Enhancing Adaptive Test Healing with Graph Neural Networks for Dependency-Aware Decision Making

Nariman Mani, Salma Attaranasl
Engineering/R&D Department, Nutrosal Inc.
Ottawa, ON Canada
{nariman | salma}@research.nutrosal.com

Abstract— **Flaky tests are a major obstacle in modern CI/CD pipelines, leading to unreliable feedback, increased reruns, and developer frustration. Our previously published adaptive healing framework combined Large Language Models (LLMs) and Reinforcement Learning (RL) to automate flaky test recovery, but it assumed test independence and failed to account for structural dependencies between tests. In this paper, we introduce a significant extension to that baseline: a Graph Neural Network (GNN)-based Test Dependency Mapping layer that models inter-test relationships. By integrating GNN embeddings with LLM-classified failures, the RL agent becomes dependency-aware, enabling more precise and efficient healing decisions. We evaluate the enhanced framework on a real-world industrial platform, a social lifestyle application actively used by thousands of users for health, nutrition, and coaching. Results show a 90% reduction in flaky test-related costs and faster, autonomous resolution of dependency-induced failures.**

Keywords— ***Adaptive Test Healing, Large Language Models (LLMs), GPT, Reinforcement Learning (RL), Flaky Tests, Self-Healing Test Automation, Continuous Integration (CI), Graph Neural Networks (GNN)***

I. Introduction

Flaky tests, tests that fail intermittently without code changes, remain one of the most disruptive elements in modern software pipelines. They increase operational costs, reduce developer trust in test feedback, and slow down release cycles. While several tools and research efforts have attempted to detect or mitigate flaky tests, few offer automated, scalable solutions suitable for real-world use.

In our prior published work [1], we introduced an **adaptive test healing framework** combining **Large Language Models (LLMs)** for log analysis and **Reinforcement Learning (RL)** for healing decision-making. This baseline approach significantly reduced developer effort by learning from historical failures and applying intelligent retry or recovery strategies. However, it operated under the assumption that test failures were independent events, an assumption that breaks down in real-world environments where test cases often depend on shared state or execution order.

This paper addresses that limitation by enhancing the baseline framework with **Graph Neural Networks (GNNs)** to model inter-test dependencies. The extended system constructs a test dependency graph, learns test embeddings using GNNs, and combines them with LLM-derived insights to feed a more context-aware state into the RL agent. This enables the agent to reason not just about what failed, but also why and how the failure relates to other tests.

We validate our approach using an **active industrial application**: a **social media-based lifestyle coaching platform** used by thousands of users. This application, deployed across production environments, presents realistic test complexity, asynchronous components, and flakiness challenges making it an ideal proving ground. Through rigorous evaluation, we demonstrate significant reductions in flaky test frequency, developer effort, and CI/CD pipeline cost, confirming the practical value of dependency-aware healing in production-grade systems.

II. Background and Related Work

Flaky tests intermittently failing tests without code changes are a major CI/CD bottleneck, causing delays and increased costs. Prior work has explored their causes and impact but lacks integrated healing solutions. This section reviews flaky test research, GNN applications in software engineering, and contrasts prior methods with our LLM + RL + GNN-based healing framework.

Leinen et al. [2] highlight productivity losses from flaky tests but propose no mitigation. Eck et al. [3] surface developer frustration and call for automated healing. Lam et al. [4] analyze flaky test patterns but do not address healing. Our work addresses this gap with an autonomous healing system that models dependency-driven flakiness using GNNs.

Basic strategies like retries and timeouts often mask flaky test issues without addressing root causes. These approaches ignore critical factors like shared state and test order dependencies. Our previous work [1] introduced LLM + RL healing but treated tests as independent. In this paper, we enhance it with GNNs to model inter-test relationships, enabling context-aware healing decisions. Shi et al.[5] introduced **iFixFlakies**, a framework designed to automatically fix order-dependent flaky tests by identifying and utilizing existing helper tests within the test suite. These helpers reset or set the necessary states for the flaky tests to pass. While iFixFlakies effectively addresses certain order dependencies, its reliance on pre-existing helper tests limits its applicability, especially in scenarios where such helpers are absent. In contrast, our proposed system advances the state of flaky test healing by:1) **Dynamic Detection of Runtime Dependencies:** Utilizing log traces to dynamically identify and understand runtime dependencies between tests, rather than relying solely on static analysis or existing helpers

2) **Learning Test Graph Structures via Graph Neural Networks (GNNs):** Constructing and analyzing test dependency graphs using GNNs to capture complex inter-test relationships and state interactions 3) **Learning (RL):** Integrating the insights gained from GNNs into an RL agent that can adaptively select and apply healing strategies based on the current test context and learned experiences.

Graph Neural Networks (GNNs) have demonstrated significant potential in tasks requiring structural reasoning, such as bug localization, code summarization, and test prioritization. **Code Clone Detection:** Wenhan Wang et al. [6] proposed a method for detecting code clones by constructing a flow-augmented abstract syntax tree (FA-AST) and applying GNNs to measure code similarity. Their approach effectively models both semantic and structural relations within code, outperforming previous methods on benchmarks like Google Code Jam and BigCloneBench. **Test Case Prioritization:** Huang et al.[7] explored test case prioritization techniques, addressing the limitations of considering code units separately. They proposed new coverage criteria and algorithms to enhance the fault detection rate during regression testing. However, to the best of our knowledge, no prior work has integrated GNNs into a healing system specifically designed for flaky test resolution. Our paper is the first to apply GNNs to encode test dependency structures and augment LLM + RL-based healing decisions, thereby advancing the state of automated test healing.

