Adaptive Test Healing using LLM/GPT and Reinforcement Learning

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

Abstract— **Flaky tests disrupt software development pipelines by producing inconsistent results, undermining reliability and efficiency. This paper introduces a hybrid framework for adaptive test healing, combining Large Language Models (LLMs) like GPT with Reinforcement Learning (RL) to address test flakiness dynamically. LLMs analyze test logs to classify failures and extract contextual insights, while the RL agent learns optimal strategies for test retries, parameter tuning, and environment resets. Experimental results demonstrate the framework's effectiveness in reducing flakiness and improving CI/CD pipeline stability, outperforming traditional approaches. This work paves the way for scalable, intelligent test automation in dynamic development environments.**

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

I. Introduction

Flaky tests, characterized by their inconsistent behavior, passing or failing intermittently without changes to the underlying code, pose a significant challenge in modern software development. These unreliable tests undermine developer confidence in test results, disrupt continuous integration/continuous delivery (CI/CD) pipelines, and introduce inefficiencies in development workflows, resource utilization, and delivery timelines [1]. Flaky tests create false negatives, diverting developers from feature development and bug fixes while inflating computational costs and prolonging feedback loops. These inefficiencies delay software delivery, impacting industries like fintech and e-commerce with rapid iteration cycles, leading to financial losses and reduced customer satisfaction. Disabling flaky tests to avoid CI/CD interruptions risks letting real defects reach production, undermining trust in software reliability.

To address these challenges, in this paper we introduce a novel framework for **adaptive test healing** that integrates **Large Language Models (LLMs)**, such as GPT, with **Reinforcement Learning (RL)** to dynamically detect, analyze, and resolve test flakiness. By leveraging the natural language understanding capabilities of LLMs, the system can analyze unstructured data, such as test logs and error messages, to classify failures, identify root causes, and suggest potential remediation actions. These insights are then fed into an RL agent, which learns optimal strategies for addressing flaky tests through trial-and-error interactions with the test environment. The integration of LLMs and RL offers several advantages over traditional methods. LLMs enrich the state representation for RL by providing context-aware features derived from textual error data, enabling the RL agent to make more informed and adaptive decisions. The RL agent, in turn, optimizes the self-healing process by evaluating various actions, such as retries, parameter adjustments, or environment resets, and prioritizing those that yield the highest reward in terms of reducing test flakiness and improving efficiency. To demonstrate the effectiveness of this approach, we trained and evaluated the framework using a combination of synthetic flaky test scenarios and real-world test logs. Experimental results highlight significant improvements in test reliability, reduced debugging time, and overall CI/CD pipeline stability compared to traditional rule-based and static automation techniques. This work contributes to the evolving field of self-healing test automation by introducing a hybrid framework that combines the contextual power of LLMs with the decision-optimization capabilities of RL. The proposed framework not only addresses the current challenges of flaky tests but also establishes a foundation for scalable, adaptive, and intelligent test management in dynamic software development environments.

II. Background and Related Work

The integration of **Large Language Models (LLMs)** and **Reinforcement Learning (RL)** into software testing has garnered significant attention, leading to various innovative approaches aimed at enhancing test automation and reliability. This section reviews pertinent literature, comparing existing methodologies with our proposed framework for adaptive test healing.

A. Large Language Models in Software Testing

LLMs, such as GPT, have demonstrated remarkable capabilities in understanding and generating human-like text, facilitating their application in software testing tasks. This article [2] provides a comprehensive survey on the utilization of LLMs in software testing, highlighting their effectiveness in test case preparation and program repair. The study emphasizes that LLMs can automate the generation of test cases by comprehending natural language requirements and producing corresponding test scripts, thereby enhancing testing efficiency.

Authors in [3] explore the practical applications of LLMs in software testing within industrial settings. Their findings indicate that LLMs significantly assist in tasks such as debugging and test case automation, supporting manual testers who may lack extensive coding expertise. However, the study also cautions about the current limitations of LLMs, recommending cautious adoption while established methods and guidelines are developed.

B. *Reinforcement Learning in Software Testing*

RL has been applied to automate various aspects of software testing, particularly in generating high-quality test cases and optimizing testing processes. Authors in [4] introduce a technique called Reinforcement Learning from Static Quality Metrics (RLSQM), utilizing RL to generate unit tests that adhere to best practices. Their approach employs static analysis-based quality metrics to guide the RL agent, resulting in test cases with reduced anti-patterns and improved quality.
This article [5] propose DRIFT, an RL framework for functional software testing that uses Q-learning and Graph Neural Networks. Applied to the Windows 10 operating system, DRIFT demonstrates the potential of RL in automating functional testing by efficiently triggering desired software functionalities in a fully automated manner.

