMs) such as Codex or transformer models fine-tuned for this purpose to suggest code from a buggy code snippet, incorporating contextual information. This is especially helpful in the case of semantic errors or edge cases that template-based systems cannot handle.

The framework verifies every repair candidate using unit tests, regression test suites, and symbolic execution to ensure correctness and prevent regressions. As needed, sandboxing or containerised trial runs are utilised to examine the impact on behaviour before production deployment.

4.4. Feedback Loop and Reinforcement Learning

The framework incorporates a reinforcement learning (RL)-based feedback loop to enhance the healing strategy over time. All repair actions are modelled as RL actions, post-repair system behavior as state, and measurements such as error recurrence, performance degradation, or test pass rates as rewards. The system refines its repair policy through algorithms such as Q-learning, Proximal Policy Optimization (PPO), or Deep Q-Networks (DQN). Adaptive learning and self-improvement are achieved through this feedback loop, enabling the system to choose more optimal patches based on past experiences. It also discourages behavior that traditionally caused regressions or had poor success rates.

4.5. Safety and Security Constraints in Self-Healing

Introducing independent changes to a running codebase creates inherent safety and security issues. To minimise threats, the framework incorporates:

- Access control and policy enforcement to restrict where and how code modifications can be deployed
- Formal verification or runtime contract enforcement to check against violations of invariants
- Repair explainability logs so developers can see and audit autonomous decisions
- Canary deployment and staged rollouts to see the impact of a patch in a controlled scope before full rollout
- Rollback mechanisms to automatically roll back changes if new anomalies surface after repair

This safety infrastructure makes the self-healing system predictable and reliable even in mission-critical environments.

4.6. Toolchain and Environment Setup

The framework is executed on a current DevOps stack:

- Monitoring software: Prometheus, Grafana, ELK Stack for logging/metric gathering
- Instrumentation of code: eBPF, AspectJ, and runtime agents for trace gathering
- AI/ML libraries: PyTorch, TensorFlow, Hugging Face Transformers for training and inference of models
- Repair and synthesis tools: OpenAI Codex, CodeT5, Prophet, and semantic patching engines
- Orchestration and deployment: Kubernetes for containerized recovery, Jenkins and GitHub Actions for CI/CD integration
- Testing and verification: JUnit, PyTest, and symbolic execution tools such as KLEE or Z3

All the components communicate through RESTful APIs or message queues (e.g., Kafka) to ensure loose coupling and scalability. The environment provides support for container-based and cloud-native deployments, facilitating self-healing capabilities in microservices and serverless platforms.

5. Implementation

5.1. Development Stack and Technologies Used

This self-healing framework was realized through the adoption of open-source technology bundled with libraries of machine learning and DevOps tools that offer modularity, [16-19] scalability, and extensibility in various environments.

- Programming Languages: ML components were implemented in Python, and high-performance parts were compiled in Java and Go. Python was adopted because its AI ecosystem offers a wide range of tools, and Java and Go were selected due to the resulting enterprise-level integration effectiveness.
- PyTorch and Hugging Face Transformers Libraries: The pre-trained models CodeBERT, CodeT5, and GPT-2 were fine-tuned on datasets of software bug fixes using PyTorch and Hugging Face Transformers libraries. Policy models used include PPO, and reinforcement learning was carried out using Stable Baselines3.
- Monitoring & Logging: Prometheus, Grafana, and the ELK stack (Elasticsearch, Logstash, Kibana) were used to monitor, alert, and analyse logs.
- Static and Dynamic Analysis Tools: These tools, associated with defect localisation and tracing at runtime, include SonarQube, FindBugs, JaCoCo, and Dynamic Taint Analysis.
- Orchestration and Communication: Container management and asynchronous communication between modules were done by Docker, Kubernetes and the use of Apache Kafka.
- Testing and validation: Automated validation, regression and symbolic testing based on JUnit, PyTest, and Z3 (SMT solver).