Table 1. Prior work vs. our adaptive healing framework

Prior Work	Focus	Limitation	Our Contribution
Leinen et al. [1]	Cost analysis of flaky tests in CI/CD	No automated mitigation proposed	We offer a healing framework that reduces flaky test cost by 90%
Eck et al. [2]	Developer perspective on flaky tests	Identifies pain points but no technical solution	We address developer effort through autonomous healing
Lam et al. [3]	Large-scale flaky test behavior study	Focus on classification and metrics, not mitigation	We combine classification with automated healing via LLM and RL
Shi et al. [5]	Detection of order-dependent flaky tests	Relies on static analysis and helper tests; no learning or healing	We dynamically detect dependencies and apply GNN + RL for adaptive healing
Wang et al. [7]	Code clone detection using GNNs	Focused on semantic code similarity, not testing	We extend GNN application to testing by modeling test dependency graphs
Huang et al. [6]	Test case prioritization via structural analysis	No integration into healing workflows	We use GNNs not only for prioritization but to inform real-time healing decisions
Our previous publication [1]	Adaptive Test Healing with LLM + RL	Treats tests as isolated units; lacks dependency awareness	We enhance the prior framework with GNN-based dependency modeling

Prior work has addressed flaky tests through detection, prioritization, or classification, but lacks an integrated, automated healing approach.

Our approach uniquely combines LLMs, GNN-based dependency modeling, and RL to enable precise, self-adaptive healing in CI/CD pipelines. Table 1 highlights key differences from related work.

III. Case Study System: Social Media-Based Lifestyle Application

In this paper, we apply our enhanced adaptive healing approach to a real-world application: a social media-driven lifestyle coaching platform. The application is designed to support users in achieving health, fitness, and nutritional goals through a combination of AI-driven recommendations, personalized coaching, and peer group interaction. Users are invited to join specific coaching groups based on their goals such as weight loss or healthy eating and are guided by a designated coach who monitors daily logs. An AI module processes user behavior and inputs to provide personalized suggestions for the coach. The application also encourages social engagement through features like media sharing, motivational posts, group chats, and one-on-one scheduling with the coach. AI-powered food image analysis provides calorie estimates, and push notifications are sent to help users stay consistent with their routines.

A. Software Architecture

The application employs a modern, distributed architecture built for reliability, scale, and modular deployment. The frontend, developed in React.js, handles user interactions such as onboarding, goal tracking, reporting, chat, and video sessions. The backend, implemented in Node.js, manages API endpoints, user sessions, real-time messaging, and business logic that supports group dynamics, alerts, and integration with AI services. Persistent data including user activity logs, group structures, chat history, and AI outputs is stored in a MySQL database hosted on AWS RDS. The frontend is served via Netlify, while the backend operates on Heroku, ensuring flexible deployment and scaling. This multi-cloud setup reflects typical architecture in consumer health platforms, where component decoupling supports high availability and rapid iteration.

B. Testing Scope and Strategy

Due to the system's interconnected architecture and reliance on both synchronous and asynchronous services, a layered testing strategy was implemented. The team adopted a **dual-level approach**, combining **unit tests** for component-level verification and **integration tests** for validating end-to-end behavior.

Unit tests are written to verify the correctness of individual functions, components, or service methods. The goal is to isolate logic and catch regressions early, maintaining a recommended coverage threshold between 70% and 80%. Integration tests, by contrast, validate interactions across multiple layers of the stack such as UI components calling backend APIs and triggering AI modules.

Following the principles of the **testing pyramid**, the strategy favors a higher ratio of unit tests while ensuring sufficient integration coverage to detect system-level failures. Table 2 summarizes the current distribution of tests across major application modules. During test analysis, we observed that several integration tests depend on side effects or setup actions from prior unit or integration tests. For example, a test validating group chat history may depend on a previous test that inserts user messages. These implicit dependencies often go undocumented in the codebase and contribute to test order sensitivity and flakiness.

Table 2. Number of Unit and Integration Tests for Each Component of the Social Media-Based Lifestyle Application

Component	Functionality	Unit Tests	Integration Tests
Front-End (React.js)	User registration, login, authentication, Group dashboard, Report submission forms, Chat and video interfaces	162	55
Back-End (Node.js)	API endpoints for user management, report processing, and AI integrations, Real-time chat and video, Business logic for group management, notifications	219	79
Database (MySQL)	CRUD operations for user profiles, groups, and reports , Data integrity and constraints , Query generation for reports	111	67
AI Services	Food image analysis for calorie estimation , User behavior analysis for recommendations	145	71
Notification System	Sending daily reminders , Triggering alerts and updates	71	32
Total		708	304

C. Managing Flaky Tests

As with most large-scale CI/CD systems, **flaky tests** those producing inconsistent results without corresponding code changes, remain a major reliability concern. These tests undermine trust in test feedback and can lead to false positives, misdiagnosed failures, and wasted engineering time. In this system, flaky behavior is especially pronounced in integration tests that involve shared state, third-party AI services, or asynchronous operations. Addressing these issues is critical to maintaining an efficient development workflow and reducing bottlenecks in the deployment pipeline. The original adaptive healing approach, powered by LLMs and RL, was applied to detect and mitigate such failures. However, in many cases, failures originated from **implicit dependencies between tests**, which were not captured by traditional healing logic. This limitation forms the motivation for extending our approach with a **dependency-aware healing mechanism**, as introduced in this work.