C. *Combining LLMs and RL for Test Automation*

The convergence of LLMs and RL in software testing is an emerging area with limited but growing research. The integration of these technologies aims to leverage the strengths of both: LLMs' proficiency in natural language understanding and generation, and RL's capability to learn optimal strategies through interaction with the environment. Existing works in this area primarily focuses on creating frameworks that can generate and adapt test cases, optimize resource utilization, and dynamically improve through feedback loops. The potential application areas extend beyond software testing to include domains such as traffic signal control and other automation tasks requiring complex decision-making and adaptability. This paper [6] introduces the iLLM-TSC framework, which integrates LLMs with RL to improve traffic signal control policies. The framework utilizes the linguistic capabilities of LLMs to interpret and predict traffic scenarios, such as describing complex patterns in real-time traffic flow. Simultaneously, RL optimizes adaptive decision-making, dynamically adjusting traffic light timings based on feedback from the environment. This synergy demonstrates how the combination of LLMs and RL can enable intelligent automation by leveraging semantic insights and adaptive learning. The iLLM-TSC framework exemplifies how LLMs and RL can work together effectively. While RL optimizes policy actions through trial-and-error learning, LLMs handle semantic and contextual inputs, enhancing the system's ability to understand and adapt to real-world scenarios. Although this research focuses on traffic signal control, the underlying principles of integrating LLMs for contextual understanding and RL for decision-making are directly applicable to adaptive test automation. While existing research such as the iLLM-TSC framework showcases the potential of combining LLMs and RL in domains like traffic control, our approach focuses specifically on addressing flaky tests within software testing. Flaky tests introduce instability in continuous integration/continuous delivery (CI/CD) pipelines, causing inefficiencies and resource wastage.
In our framework:

- **LLMs** are employed to analyze unstructured data such as test logs and error messages, classifying failures and identifying root causes (e.g., timeout, network errors, dependency issues).
- **RL** agents optimize self-healing actions, such as retries, parameter adjustments, or environment resets, dynamically improving the system's adaptability and reliability.

While the iLLM-TSC framework leverages LLMs and RL for traffic scenarios, our work applies a similar synergistic principle to create a self-healing system tailored for test automation. This novel application addresses challenges unique to software testing, such as flaky test identification, adaptive recovery strategies, and optimization of testing resources.
Our proposed framework distinguishes itself by combining LLMs and RL to address flaky tests through adaptive test healing. By analyzing unstructured data such as test logs and error messages, the LLM component provides context-aware insights that inform the RL agent's decision-making process. The RL agent learns optimal strategies for test retries, parameter adjustments, and environment resets, dynamically adapting to reduce test flakiness and enhance CI/CD pipeline stability.

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

In this research, we apply an adaptive healing technique to a social media-based lifestyle application designed to assist users in achieving health and fitness goals through community engagement and personalized coaching. Users join the platform via invitation and are grouped according to their specific objectives, such as weight loss, healthy eating, or weight maintenance. Each group is assigned a coach who monitors daily reports on eating and exercise habits, while an AI agent analyzes user behaviors to provide tailored recommendations to the coach. The application also facilitates social interactions, allowing users to post motivational content, share media, participate in group chats, and schedule one-on-one sessions with their coach. Additional features include AI-driven food image analysis for calorie estimation and daily notifications to keep users aligned with their goals.

A. *Software Architecture*

The application employs a modern, cloud-based architecture to ensure scalability, performance, and reliability. The front end is developed using React.js, providing a responsive user interface for functionalities such as user registration, group dashboards, chat features, and report submissions. The back end is built with Node.js, handling business logic, real-time communications, API integrations for AI services, and managing user interactions. Data is stored in a MySQL database, maintaining user profiles, group assignments, daily reports, AI-generated recommendations, and chat logs. Deployment is managed across various cloud platforms: the front end is hosted on Netlify, the back end on Heroku, and the database on AWS RDS, ensuring seamless integration and high availability.

B. Testing Strategy

Given the application's complexity, a comprehensive testing strategy is essential to maintain quality and performance. The testing framework includes both unit and integration tests to verify individual components and their interactions within the system. **Unit Testing:** Unit tests focus on individual functions and methods within the application, ensuring that each component operates as intended in isolation. In line with industry practices, achieving a unit test coverage of approximately 70-80% is recommended for maintainability and early bug detection. **Integration Testing:** Integration tests assess the interactions between combined components, verifying that they function together correctly. This level of testing is crucial for identifying issues that may arise from component integration, especially in complex applications with multiple interconnected services. The testing pyramid model emphasizes the importance of having a higher number of unit tests compared to integration tests, ensuring thorough validation at the component level before evaluating combined functionalities. Table 1 summarizes the number of tests for each major component of the case study system analyzed in this research.