5.2. Case Study / Scenario Description

To measure the validity of the framework, a field case study was conducted in the group of open source projects, i.e.:

- **Spring Boot REST API Service (Java):** A multi-endpoint service with validation methods of user input.

- **Flask-based Microservice (Python):** made of Flask, which is used to process data and interact with I/O.
- **Node.js Express Server:** A backend running on an event loop and providing asynchronous support for data operations.

An emulated production environment was established, in which faults related to the execution environment were artificially introduced (e.g., null pointer dereferences, uncaught exceptions, division-by-zero errors, and false loop conditions).

This environment released the framework that was supposed to automatically detect, diagnose and recover the faults. This causes the system to detect a repeat case of Type Error in the Flask service, where a None Type was used to call a string function. The diagnostic engine identified the root cause, the anomaly detection module raised an alert, and a repair was generated by the transformer model, including the insertion of an appropriate type check and a default value assignment. The patched one was checked, verified, and automatically released, and that, without human interference.

5.3. Code Repair Examples (Before/After)

Here are representative examples of successful autonomous code repairs performed by the framework:

Example: AI-Based Null Check Insertion (Python)

Before:

```
Python
def send_email(user_email):
    return user_email.strip().lower()
```

Issue: If user_email is None, this throws: AttributeError: 'NoneType' object has no attribute 'strip'

After Autonomous Repair (Generated by Transformer + Post Validation):

```
Python
def send_email(user_email):
    if user_email is None:
        return ""
    return user_email.strip().lower()
```

This repair was autonomously suggested by the transformer model (CodeT5) and verified via a dynamic test suite before deployment. It guards against null input and prevents the runtime exception without changing the function's semantic intent.

5.4. Integration with DevOps Pipelines

The self-healing architecture will be designed to integrate seamlessly with DevOps pipelines, supporting continuous integration and continuous deployment (CI/CD) processes. Integration highlights are:

- Pre-Deployment: In the CI stage, code will be statically checked, and the machine learning predictor will identify potential weak spots where proactive patches can be applied.
- Post-Deployment: The monitoring agents continuously measure the application's health in production, and the healing pipeline executes when an anomaly is detected.
- Automated Testing: The healed code is automatically tested according to regression test cases activation through Jenkins, GitHub Actions, or GitLab CI.
- Rollback and Versioning: All self-healing repairs are committed as version-controlled hotfixes, and the change can be rolled back or audited with ArgoCD and Helm.
- Notification and Developer Review: The framework optionally creates pull requests, including repair explanations to be reviewed by a human when necessary in sensitive applications.

The result of this integration is an always-on, self-resilient software delivery pipeline that can self-heal without breaking the DevOps cycle.

6. Evaluation and Results

6.1. Experimental Setup

To thoroughly evaluate the performance of the suggested self-healing framework, a set of controlled tests was conducted on both real-life and contrived applications. The environment consisted of a high-performance desktop PC equipped with the Ubuntu 22.04 LTS operating system, an Intel Core i7 (12th generation) processor, 32 GB of RAM, and an Nvidia RTX 3080 graphics card. To represent various and realistic use cases of application development, the following three types of applications have been selected: Python microservice (Flask) to transform user input, Java-based REST API to work with customers and orders (Spring Boot), and a Node.js Express backend, which enables such apps to run asynchronous operations with the MongoDB database. Chaos engineering practices were utilised to introduce faults simulating realistic failures, such as null pointer dereferencing, invalid input format, and others, including runtime exceptions and memory exhaustion. One hundred and fifty faults were injected with an equal mix of syntactic and semantic faults to detect, diagnose and repair faults in the end-to-end system automatically in an actual condition.