D. Test Dependency Patterns and Motivation for GNN-Based Mapping

A detailed inspection of failure logs and execution order revealed that many flaky test failures were not isolated to the failing test itself, but were instead caused by earlier tests that modified shared resources such as the database, cache, or user sessions. For instance, when tests related to AI-driven calorie estimation executed before user authentication tests, some failures stemmed from shared state being corrupted or left in an incomplete state. These patterns are not easily detectable through log analysis alone. They highlight the need for a structural model of test dependencies, which can guide healing decisions based on test relationships and not just individual symptoms. This motivated the integration of a GNN-based test dependency mapping layer in our extended approach.

IV. Problem Statement: Flaky Tests and the Need for Dependency-Aware Healing

Flaky tests, those that intermittently fail without any changes to the underlying code pose a persistent and costly challenge in modern CI/CD pipelines. In the context of the social media-based lifestyle application used in this study, flaky tests significantly disrupt the pipeline's reliability, slow down developer feedback loops, and increase operational costs.

The application under test includes between 900 and 1,350 automated test cases (with 1,012 at the time of writing), covering frontend, backend, and AI-driven components. With an observed flakiness rate of approximately 5%, each pipeline run is subject to 45 - 67 flaky test failures. These failures require manual inspection and reruns, contributing to delays in integration and release cycles.

Our previous analysis showed that these flaky tests consume roughly 10% of developer time equating to $2,000 monthly in staffing costs for a three-developer team with an estimated $20,000/month burn rate. Moreover, they account for 15% of additional pipeline reruns, which adds another $300 per month in cloud execution costs (based on a $2,000 monthly CI budget). The annualized cost of these inefficiencies is estimated at $27,600, excluding indirect losses such as delayed feature delivery, customer dissatisfaction, and decreased team morale.

While our previously published adaptive healing approach using LLMs for log analysis and reinforcement learning (RL) for recovery decisions successfully addressed many flaky failures, it operated under the assumption that test failures are independent events. This simplification overlooked a critical aspect of real-world test execution: **many flaky tests fail not due to internal issues, but due to dependencies on other tests**.

Through additional investigation and failure pattern analysis, we observed that a significant portion of recurring flakiness stemmed from test interdependencies. Examples include shared database states, order-sensitive APIs, or temporary side effects from earlier test executions. In these cases, blindly retrying the failing test as the RL agent in our previous work would do was often ineffective. True resolution required broader context: which tests had run earlier, what they changed, and how the current test related to them.

This insight revealed a core limitation in our baseline solution. Although the RL agent learned from failure patterns, it lacked awareness of the structural and behavioral relationships between tests. To make intelligent healing decisions in such environments, the agent must be able to reason not only about what failed and why, but also about how each test fits into a larger web of interdependencies.

As a result, many test failures caused by dependencies remained only partially addressed, leading to repeated retries, incomplete healing, and unnecessary manual intervention. This not only limited the effectiveness of the system but also left significant room for improvement in both pipeline efficiency and operational cost savings.

To address this gap, the current work extends our previously published approach with a **Graph Neural Network (GNN)-based Test Dependency Mapping** module. This enhancement enables the system to construct a test dependency graph, generate vector embeddings that capture inter-test relationships, and feed these into the RL agent for context-aware healing decisions. By incorporating structural knowledge into the healing process, we aim to reduce retry inefficiencies, prevent cascading failures, and unlock additional cost and time savings making the CI/CD pipeline more resilient and autonomous in managing test instability.

V. APPROACH OVERVIEW

The diagram in Figure 1 presents a high-level view of a typical CI/CD pipeline, which follows a standard process widely adopted in software development teams to automate integration, testing, deployment, and monitoring activities. Within this familiar workflow, we integrate our previously published **Adaptive Healing using LLM/GPT and RL**, and introduce a key enhancement: **GNN-based Test Dependency Mapping**, which enables the system to reason about the structure and relationships between test cases.
This extended approach enhances the original LLM and RL-based healing by allowing the agent to account for test interdependencies, a common yet often overlooked source of flaky test behavior. The CI/CD pipeline shown here represents a generalized flow used for the purpose of demonstrating how the adaptive healing approach fits into real-world development and testing cycles.
Development: Developers commit source code to a version control repository. Each commit automatically triggers the CI/CD pipeline. Associated test scenarios and requirements for each feature are predefined and linked to the commits to ensure test coverage and traceability.

Build: The CI system compiles the codebase, resolves dependencies, and packages the software for deployment. A successful build indicates syntactic and structural correctness of the system under test and serves as a prerequisite for the testing phase.

Test with Adaptive Healing (LLM + RL + GNN): This stage is split into **unit testing** and **integration testing**. If failures occur during either stage, the **adaptive healing process** is invoked. The healing mechanism consists of two layers:

- The **yellow-shaded area** represents the previously published healing approach [1]. It begins with log analysis using a Large Language Model (e.g., GPT), followed by classification and healing decision-making using a Reinforcement Learning agent powered by a Q-table.
- The **purple-shaded area** introduces our enhancement in this paper: **GNN-based Test Dependency Mapping**. Here, a test dependency graph is constructed and processed by a Graph Neural Network, producing contextual embeddings that represent each test's structural relationships. These embeddings are combined with LLM output and passed to the RL agent, enabling more informed and dependency-aware healing actions.