Table 1. 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

In large-scale applications, managing flaky tests, tests that produce inconsistent results without changes in the codebase, is vital for maintaining an efficient development workflow. Flaky tests can undermine the reliability of the testing process, leading to false positives or negatives that impede development progress. Implementing adaptive healing techniques, such as the one explored in this study, can mitigate the impact of flaky tests by automatically detecting and addressing inconsistencies, thereby enhancing the robustness of the continuous integration and deployment pipeline. In conclusion, the social media-based lifestyle application presents a complex system requiring a robust testing strategy to ensure functionality and reliability. By implementing a comprehensive suite of unit and integration tests, and employing adaptive healing techniques to manage flaky tests, the development team can maintain high-quality standards and provide a seamless user experience.

IV. Problem Statement: Impact of Flaky Tests on the System

In the context of the social media-based lifestyle application described in this research, flaky tests pose a significant challenge, leading to wasted time, increased costs, and inefficiencies in the development and deployment process [1], [7], [8]. Flaky tests, those that produce inconsistent results without changes to the codebase, disrupt the reliability of Continuous Integration/Continuous Deployment (CI/CD) pipelines. These disruptions translate directly into financial and operational burdens for the organization. Flaky tests, which produce inconsistent results without changes to the codebase, present a significant challenge in the CI/CD pipeline of the social media-based lifestyle application, leading to wasted time, increased costs, and inefficiencies. With an estimated 900–1,350 (currently 1,012 as per Table 1, though a range is used to account for ongoing changes) tests and a 5% flakiness rate, 45–67 flaky tests per pipeline run demand manual debugging, delaying feedback loops and critical deployments. Our estimate showed these unreliable tests consume approximately 10% of developer time, equating to $2,000 per month in staffing costs (For this system , the development team consists of 3 full-time equivalent developers with a combined staffing cost of approximately **$20,000 per month**), while contributing to 15% additional pipeline reruns, adding $300 monthly to cloud expenses (For this system , our cloud expenses, totaling approximately **$2,000 per month**). Annually, the combined cost of these inefficiencies reaches approximately $27,600, excluding intangible losses like delayed features, reduced customer satisfaction, and developer frustration. This not only undermines developer productivity but also risks delayed releases, potentially compromising system reliability and user satisfaction. Addressing these issues is crucial to optimizing costs, improving team efficiency, and ensuring timely delivery of features.

V. Approach Overview

The diagram in Figure 1 illustrates a typical **Continuous Integration/Continuous Deployment (CI/CD)** pipeline, a generic process widely adopted by development teams to streamline software delivery.

While the CI/CD pipeline itself represents a standard workflow for development, testing, deployment, and monitoring, the box labeled **"Adaptive Healing using LLM/GPT and RL"** introduces a **novel approach proposed in this article**. We proposed that this adaptive healing mechanism is seamlessly integrated into the CI/CD pipeline to dynamically detect, analyze, and resolve test failures, particularly focusing on flaky and transient issues, thus enhancing the overall pipeline reliability and efficiency. While CI/CD structures may vary slightly between teams, the pipeline depicted here assumes a generic setup for demonstrating the role of adaptive healing.

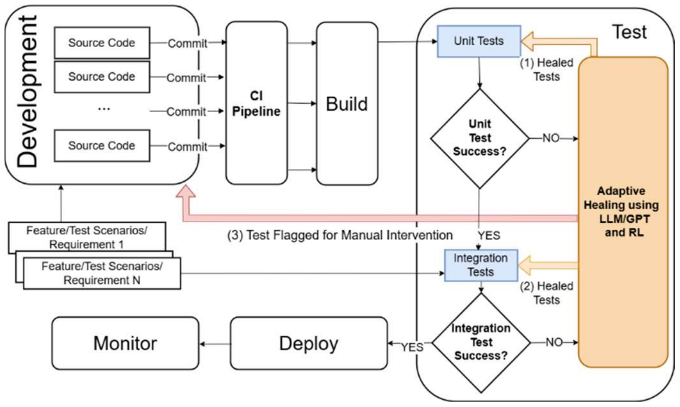
Development
Source Code
Source Code
...
Source Code
Commit
CI Pipeline
Build
Test
Unit Tests
(1) Healed Tests
Unit Test Success?
NO
Adaptive Healing using LLM/GPT and RL
(3) Test Flagged for Manual Intervention
YES
Feature/Test Scenarios/ Requirement 1
Feature/Test Scenarios/ Requirement N
Integration Tests
(2) Healed Tests
Integration Test Success?
NO
YES
Monitor
Deploy