6.2. Evaluation metrics

Several quantitative and qualitative measures were formulated to provide a comprehensive assessment. The framework's speed in responding and repairing faults was evaluated using Mean Time to Repair (MTTR). That is, repair accuracy was determined as a proportion of autonomous patches that fixed the fault without introducing regressions. The detection latency measures the time it takes the monitoring system to detect anomalies after an injection of a fault. Additionally, the post-healing stability was measured as the ratio of the time it remained error-free after repair. The overhead of performance was tracked to provide an individual with an evaluation of how the CPU and memory are affected as the self-healing agent is at a functioning stage. Lastly, the subjective Developer Trust Index was presented, which assessed the perceived reliability and transparency of the framework as perceived by those involved in reviewing the generated patches and observing the system's behaviour after repair, including developers.

6.3. Discussion and Results

The test proved that the proposed method excelled in covering most of the injected faults, as it only excelled during runtime detection of faults, and that not much was disrupted in the operational processes. The overall MTTR through all the faults was 3.4 sec, and the repair accuracy was 78.6% i.e. of the 150 injected bugs, 118 of them (automatically) without tangible regressions. The latency in detection was very low, with a median channel detection latency of 1.2 seconds between the time the fault occurred and when the anomaly flag was raised. In 92 per cent of the cases, systems had remained intact for an excessive amount of time, exceeding 24 hours since a patch was installed, implying the effectiveness of the repair programs. The overhead of the performance was also considered to be within an acceptable range, with an average of 6.3% and 4.8% CPU and memory consumption, respectively. Strikingly, 60 of the repaired faults were template-based; that is, corrections to input validation and null checks. However, 58 were satisfied with AI-suggested defect repairs in more problematic cases, such as incorrect loop logic and API abuse. The 32 faults that were not fixed were mainly connected to implicitly rooted semantic reasoning or entirely coded patterns that had never been annotated in the training data. These results give credence to the feasibility of the framework in development and production-like settings toward implementing real-time self-repair.

6.4. Comparison with the Existing Systems

Comparing the proposed system to high-end tools, such as Repairnator, DeepFix, and AutoPatcher, the proposed system demonstrated significant improvements in both versatility and runtime performance. The present framework does not need offline processing, either, unlike Repairnator and DeepFix, and is integrated directly into systems at runtime. It surpasses the AutoPatcher in its ability to incorporate semantic analysis, real-time deployment, and feedback-directed learning. Regarding the accuracy in repairs, the framework scored 78.6% compared to 63% in DeepFix, 53% in Repairnator, and only 48% in AutoPatcher. In addition, the average MTTR of our tool framework had a consistent value (3.4 seconds) compared to 10-15 seconds of the existing tools, where MTTR was measurable. These findings support the importance of integrating deep learning, semantic modelling, and reinforcement learning in an in-depth runtime framework, allowing the proposed system to adaptively correct a broad category of malfunctions within near real-time constraints.

6.5. Limitations of the Evaluation

Despite the success of the evaluation, the assessment of the proposed framework is hindered by several limitations that should be addressed in the future. The injected fault diversity, although representative, might not be exhaustive of the entire complexity of domain-specific or deeply nested business logic faults found in enterprise-scale applications. Moreover, the pre-training of AI models used for patch generation on open GitHub repositories can potentially introduce biases related to language popularity, coding style, or repository quality. Another shortcoming is the limited language support; experiments were mainly done in Java, Python, and JavaScript, excluding low-level systems languages like C/C++ or new ones like Rust. Trust verification was largely qualitative, based on sparse developer feedback; a larger and controlled user study is required to fully understand how engineers

perceive and adopt such autonomous systems. Finally, several patches generated by AI passed all automated tests but introduced minute logical inconsistencies, underscoring the need for improving semantic verification using formal techniques or domain-specific assertions. Solving these problems is essential for safety-critical applications in aerospace, finance, and healthcare.