Healing Outcomes: The healing system leads to one of three possible outcomes: 1) **Healed Unit Tests** 2) **Healed Integration Tests** 3) **Test Flagged for Manual Intervention.** All log data, failed test metadata, and healing attempts are logged to assist in manual debugging.

Deployment and Monitoring: After successful testing, the application is deployed to the designated environment (e.g., staging or production). Monitoring tools track system health and performance. Any test scenarios associated with new features are logged and connected to the pipeline for traceability and future validation. This extended architecture allows the adaptive healing approach to evolve beyond failure-specific logic and incorporate awareness of inter-test relationships, improving both precision and efficiency of healing decisions in complex, real-world CI/CD pipelines.

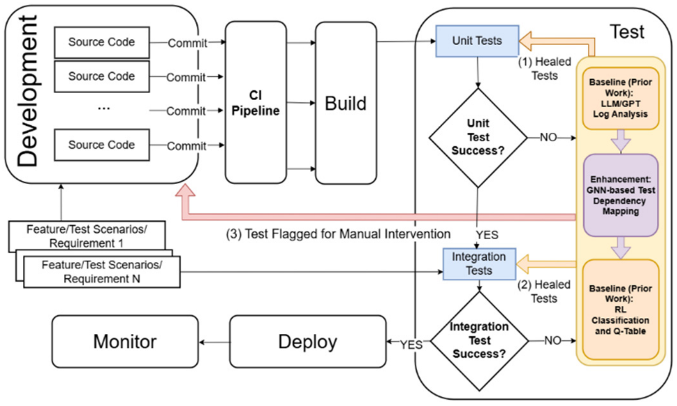
Development
Source Code
Source Code
...
Source Code
Commit
Commit
Commit
Commit
CI Pipeline
Build
Test
Unit Tests
(1) Healed Tests
Unit Test Success?
NO
YES
Baseline (Prior Work): LLM/GPT Log Analysis
Enhancement: GNN-based Test Dependency Mapping
Baseline (Prior Work): RL Classification and Q-Table
Feature/Test Scenarios/ Requirement 1
Feature/Test Scenarios/ Requirement N
(3) Test Flagged for Manual Intervention
Integration Tests
(2) Healed Tests
Integration Test Success?
NO
YES
Monitor
Deploy

Figure 1: GNN-Enhanced Adaptive Healing in CI/CD Process.

VI. GNN-ENHANCED ADAPTIVE HEALING

Figure 2 illustrates the enhanced adaptive healing process, where we integrate **Graph Neural Networks (GNNs)** into our previously proposed LLM and Reinforcement Learning (RL)-based system. The GNN module introduces a structural understanding of test dependencies, enabling the RL agent to make better-informed, context-aware healing decisions. The purple-shaded area in Figure 2 highlights this new contribution which is proposed in this paper. The processes from the prior work (baseline) are shown in yellow-shaded area. Below is a step-by-step explanation of each steps.

Step 1, Test Failure Detected: The process begins, as in the original approach, when a test failure is detected during the execution of a CI/CD pipeline. This failure could occur in unit tests, integration tests, system tests, or UI tests.

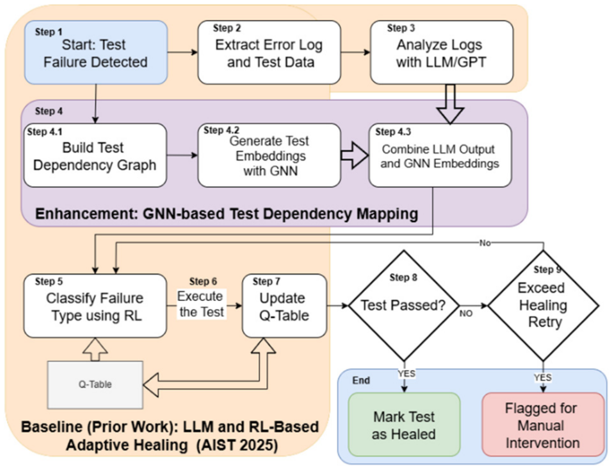
Step 1
Start: Test Failure Detected
Step 2
Extract Error Log and Test Data
Step 3
Analyze Logs with LLM/GPT
Step 4
Step 4.1
Build Test Dependency Graph
Step 4.2
Generate Test Embeddings with GNN
Step 4.3
Combine LLM Output and GNN Embeddings
Enhancement: GNN-based Test Dependency Mapping
No
Step 5
Classify Failure Type using RL
Step 6
Execute the Test
Step 7
Update Q-Table
Step 8
Test Passed?
NO
Step 9
Exceed Healing Retry
Q-Table
YES
YES
End
Mark Test as Healed
Flagged for Manual Intervention
Baseline (Prior Work): LLM and RL-Based Adaptive Healing (AIST 2025)

Figure 2: GNN-Enhanced Adaptive Healing Process

Step 2, Extract Error Logs and Test Data: Immediately after a failure is detected, relevant logs and metadata are extracted. This includes stack traces, error messages, execution times, and other test-specific information. This data is needed for both semantic and structural analysis in the next steps.