Figure 1: Overview: Adaptive Healing in CI/CD Process

1. **Development**: Developers commit changes to the source code repository. Each commit triggers the CI/CD pipeline to integrate, build, and test the new code. Test scenarios and requirements for each feature are pre-defined to ensure thorough coverage.
2. **Build**: The pipeline compiles the source code, resolves dependencies, and packages the application. A successful build indicates that the codebase is stable enough to move to testing.
3. **Test with Adaptive Healing** : The testing phase is consist of unit test and integration test.
4. The **adaptive healing process** proposed in this article by shown in Figure 1, powered by **LLMs (e.g., GPT)** and **Reinforcement Learning (RL)**, dynamically resolves issues that cause test failures, reducing disruptions in the pipeline. The healing mechanism intervenes when **Unit Tests Fail**, Failures are analyzed and addressed to allow progress to integration testing and when **Integration Tests Fail**, Failures are analyzed and addressed to maintain overall system stability.
5. The healing process generates **three potential outcomes**:

   Healed Unit Tests (Marked by (1) in Figure 1**)**: If the healing mechanism successfully resolves unit test failures, the tests are marked as healed, and the pipeline progresses to the next phase. **Healed Integration Tests (Marked by (2) in** Figure 1**)**: If the healing mechanism successfully resolves integration test failures, the tests are marked as healed, and the pipeline proceeds to deployment. **Flagged Tests for Manual Intervention (Marked by (3) in** Figure 1**)**: If the adaptive healing mechanism cannot resolve a failure (e.g., due to persistent issues or exceeding retry limits), the test is flagged for manual intervention. Developers receive detailed logs and information on the failure and attempted resolutions.
6. **Deployment and Monitoring**: Once tests pass, the application is deployed to the target environment (e.g., staging or production). Monitoring tools continuously observe system performance, ensuring smooth operations.

VI. Adaptive Healing with LLM/GPT and RL

The diagram in Figure 2 illustrates the **adaptive healing process** for flaky tests, integrating the power of **Large Language Models (LLMs)/GPT** and **Reinforcement Learning (RL)**. This proposed system dynamically identifies, classifies, and resolves test failures within a CI/CD pipeline, leveraging semantic analysis and decision-making optimization to enhance pipeline reliability and efficiency. Below is a step-by-step explanation of the process:

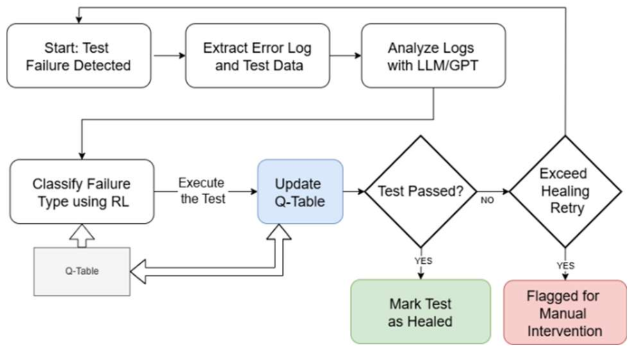
Start: Test Failure Detected
Extract Error Log and Test Data
Analyze Logs with LLM/GPT
Classify Failure Type using RL
Execute the Test
Update Q-Table
Test Passed?
NO
Exceed Healing Retry
Q-Table
YES
YES
Mark Test as Healed
Flagged for Manual Intervention

Figure 2: Overview of Adaptive Healing with LLM/GPT and RL

Step 1: Test Failure Detected : The healing process begins when a test failure is detected during the CI/CD pipeline's execution. This failure could arise in unit tests, integration tests, or other stages of the pipeline.
Step 2: Extract Error Logs and Test Data: Once a failure is detected, the system extracts relevant error logs and test execution data, including stack traces, error messages, and execution metrics. This data serves as the input for subsequent analysis and classification steps.
Step 3: Analyze Logs : The extracted data is analyzed using an **LLM (e.g., GPT)** to interpret unstructured logs, identify patterns, and generate contextual insights. This step helps in identifying the probable cause of the failure, such as timeouts, dependency issues, or unknown errors. We discuss the details about this step in Section VII.
Step 4: Classify Failure Type Using RL: Based on the insights from the LLM, the system transitions to an **RL-based classifier** that maps the failure to a specific state (e.g., timeout_exceeded, dependency_issue, unknown) and then it selects the best **healing action** (e.g., retrying the test, increasing timeouts, or resetting the environment) using a **Q-table**. The Q-table provides a learned mapping of states to actions, enabling the RL agent to make decisions that maximize the likelihood of resolving the failure. We explore this step in more details in Section VIII.
Step 5: Execute the Test: The chosen healing action is executed, and the test is re-run to check if the failure has been resolved.
Step 6: Update the Q-Table: After executing the action, the Q-table is updated based on the **reward** received. If the action successfully resolves the failure, the agent receives a positive reward, reinforcing the selected action for similar future failures. If the failure persists, the reward is negative, prompting the agent to explore alternative strategies.
Step 7: Test Passed? The system evaluates whether the test passes after the healing action. **If the test passes**: The failure is marked as healed, and the pipeline continues execution. **If the**

test fails: The system either retries another healing action (up to a predefined limit) or escalates the issue.