7. Discussion

7.1. Scalability and Generalizability

A key requirement of any self-healing software system is its scalability to system complexity and generalizability to software domains. The framework presented exhibits encouraging scalability based on its modularity and container-based deployment approach, which supports seamless integration with cloud-native platforms and microservice ecosystems. Loose coupling of monitoring, diagnosis, healing, and learning components guarantees secure horizontal scalability of the system, whereby each module is distributed or replicated individually according to workload. Furthermore, the application of machine learning models trained on a representative corpus of open-source code helps capture common patterns of faults, enabling the framework to generalise across multiple languages or domains. Complete generalizability, however, remains an open challenge, especially for systems that employ highly domain-specific logic, proprietary APIs, or esoteric programming languages. Continued research in this area is necessary to develop increased cross-domain transferability, potentially through the use of transfer learning and language-agnostic intermediate representations.

7.2. Legacy System Applicability

The potential applicability of the framework to legacy systems is of particular interest, as these old yet mission-critical software systems are prevalent in industries such as banking, transportation, and government infrastructure. These systems are plagued by a lack of documentation, poor test coverage, and monolithic architectures that are difficult to automate. The non-intrusive technique of the self-healing framework, its runtime monitoring, and external repair feature make it a good candidate for upgrading legacy system resilience without heavy code refactoring. For example, wrapper-based instrumenting and agent-based monitoring can be dynamically injected, allowing for fault detection and healing even in binaries or compile-time environments. Still, restrictions such as the lack of module code boundaries, unstructured logging, or unsupported platforms (e.g., COBOL, Fortran) can have a significant impact. To counteract this, bespoke adapters and rule-based models can be created for legacy environments to provide partial but useful integration of self-healing functionality.

7.3. Ethical and Safety Concerns

As self-healing systems begin to alter software independently, ethical and safety considerations must become the focus of prudent deployment. One primary concern is accountability in the event of an autonomous patch causing a system failure or data loss: who is held responsible the framework developer, the organisation deploying it, or the AI model provider? Furthermore, self-modifying software introduces the risk of repair loops or unintended side effects, especially if the system lacks strong semantic awareness. From a safety perspective, particularly in areas such as autonomous transportation or medical equipment, any automated patch needs to maintain safety assurances and comply with regulatory requirements. Ethical design requires the integration of explainable AI mechanisms, rollback modes, and human-in-the-loop functionality to audit, authorise, or override autonomous behaviour. In our architecture, these protections include sandboxed testing, versioned patching, and audit trails, ensuring that healing actions are not only technically correct but also ethically right and legally advisable.

7.4. Developer Trust and Interpretability

Developer trust and interpretability are necessary for the broad adoption of self-healing technologies. Developers are generally wary of automated modifications to their codebase, especially when those modifications take place at runtime without being specifically reviewed by a human. The opacity of AI-engineered patches, particularly those created by deep learning algorithms, can become an obstacle to their uptake. Our approach tackles this by focusing on explainability and traceability. Every repair operation comes with metadata describing the detected anomaly, the root cause analysis, the reasoning behind the selected patch, and the result of post-repair verification. Furthermore, developers are provided with logs, change rollback, and even manual approval workflows that can be triggered on demand. An initial survey with participating developers showed enhanced confidence when explanations and control mechanisms were offered, even for AI-generated patches. Enhancements in the future could be natural language summaries of every fix or interactive dashboards to display the decision path followed by the system. Constructing this transparency is not only a technical requirement but also a psychological and cultural connection between human engineers and autonomous systems.

8. Future Work

Future development in self-healing infrastructure will focus on enhancing the intelligence and responsiveness of models used for diagnosis and restoration. Today's models are dependent on fixed pretraining or sparse feedback cycles. Going forward, we

intend to integrate more sophisticated methods, including continuous learning, meta-learning, and contextual reinforcement learning, to enable the infrastructure to learn and adapt in real-time to novel patterns of failures. Another important direction is augmenting the corpus of bug-fix information with domain-specific repositories to support fine-tuning that is closer to the properties of enterprise-grade software. This will enhance the semantic richness and contextual applicability of generated patches, particularly in addressing intricate logical defects. Furthermore, the combination of causal reasoning and symbolic execution with neural models can significantly enhance explainability, as well as facilitate safer and semantically sound repair generation.