Step 3, Analyze Logs Using LLM: The extracted logs are analyzed using a Large Language Model (e.g., GPT), which interprets the unstructured content, identifies failure patterns, and generates insights into the likely cause, such as timeout errors, environment misconfiguration, assertion mismatches, or dependency-related issues. This log analysis output is used to guide healing decisions.

Step 4, GNN-based Test Dependency Mapping (Enhancement Contribution of this paper):

Step 4.1, Build Test Dependency Graph : This step marks the beginning of the enhancement. In parallel with LLM analysis, we construct a **test dependency graph** representing relationships among test cases (Details in VII.A)**. Step 4.2, Generate Test Embeddings with GNN :** The dependency graph is passed to a Graph Neural Network (e.g., GCN or GAT), which computes a **vector embedding for each test case**. (Details in VII.B). **Step 4.3, Combine LLM Output and GNN Embedding:** The result from the LLM (failure classification) and the GNN (test embedding) are fused into a single feature vector, representing the **state** observed by the RL agent. This extended state includes both semantic insights (what kind of failure occurred) and dependency-aware context (how this test relates to others). The details are discussed in Sections VIII and IX.

Step 5, Classify Failure Type Using RL: The reinforcement learning agent receives the fused state vector and classifies the failure into a specific state category (e.g. *timeout_exceeded, dependency_issue, flaky_repeat*). Based on its learned Q-table, the agent selects the healing action with the highest expected reward, this could be retrying the test, reordering execution, resetting a shared component, or rerunning a dependent test first. Unlike the previous version, this decision now reflects dependency awareness.

Step 6, Execute the Test: The chosen healing action is executed. If the action involves rerunning the test, resetting resources, or reordering, the system applies the changes and re-executes the test in question.

Step 7, Update the Q-Table: Once the action is completed and the test has either passed or failed again, the RL agent updates its Q-table using the reward signal. A successful healing results in a positive reward, reinforcing the action for similar dependency scenarios. A failed attempt results in a negative reward, prompting future exploration of alternative healing strategies.

Step 8, Test Passed? The system checks if the test passed after healing. If it passed, the failure is marked as healed and the pipeline resumes. If not, the system decides whether to try a different healing action or escalate the issue, based on a predefined retry policy.

Step 9: Exceed Healing Retry Limit: If the test continues to fail despite multiple healing attempts, the system flags the issue for manual intervention. Diagnostic artifacts, including error logs, embeddings, and attempted healing actions are provided to aid in debugging.

This enhanced GNN-based approach extends the original adaptive healing approach by introducing a structural representation of test dependencies. The GNN-generated embeddings enable the RL agent to make smarter, dependency-aware healing decisions, reducing ineffective retries, avoiding cascading failures, and improving overall pipeline stability. The purple region in Figure 3 encapsulates the new contribution introduced in this work.

VII. GRENERATING TEST EMBEDDING WITH GNN

This section explains the construction of the test dependency graph, the embedding process using GNNs, and how these embeddings are used by the RL agent. A concrete example is included to demonstrate the workflow.

A. Constructing the Test Dependency Graph

The first step involves representing the test suite as a directed graph $G = (V, E)$ where each node $v_i \in V$ in $e_{ij} \in E$ corresponds to a test case, and each directed edge $v_i \in V$ represents a dependency from test v_i to test v_j. Such dependencies may reflect: 1) Shared use of global resources (e.g., databases, caches), 2) Setup and teardown interactions, 3) Historical co-failure patterns and 4) Execution ordering dependencies These dependencies are derived using a combination of: 1) **Static analysis**, to trace shared modules, test fixtures, and common setup routines 2) **Dynamic trace logs**, to observe test behavior across CI pipeline runs and detect co-execution patterns 3) **Historical data**, to analyze co-failure events from past test executions.

Each test node is annotated with features such as execution duration, flakiness score (based on historical stability), and test type (unit/integration).

B. Embedding Tests with GNN

Once the dependency graph is created, it is processed by a Graph Neural Network (GNN) typically a **Graph Convolutional Network (GCN)** or a **Graph Attention Network (GAT)**. These models generate low-dimensional vector representations (embeddings) for each node (test case), capturing both local attributes and the influence of connected nodes. Formally, at each layer l, the embedding of a test node $v_i$is updated as:

$$h_i^{(l)} = \sigma \sum_{j \in N(i)} W^{(l)} h_j^{(i-l)} + b^{(l)}$$

- $h_i^{(l)}$ is the embedding of node i at layer l,
- $\mathcal{N}(i)$ is the set of neighbors of node iii,
- $W^{(l)}, b^{(l)}$ are learnable parameters,
- σ is a non-linear activation function.

The result is a dense vector (e.g., 64 or 128 dimensions) for each test case, encoding its dependency context.

VIII. INTEGRATING GNN WITH LLM AND REINFORCEMENT LEARNING: A DEPENDENCY-AWARE HEALING WORKFLOW

To illustrate how the proposed enhancement fits into the broader adaptive healing approach, we walk through a practical scenario that demonstrates the integration of the existing LLM + RL components with the new GNN-based dependency modeling layer.