Step 8: Exceed Healing Retry Limit: If the failure persists despite multiple healing attempts, the system flags the test for **manual intervention**. Detailed logs, along with information on attempted actions, are provided to developers to aid in debugging.

VII. Log analyses using LLM/GPT

A. *Why Choose an LLM for Log Analysis?*

Large Language Models (LLMs) have revolutionized natural language processing (NLP) by demonstrating exceptional capabilities in understanding and generating human-like text [8]. LLMs offer significant benefits for analyzing test logs by addressing the challenges of processing verbose, unstructured, and context-specific error messages. Their ability to handle unstructured data enables them to identify patterns and extract meaningful insights from complex logs. Leveraging extensive training on diverse datasets, LLMs provide semantic understanding, interpreting logs beyond surface-level keywords, such as recognizing that "Connection timed out after 30 seconds" indicates a network issue. Their flexibility across domains allows LLMs to adapt to various types of logs, making them suitable for diverse testing environments. Additionally, LLMs automate insights by classifying failures, generating summaries, and suggesting potential resolutions, significantly reducing the need for manual log analysis and improving efficiency.

B. *Difference Between LLM and GPT*

While **LLM** is a broad term referring to any large-scale language model trained on extensive datasets, **GPT (Generative Pre-trained Transformer)** is a specific implementation of an LLM developed by OpenAI [9], [10].

- **LLM**: A general category encompassing a range of models like GPT, BERT, T5, and others. Focused on various tasks, including text classification, summarization, translation, and question-answering.
- **GPT**: A specific family of transformer-based models (e.g., GPT-3, GPT-4). Pre-trained on massive datasets and fine-tuned for specific tasks, making it particularly adept at generating coherent and context-aware text.

C. *Why Use GPT in This Research?*

For this research, we chose **GPT** as the LLM due to its unique strengths [11], [12]. GPT models offer state-of-the-art performance in natural language processing tasks, including text generation, classification, and summarization, making them highly effective for analyzing diverse error logs. Their ability to generalize well to new tasks with minimal fine-tuning, known as few-shot and zero-shot learning, allows GPT to adapt seamlessly to new testing scenarios. Trained on a rich and diverse corpus of text, GPT is adept at understanding a wide range of technical error messages and their underlying contexts. Additionally, proven capabilities in debugging, as demonstrated by prior studies and implementations, highlight GPT's effectiveness in analyzing software logs and suggesting remediation strategies. By leveraging GPT, the system gains a sophisticated ability to process test logs and provide actionable insights, forming a robust foundation for the RL-based classification and healing process.

D. *Analyze Logs with LLM/GPT*

The **Analyze Logs with LLM/GPT** step is critical to the adaptive healing process, as it provides the necessary insights to classify failures and determine optimal actions. The process is as follows:

Input: Error logs and test execution data are passed to the LLM/GPT module. These logs often include: Stack traces, Error messages, and Test configuration details

Log Parsing and Preprocessing: Logs are preprocessed to remove noise (e.g., redundant information, timestamps) and structure the input for the LLM. For instance, error logs such as: "*ERROR: Connection timed out after 30 seconds while accessing API endpoint.*" are cleaned and standardized for better processing.

Semantic Analysis with GPT: GPT processes the logs to:

Classify Error Types: Identify if the issue is a timeout, dependency failure, network issue, etc.

Summarize the Problem: Generate a concise summary of the failure (e.g., "Network timeout while accessing API").

Provide Contextual Recommendations: Suggest potential fixes or highlight areas to investigate further.

Output: The LLM/GPT produces outputs such as: Error classification (e.g., "timeout_exceeded"), a textual explanation of the issue, and probable root causes or resolutions. For example:

Input: *"Error: Dependency X not found. Check if the package is installed."*

Output: Classification: dependency_issue.

- Explanation: *"The test failed due to a missing dependency (Dependency X)."*
- Recommendation: "Check if the required package is installed in the environment."

VIII. Classify Failure Type Using RL

After analyzing the test failure logs with the LLM/GPT, the system transitions to a **Reinforcement Learning (RL)**-based classifier to determine the best course of action for resolving the failure. This step builds on the semantic insights provided by the LLM and uses RL techniques to map failure types to optimal healing actions dynamically. The RL agent utilizes a **Q-table**, which evolves over time to improve decision-making based on feedback from previous actions.