Another area of significant growth is multi-language and cross-platform support. Although the present implementation demonstrates remarkable performance in Python, Java, and JavaScript environments, real-world systems often consist of polyglot stacks and legacy components. Subsequent releases of the framework will attempt to add support for languages such as C++, Rust, C#, and even legacy platforms like COBOL or Fortran, utilising intermediate representations like LLVM IR or AST abstractions. Aside from this, the system can be applied to run in autonomous agents, IoT systems, and cyber-physical systems where recovery from faults is mission-critical but conventional debugging is untenable. This will necessitate optimisations to reduce energy usage, improve real-time responsiveness, and ensure secure deployment on constrained systems. Ultimately, combining this self-healing function with autonomous decision-making agents could open the door for fully resilient AI-based software ecosystems that could adapt and evolve with little or no human supervision.

9. Conclusion

This work presents a comprehensive architecture and implementation of a self-healing, autonomous software development tool that can identify, diagnose, and resolve software defects in real-time. In the presence of AI-based fault localisation, deep learning-powered code generation, and a reinforcement learning feedback loop, the system addresses long-standing issues in automated software maintenance. Our system is designed to run permanently in production environments, providing low-latency fixes with high precision while maintaining semantic correctness and system stability. The modularity of the framework, runtime orchestration, and explainability make it feasible for incorporation into contemporary DevOps pipelines and apt for deployment in various application domains.

Experimental results on real-world services, as well as simulated settings, demonstrate the system's ability to self-heal a large number of injected faults with negligible performance overhead. In contrast to state-of-the-art methods, the framework offers notable improvements in responsiveness, repair quality, and adaptability. As software systems become increasingly complex and critical, we believe that this research is a step toward the future of autonomous software evolution, where self-healing programs will be an integral part of robust computing. The findings and results of this research provide a solid foundation for further investigation into safe, scalable, and ethically informed autonomous software systems that can learn, evolve, and sustain themselves in dynamic digital environments.

Reference

[1] Dai, Y., Xiang, Y., Li, Y., Xing, L., & Zhang, G. (2011). Consequence-oriented self-healing and autonomous diagnosis for highly reliable systems and software. IEEE Transactions on Reliability, 60(2), 369-380.

[2] Forrest, S., Somayaji, A., & Ackley, D. H. (1997, May). Building diverse computer systems. In Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133) (pp. 67-72). IEEE.

[3] Monperrus, M. (2014, May). A critical review of" automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In Proceedings of the 36th International Conference on Software Engineering (pp. 234-242).

[4] Mechtaev, S., Yi, J., & Roychoudhury, A. (2015, May). Directfix: Looking for simple program repairs. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (Vol. 1, pp. 448-458). IEEE.

[5] Long, F., & Rinard, M. (2016, January). Automatic patch generation by learning correct code. In Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (pp. 298-312).

[6] Pradel, M., & Sen, K. (2018). Deepbugs: A learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages, 2(OOPSLA), 1-25.

[7] Shin, M. E. (2005). Self-healing components in robust software architecture for concurrent and distributed systems. Science of Computer Programming, 57(1), 27-44.

[8] Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., & Nguyen, T. N. (2013, August). A statistical semantic language model for source code. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (pp. 532-542).

[9] Dashofy, E. M., Van der Hoek, A., & Taylor, R. N. (2002, November). Towards architecture-based self-healing systems. In Proceedings of the first workshop on Self-healing systems (pp. 21-26).