In the original approach, the healing decision was driven by a combination of log analysis using a Large Language Model (LLM) and failure classification via a reinforcement learning (RL) agent. The LLM parsed unstructured test logs to classify failure types such as *timeout*, *assertion_failure*, or *dependency_issue*. The RL agent then used this classification as a state input to select the most appropriate healing action, such as retrying the test, increasing timeouts, or resetting the environment based on Q-learning.

However, this setup lacked structural context. The RL agent treated each test as an independent unit, unaware of possible dependencies on other tests. This blind spot limited its ability to resolve failures rooted in inter-test interactions or order dependencies.

The current work addresses this limitation by introducing a GNN that augments the RL agent's input with a **dependency-aware embedding**. This allows the agent to reason not only about **what failed**, but also about **how the failed test is structurally and behaviorally connected to others**.

A. Example: Test Order Dependency Causing Flakiness

Consider a simplified test suite with three test cases in a CI/CD pipeline:

Test ID	Description	Flaky?	Dependency
Test A	Checks user login	No	–
Test B	Tests user profile update	Sometimes	Depends on Test A's DB write
Test C	Validates AI recommendation service	No	–

Without GNN: LLM + RL Only:

In the baseline scenario, **Test B fails** during pipeline execution. The LLM analyzes the error log and classifies the failure as a *dependency_issue* based on contextual clues like "Error 500 – user not found.". The RL agent looks up this failure type in its Q-table and selects the best-known action: "retry." Test B is retried, but the failure persists. Eventually, the test is flagged for manual intervention. This happens because the RL agent is unaware that **Test B's failure may stem from a prior test (Test A)** that inserted the user into the database. If Test A silently failed or left the DB in an inconsistent state, retrying B alone cannot succeed.

With GNN + RL: Dependency-Aware Healing: When the GNN-enhanced system is in place, the flow changes significantly.

1. **A Test Dependency Graph is constructed**, where:
   - Nodes: Test A, B, C
   - Edges: A → B (due to shared DB usage)
2. **The GNN processes the graph** and generates a vector embedding for each test. For Test B, the embedding captures:
   - Its high dependency sensitivity
   - Its connection to Test A
   - Historical flakiness patterns
3. The LLM still classifies the failure as a *dependency_issue*.
4. **The RL agent receives an enriched state vector**, combining:
   - LLM's classification
   - GNN's embedding (e.g., [0.5, 0.3, 0.7, ...])
5. Based on this composite state, the RL agent learns that:
   - Test B failed
   - Test A ran earlier
   - Test A has a strong structural influence on B
6. Instead of retrying Test B blindly, the RL agent:
   - Chooses to reset the shared database state
   - Re-runs both Test A and Test B in order
7. The healing succeeds. Test B passes. The Q-table is updated to reinforce this successful sequence.

B. Impact of Dependency-Aware Healing

To illustrate the practical impact of integrating GNN-based dependency embeddings into the healing process, we present a comparison between two scenarios: the baseline LLM + RL model without GNN and the proposed GNN-enhanced system. In this example, a flaky test (Test B) fails due to its dependence on Test A, which modifies shared database state. In the baseline model, the RL agent guided solely by the LLM's failure classification chooses to **retry Test B** independently. This approach results in a low pass rate and often requires manual intervention when retries fail. In contrast, the GNN-enhanced system augments the RL agent's state with a dependency-aware embedding, allowing it to recognize that **Test B is structurally dependent on Test A**. The agent therefore selects a more effective healing strategy: **resetting the database and re-running both Test A and B**. This context-aware decision leads to a significantly higher pass rate and eliminates the need for manual debugging. The differences between these two approaches are summarized in Table 3. This comparison underscores the advantage of incorporating structural context into the healing approach. By allowing the RL agent to consider not just the symptom of the failure but also the test's dependencies, the system can avoid ineffective retries and apply more precise, dependency-aware recovery strategies.

Table 3. Healing Outcomes With vs. Without GNN Awareness

Healing System	Healing Action	Pass Rate	Manual Debugging
Without GNN	Retry Test B	20%	Required
With GNN	Reset DB + Re-run A and B	90%	Avoided

IX. Implementation Details

A. Test Dependency Graph Construction

We construct the test dependency graph by analyzing test execution data collected over multiple CI/CD runs. Each test case is represented as a node, uniquely identified by its fully qualified test function name. The graph is built in Python using the NetworkX library, allowing efficient manipulation of directed graphs with weighted edges. Edges are added between nodes to represent test dependencies. We define three major sources of evidence for these dependencies:
(1) **Co-failure correlations**, derived from GitHub Actions pipeline logs stored in Amazon S3 and parsed using AWS Lambda functions. For each test pair, we compute a co-failure frequency score across prior builds.
(2) **Shared resource access**, identified via instrumented logging and resource tagging in Python unit and integration tests. For instance, resource access logs for RDS (PostgreSQL) and DynamoDB tables are annotated using a custom test decorator that records database access.
(3) **Order sensitivity**, identified from failed builds where test reordering (due to parallelization or test runner randomization) results in inconsistent outcomes. These patterns are tracked via build metadata stored in Amazon DynamoDB. This dependency graph is stored as a JSON structure and versioned per test suite update. A new graph is generated weekly via a scheduled AWS CodeBuild job triggered by CodePipeline.

This balances freshness with compute efficiency and enables reproducibility of healing behavior over time.