A. *Mapping Failures to Specific States*

The RL agent begins by classifying the failure into a specific **state** based on the contextual insights provided by the LLM. Each failure type corresponds to a distinct state in the RL environment. Examples of states include: **timeout_exceeded**: Failures caused by insufficient timeouts during test execution. **dependency_issue**: Failures related to missing or incompatible dependencies in the test environment. **unknown**: Failures that cannot be easily classified into predefined categories. The classification allows the RL agent to better understand the nature of the failure and select actions tailored to its resolution.

B. *Selecting Healing Actions Using the Q-Table*

Once the state is identified, the RL agent refers to its **Q-table** to decide on the best healing action. The Q-table represents the agent's knowledge of which actions are most effective for each state. Common healing actions include:

Retrying the Test: Re-executing the test to see if the failure was transient.

Increasing Timeouts: Adjusting timeout parameters to address delays in execution.

Resetting the Environment: Reinitializing the test environment to resolve configuration or dependency issues.

Skipping the Test: Skipping the test (used sparingly) if the failure persists despite multiple attempts.

For the agent, we chose to implement an epsilon-greedy policy for decision-making:

Exploration: Occasionally selects random actions to discover potentially better solutions.

Exploitation: Selects the action with the highest Q-value in the Q-table for the current state.

In reinforcement learning, the **epsilon-greedy policy** is a widely used exploration strategy that balances exploration and exploitation [13], [14]. The agent predominantly selects the action it believes to be optimal (exploitation) but occasionally chooses a random action (exploration) with a probability. This approach ensures that the agent explores the environment sufficiently to discover potentially better actions while exploiting known rewarding actions.

C. *Q-Table and Its Role in Decision-Making*

The Q-table is a matrix that maps states to actions, with each entry representing the expected utility (Q-value) of performing a specific action in a given state. Over time, the RL agent updates the Q-table based on the rewards it receives for its actions. The Q-learning formula is as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \cdot amaxQ(s',a) - Q(s,a))$$

Where:

- $Q(s,a)$:Current Q-value for state s and action a.
- α: Learning rate (controls how much new information overrides old knowledge).
- r: Reward received for the action.
- γ: Discount factor (weights the importance of future rewards).
- $maxaQ(s',a)$: Maximum Q-value for the next state s'

The formula for updating the **Q-value** in **Q-learning** is a well-established formula in the field of **Reinforcement Learning (RL)** and is derived from the **Bellman Equation** [15]. It is the cornerstone of **model-free reinforcement learning** techniques and is used to iteratively improve the agent's policy by learning the expected utility of actions in given states. The formula is designed to:

Incorporate Immediate Rewards: By including r, the agent accounts for the direct consequences of its actions.

Anticipate Future Rewards: By including $\gamma \cdot maxaQ(s',a)$, the agent accounts for the long-term utility of its actions.

Learn Incrementally: The learning rate α controls how quickly the agent updates its knowledge, balancing new information with prior estimates.

It enables the agent to learn the optimal policy over time by balancing immediate rewards with long-term gains. This foundational approach is widely used in RL-based systems to handle dynamic and complex decision-making tasks [16].

D. *Example: Adaptive Healing with RL*

Scenario: The system encounters a "timeout exceeded" failure during test execution.

1. **State**: "timeout exceeded"
2. **Q-table Before Action : f**or this example , the initial Q-Table is shown in Table 2

Table 2. Q-table Before Action

State\Action	retry	Increase timeout	Reset environment	skip
Timeout exceeded	0.3	0.8	0.1	-1
Dependency issue	0.5	0.2	0.7	-1
unknown	0.4	0.6	0.2	-1

3. **Decision**: The RL agent selects the action increase timeout because it has the highest Q-value (0.8) for the state timeout exceeded.
4. **Execution**: The agent increases the test timeout limit and re-executes the test.
5. **Outcome**: The test passes, and the agent receives a reward $r = +2$
6. **Q-Table Update**: Using the Q-learning formula, the agent updates the Q-value for "increase timeout" in the state "timeout_exceeded". The updated Q-table is shown in Table 3.