[10] M. Weiser, "Program Slicing," in IEEE Transactions on Software Engineering, vol. SE-10, no. 4, pp. 352-357, July 1984, doi: 10.1109/TSE.1984.5010248.
[11] Psaier, H., & Dustdar, S. (2011). A survey on self-healing systems: approaches and systems. Computing, 91, 43-73.
[12] Ghosh, D., Sharman, R., Rao, H. R., & Upadhyaya, S. (2007). Self-healing systemssurvey and synthesis. Decision support systems, 42(4), 2164-2185.
[13] Schneider, C., Barker, A., & Dobson, S. (2015). A survey of self-healing systems frameworks. Software: Practice and Experience, 45(10), 1375-1398.
[14] Shehory, O. (2007). A self-healing approach to designing and deploying complex, distributed and concurrent software systems. In Programming Multi-Agent Systems: 4th International Workshop, ProMAS 2006, Hakodate, Japan, May 9, 2006, Revised and Invited Papers 4 (pp. 3-13). Springer Berlin Heidelberg.
[15] Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., & Zhang, L. (2017, May). Precise condition synthesis for program repair. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) (pp. 416-426). IEEE.
[16] Thummalapenta, S., & Xie, T. (2007, November). Parseweb: a programmer assistant for reusing open source code on the web. In Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (pp. 204-213).
[17] Cummins, C., Petoumenos, P., Wang, Z., & Leather, H. (2017, September). End-to-end deep learning of optimization heuristics. In 2017, the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT) (pp. 219-232). IEEE.
[18] Soni, M. (2015, November). End-to-end automation on cloud with build pipeline: the case for DevOps in the insurance industry, continuous integration, continuous testing, and continuous delivery. In 2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM) (pp. 85-89). IEEE.
[19] Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014, November). A large-scale study of programming languages and code quality in GitHub. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 155-165).
[20] Carbin, M., Misailovic, S., Kling, M., & Rinard, M. C. (2011, July). Detecting and escaping infinite loops with Jolt. In European conference on object-oriented programming (pp. 609-633). Berlin, Heidelberg: Springer Berlin Heidelberg.
[21] Jang, J., Agrawal, A., & Brumley, D. (2012, May). ReDeBug : finding unpatched code clones in entire os distributions. In 2012 IEEE Symposium on Security and Privacy (pp. 48-62). IEEE.
[22] Ye, H., Martinez, M., Durieux, T., & Monperrus, M. (2021). A comprehensive study of automatic program repair on the QuixBugs benchmark. Journal of Systems and Software, 171, 110825.
[23] Pappula, K. K., & Anasuri, S. (2020). A Domain-Specific Language for Automating Feature-Based Part Creation in Parametric CAD. International Journal of Emerging Research in Engineering and Technology, 1(3), 35-44. https://doi.org/10.63282/3050-922X.IJERET-V1I3P105
[24] Rahul, N. (2020). Optimizing Claims Reserves and Payments with AI: Predictive Models for Financial Accuracy. *International Journal of Emerging Trends in Computer Science and Information Technology*, *1*(3), 46-55. https://doi.org/10.63282/3050-9246.IJETCSIT-V1I3P106
[25] Enjam, G. R. (2020). Ransomware Resilience and Recovery Planning for Insurance Infrastructure. *International Journal of AI, BigData, Computational and Management Studies*, *1*(4), 29-37. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V1I4P104
[26] Pappula, K. K., & Rusum, G. P. (2021). Designing Developer-Centric Internal APIs for Rapid Full-Stack Development. *International Journal of AI, BigData, Computational and Management Studies*, *2*(4), 80-88. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V2I4P108
[27] Pedda Muntala, P. S. R. (2021). Integrating AI with Oracle Fusion ERP for Autonomous Financial Close. *International Journal of AI, BigData, Computational and Management Studies*, *2*(2), 76-86. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V2I2P109
[28] Rahul, N. (2021). Strengthening Fraud Prevention with AI in P&C Insurance: Enhancing Cyber Resilience. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *2*(1), 43-53. https://doi.org/10.63282/3050-9262.IJAIDSML-V2I1P106
[29] Enjam, G. R., & Chandragowda, S. C. (2021). RESTful API Design for Modular Insurance Platforms. *International Journal of Emerging Research in Engineering and Technology*, *2*(3), 71-78. https://doi.org/10.63282/3050-922X.IJERET-V2I3P108