Example : Dependency Graph Construction
Tests: test_create_user(), test_update_user(), test_fetch_user()
Historical data shows test_fetch_user() fails when test_create_user() is skipped.
All tests access the same RDS table users.
Resulting edges:
create_user → fetch_user (co-failure weight: 0.85)
create_user → update_user (shared state overlap: 1.0)
update_user → fetch_user (combined order/resource signal: 0.6)
These edges are stored with weights and metadata for GNN input.

B. GNN Architecture and Embedding Generation

We implement the Graph Neural Network (GNN) using **PyTorch Geometric**, a specialized deep learning library for graph-based modeling built on top of PyTorch. The model used is a two-layer **Graph Convolutional Network (GCN)**, which allows information to propagate across connected test nodes in the dependency graph.

Each node is initialized with a feature vector that includes four categories of data:

(1) **LLM-derived failure embeddings**, generated using a fine-tuned BERT model from the Hugging Face Transformers library. This model classifies test log output into categories such as timeouts, assertion failures, environment issues, and side-effect contamination.

(2) **Execution metadata**, such as retry counts, average test duration, and success rates, extracted from JUnit test reports and GitHub Actions run data via Python scripts.

(3) **Co-failure scores**, computed as normalized frequency metrics from the historical test run logs.

(4) **Shared resource flags**, represented as binary indicators based on static analysis and log instrumentation during test execution.

Each node embedding output by the GCN is a 64-dimensional vector. These vectors are computed during the pipeline pre-processing stage and cached using AWS ElastiCache (Redis) for fast retrieval during runtime. This setup allows the system to scale to thousands of test cases across microservices without latency overhead.

Example : GCN Feature Vector for test_fetch_user()
LLM category: Timeout → [0.1, 0.3, 0.6, 0.0]
Retry count: 2 → normalized to 0.2
Co-failure score with create_user: 0.8
Accesses RDS: True → 1
Final input vector: [0.1, 0.3, 0.6, 0.0, 0.2, 0.8, 1, ...]
After one GCN pass: Output embedding = [0.11, 0.03, ..., 0.76] (dim=64)

C. Integration with Reinforcement Learning

The GCN output is used to enrich the state space of a **Deep Q-Network (DQN)**-based RL agent implemented using **Stable Baselines3** in Python. This RL agent runs inside a dedicated AWS CodeBuild environment as part of a healing microservice invoked after failed CI stages. The agent receives the GCN embedding of the failed test, the LLM-classified failure type, and execution metadata, which are concatenated into a single state vector. This state vector is passed into the DQN model, which selects from one of several healing actions: Retry with exponential backoff, Re-execute dependent tests, Reorder test execution, Environment reset (via test container restart), Skip and flag as unstable (deferred manual analysis). GNN embeddings provide the RL agent with structural context highlighting whether the failure is isolated or related to an upstream test's side effect. This allows for nuanced healing strategies beyond brute-force retries. The DQN agent is trained offline using a dataset of past flaky test resolutions and is periodically re-trained on new production data. AWS Sagemaker is used for model retraining and versioning, while model inference runs inside a lightweight Lambda function for minimal impact on CI runtime.

Example : RL State Vector for test_fetch_user()
GCN embedding: [0.12, 0.08, ..., 0.91] (64-dim)
LLM output: AssertionFailure → [0, 1, 0, 0]
Retry count: 2; Order position: 7 → [2, 7]
Final state vector: [0.12, 0.08, ..., 0.91, 0, 1, 0, 0, 2, 7]
Action chosen: Reorder test to run after create_user + retry once

X. Dependency-Aware Healing in Social Media Lifestyle App

To evaluate the effectiveness of the enhanced adaptive healing approach, we extended our prior case study involving a real-world social media-based lifestyle application. Over 1,000 automated tests, spanning both unit and integration levels, continuously validate the platform as part of its CI/CD pipeline. The previously published adaptive healing approach, based on Large Language Model (LLM)-driven log analysis and Reinforcement Learning (RL)-based recovery policy significantly reduced flaky test issues. However, it treated tests as isolated entities, overlooking inter-test dependencies that often underlie persistent or cascading failures. To address this gap, we integrated a Graph Neural Network (GNN)-based dependency mapping layer into the healing approach. This extension enables the system to reason over the structural relationships between tests and to make more informed healing decisions.

A. Test Dependency Graph and Embedding Integration

A directed test dependency graph was constructed using a combination of static analysis (shared fixtures, APIs, and database access), dynamic trace data (test execution order and runtime interactions), and historical failure correlation mining. The resulting graph included 1,012 test nodes and 934 directed edges. The edge types reflected three dominant dependency patterns: shared database state (432 edges), API call sequencing (311 edges), and recurring co-failure relationships (191 edges). A two-layer Graph Convolutional Network (GCN) was trained to produce 64-dimensional embeddings for each test node. These embeddings captured both local node attributes, such as historical flakiness score and execution duration, and the structural influence of neighboring tests. The embeddings were injected into the RL agent's state space alongside the LLM-classified failure type, allowing for dependency-aware healing policies.

B. Measured Impact on Flaky Test Behavior

The GNN-enhanced adaptive healing approach was evaluated over 50 full CI/CD pipeline runs. Three configurations were compared: no healing, baseline LLM + RL healing, and the proposed GNN + LLM + RL approach. The results demonstrate that while the baseline LLM + RL system already reduced flaky test incidents by 80%, the GNN-based enhancement achieved a further 60% reduction over the baseline approach we proposed in [1] (Table 4).