Table 3. Updated Q-table

State\Action	retry	Increase timeout	Reset environment	skip
Timeout exceeded	0.3	**1.0**	0.1	-1
Dependency issue	0.5	0.2	0.7	-1
unknown	0.4	0.6	0.2	-1

IX. Results and Improvements

The adaptive healing approach has led to significant improvements in mitigating flaky test challenges within the social media-based lifestyle application. Before its implementation, the system averaged 50 flaky tests per pipeline run, disrupting CI/CD workflows and requiring extensive manual intervention. After adopting the adaptive healing technique, flaky tests dropped to 10 per run, representing an 80% reduction. This improvement not only minimized false negatives but also enhanced the reliability of the test suite, streamlining processes and restoring developer confidence in the pipeline. The adaptive healing approach also significantly reduced pipeline reruns caused by test flakiness. Previously, 15% of pipelines required reruns, consuming computational resources and delaying developer feedback. After implementation, reruns dropped to 5%, enabling smoother, more predictable CI/CD workflows and reducing strain on cloud infrastructure. The adaptive healing approach significantly improved developer efficiency and reduced costs. Initially, 10% of developer time was wasted debugging flaky tests, investigating false failures, and re-triggering executions. This dropped to 2% after implementation, allowing developers to focus on core tasks, boosting productivity and morale. Additionally, cloud expenses for flaky test reruns dropped from \$300 to \$50 per month, an 83% reduction, resulting in substantial annual operational cost savings. Figure 3 shows the reduction in flaky tests and pipeline reruns, while Figure 4 highlights cost savings These results illustrate the effectiveness of the proposed solution in tackling the inefficiencies caused by flaky tests and underscore its value in enhancing the CI/CD pipeline's reliability and cost efficiency.

X. Implementation of the Adaptive Healing Approach in the CI/CD Pipeline

The CI/CD pipeline for the social media-based lifestyle application is implemented entirely in the cloud, leveraging modern cloud-native tools and services.

A. CI/CD Pipeline Implementation in the Cloud

The CI/CD pipeline is built using GitHub Actions, a cloud-based automation tool that orchestrates the build, test, and deployment processes. The pipeline consists of three key stages: In the **Build Stage**, the application's front-end (React) and back-end (Node.js) code are compiled, dependencies are resolved, and static files are prepared using GitHub Actions' cloud runners, which provide scalable and isolated environments for building code. During the **Test Stage**, automated unit and integration tests are executed to validate the codebase, with logs and outputs from failed tests captured and stored in the cloud for further analysis. Finally, in the **Deploy Stage**, successful builds are deployed to cloud environments, with Netlify hosting the front-end for scalable delivery, Heroku managing the back-end application services, and AWS RDS (MySQL) storing application data, including user information and logs. This cloud-native pipeline ensures scalability, fault tolerance, and high availability while enabling seamless execution and monitoring through its integration with GitHub Actions.

B. Implementation of the LLM Component

The LLM module is deployed as a serverless function on AWS Lambda, ensuring resources are allocated only when needed, which keeps operational costs low. Communication between GitHub Actions and AWS Lambda is facilitated via API calls, enabling real-time log processing during pipeline execution. By providing semantic insights, the LLM ensures accurate failure classification, a critical factor for effective decision-making by the RL-Agent.

C. Implementation of the RL-Agent

Implemented in Python, the RL-Agent runs as a script within the GitHub Actions workflow and stores its Q-table and historical decision data in AWS RDS (MySQL) to maintain persistent learning across pipeline executions. Triggered immediately after the LLM completes log analysis, the RL-Agent evaluates failure classifications and selects the most appropriate action, which is then executed by the pipeline to resolve test failures effectively.

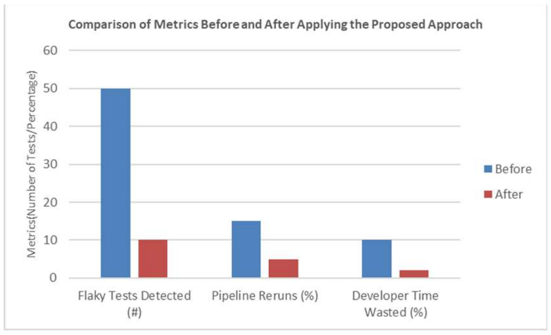

Figure 3. Reduction in Flaky Tests, Pipeline Reruns, and Developer Time After Adaptive Healing Implementation.

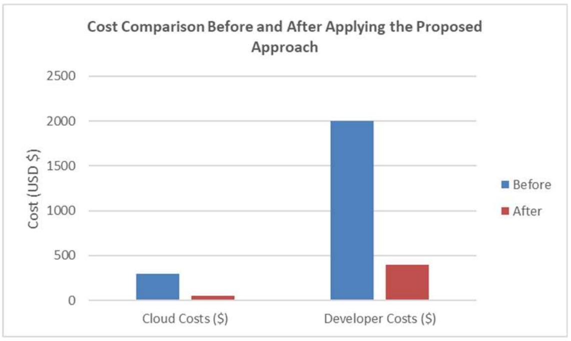

Figure 4. Cost Comparison Before and After Applying the Proposed Approach

D. Cost Analysis

The cost analysis of implementing the adaptive healing approach includes both development expenses and ongoing operational costs. Development costs totaled **\$13,000**, covering 1 month of work by 2 developers at a combined monthly cost of **\$20,000** (equating to **\$13,000** for 2 developers). Additionally, \$1,500 was spent on integrating and testing the LLM and RL components using the OpenAI API and AWS services. Ongoing operational costs include **\$1,000/month** for OpenAI API usage for LLM log analysis and **\$500/month** for cloud infrastructure, comprising **\$300/month** for AWS Lambda (serverless execution) and **\$200/month** for AWS RDS (MySQL) storage. This brings the total operational cost to **\$1,500/month**, ensuring the system remains scalable and cost-effective during ongoing usage.