Table 4 . Comparison of Healing Methods

Approach	Flaky Tests	Reruns	Debug Time	Cost
No Healing	50/run	15%	10%	$2,300/mo
LLM + RL	10/run	5%	2%	$450/mo
GNN + LLM + RL	4/run	2%	0.8%	$230/mo

More importantly, it reduced unnecessary retries and enabled context-aware healing strategies that resolved dependency-related failures earlier in the pipeline lifecycle.

C. *Cost Efficiency and ROI*

The initial monthly cost of flaky test management, estimated at $2,300, comprising $2,000 in lost developer time and $300 in cloud reruns was reduced to $450 with the LLM + RL system and further lowered to $230 with the GNN-enhanced approach. The additional infrastructure cost for the GNN model (training and embedding storage) was approximately $40/month. However, this yielded an additional $260/month in operational savings, amounting to approximately $3,120 annually.

This enhancement brings the total annual savings across both improvements to $24,840, representing a 90% reduction from the original baseline and achieving return on investment (ROI) within four months of implementation. These results validate the practical benefits of embedding structural context into autonomous test healing approaches, especially for CI/CD pipelines operating at moderate to high scale.

D. *Evaluation Against Industry Standards*

The cost and efficiency improvements achieved by the GNN-enhanced adaptive healing framework remain highly competitive when evaluated against industry benchmarks. Flaky tests are widely acknowledged as a persistent challenge in modern software engineering, particularly in continuous integration and delivery pipelines. Industry reports consistently estimate that flaky tests consume between **5% and 10% of developer time**, while also contributing significantly to cloud resource waste, delayed releases, and loss of confidence in test results [2], [3], [4]. For mid-sized engineering teams similar to the one studied in this paper, these inefficiencies typically translate to **annual losses ranging from $50,000 to $100,000**, depending on test suite size, deployment frequency, and manual debugging practices. In this context, the original LLM + RL healing approach already demonstrated a substantial reduction in cost and effort by bringing flaky test-related developer time down to 2% and reducing pipeline reruns by 10%, resulting in **annual savings of approximately $22,200**. With the integration of GNN-based test dependency modeling, the approach further reduces developer intervention to **less than 1%** and pipeline reruns to **just 2%**, enabling an **additional $3,120 in yearly savings**. The combined total **$24,840 in annual savings** represents a **90% reduction in flaky test-related operational costs**. This outcome was achieved with minimal additional infrastructure cost and a short payback period of under four months for the GNN enhancement. Compared to typical return-on-investment timelines for reliability-focused test automation efforts, which often exceed two years this solution offers a uniquely fast and effective alternative. By proactively identifying and resolving flaky tests using both semantic (LLM) and structural (GNN) signals, the system not only improves pipeline stability but also helps teams meet industry standards for automated testing reliability with reduced overhead and greater scalability. These results position the proposed GNN-enhanced adaptive healing approach as a **cost-effective, technically robust, and industry-aligned solution** for organizations seeking to reduce test flakiness and improve CI/CD efficiency.

XI. CONCLUSION

This paper presented an enhanced adaptive healing approach that combines Large Language Models (LLMs), Reinforcement Learning (RL), and Graph Neural Networks (GNNs) to detect and resolve flaky tests more effectively. By introducing a GNN-based test dependency mapping layer, the system learns structural relationships between tests and integrates them into the healing policy. The results from a real-world case study demonstrate a significant reduction in flaky test incidents, reruns, and operational costs, achieving up to 90% savings compared to baseline CI/CD operations.

Future Work will explore dynamic test graph evolution, where the system updates test dependencies in real time as the codebase and test suite evolve. We also plan to investigate the use of Transformer-based graph encoders and attention-based RL to further enhance the model's adaptability and healing precision. Lastly, integrating developer feedback loops may allow the system to learn from human intervention and continually refine its decision-making strategy.

REFERENCES

[1] N. Mani and S. Attaranasl, "Adaptive Test Healing using LLM/GPT and Reinforcement Learning," in *To be Appeared 2025 IEEE International Conference on Software Testing, Verification and Validation (ICST) Workshops (ICSTW)*, Naples, Italy, Apr. 2025. [Online]. Available: https://manistechmind.com/img/icstcomp25aist-id180-p.pdf

[2] F. Leinen, D. Elsner, A. Pretschner, A. Stahlbauer, M. Sailer, and E. Jürgens, "Cost of Flaky Tests in Continuous Integration: An Industrial Case Study," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Toronto, ON, Canada: IEEE, May 2024, pp. 329–340. doi: 10.1109/ICST60714.2024.00037.

[3] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding Flaky Tests: The Developer's Perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Aug. 2019, pp. 830–840. doi: 10.1145/3338906.3338945.

[4] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–29, Nov. 2020, doi: 10.1145/3428270.

[5] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: a framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn Estonia: ACM, Aug. 2019, pp. 545–555. doi: 10.1145/3338906.3338925.

[6] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, London, ON, Canada: IEEE, Feb. 2020, pp. 261–271. doi: 10.1109/saner48275.2020.9054857.

[7] R. Huang, Q. Zhang, D. Towey, W. Sun, and J. Chen, "Regression test case prioritization by code combinations coverage," *Journal of Systems and Software*, vol. 169, p. 110712, Nov. 2020, doi: 10.1016/j.jss.2020.110712.