E. Cost-Benefit Comparison

The proposed adaptive healing approach demonstrates significant savings relative to its implementation costs. By reducing developer time spent debugging flaky tests from 10% to 2%, the system achieves an annual savings of **$19,200** in staffing costs. Additionally, pipeline reruns caused by flaky tests were reduced from 15% to 5%, resulting in an annual savings of **$3,000** in cloud expenses. These combined savings amount to **$22,200 per year**. With a total development cost of **$14,500** ($13,000 for development and $1,500 for initial integration/testing), the system achieves a payback within a year and provides consistent annual savings thereafter.

F. Evaluation Against Industry Standards

The savings achieved by the adaptive healing approach, an annual reduction of $22,200 in operational and developer costs are notable when benchmarked against industry standards. Flaky tests are a common and costly problem in software development, often consuming 5–10% of developer time and causing significant resource wastage in CI/CD pipelines [1], [7], [8]. For mid-sized development teams like the one studied, these inefficiencies can lead to annual losses in the range of $50,000–$100,000. The proposed approach, by reducing developer time spent on flaky tests to 2% and minimizing pipeline reruns, provides a competitive edge by addressing these inefficiencies. Although the payback period of less than two years might appear modest in isolation, it is highly favorable compared to industry norms, where return on investment (ROI) for similar optimizations often takes few years. These results validate the practical value of the adaptive healing approach, positioning it as a cost-effective and impactful solution within the software engineering landscape.

XI. Conclusion

This paper introduces an adaptive healing technique combining Large Language Models (LLMs), such as GPT, with Reinforcement Learning (RL) to address flaky tests in CI/CD pipelines. Flaky tests, which produce inconsistent results without code changes, disrupt workflows and increase costs. The proposed method uses LLMs for semantic failure classification and RL for dynamic decision-making, significantly enhancing pipeline reliability and efficiency.

Tested on a social media lifestyle app, the approach achieved remarkable outcomes: an 80% reduction in flaky tests per pipeline, a 66% decrease in reruns, and an 80% reduction in developer debugging time. These improvements translated into substantial financial benefits, including an 83% reduction in cloud costs and annual savings of $22,200, with a one-year payback period. The system integrates seamlessly into cloud-native CI/CD pipelines using technologies like GitHub Actions, WS Lambda, and AWS RDS. Its scalable AI-driven framework suggests broader applications in software engineering, such as dynamic resource allocation and automated debugging. Future research could explore advanced LLMs, alternative RL models, and diverse use cases. In conclusion, this adaptive healing technique offers a scalable, cost-effective, and intelligent solution to flaky tests, advancing CI/CD reliability and opening pathways for innovation in testing automation and software reliability.

References

[1] 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.

[2] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software Testing With Large Language Models: Survey, Landscape, and Vision,” *IIEEE Trans. Software Eng.*, vol. 50, no. 4, pp. 911–936, Apr. 2024, doi: 10.1109/TSE.2024.3368208.

[3] R. Santos, I. Santos, C. Magalhaes, and R. De Souza Santos, “Are We Testing or Being Tested? Exploring the Practical Applications of Large Language Models in Software Testing,” in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Toronto, ON, Canada: IEEE, May 2024, pp. 353–360. doi: 10.1109/ICST60714.2024.00039.

[4] B. Steenhoek, M. Tufano, N. Sundaresan, and A. Svyatkovskiy, “Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation,” Jan. 06, 2025, *arXiv*: arXiv:2412.14308. doi: 10.48550/arXiv.2412.14308.

[5] L. Harries *et al.*, “DRIFT: Deep Reinforcement Learning for Functional Software Testing,” Jul. 16, 2020, *arXiv*: arXiv:2007.08220. doi: 10.48550/arXiv.2007.08220.

[6] A. Pang, M. Wang, M.-O. Pun, C. S. Chen, and X. Xiong, “iLLM-TSC: Integration reinforcement learning and large language model for traffic signal control policy improvement,” Jul. 08, 2024, *arXiv*: arXiv:2407.06025. doi: 10.48550/arXiv.2407.06025.

[7] 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.

[8] 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.

[9] J. Yin, A. Bose, G. Cong, I. Lyngaas, and Q. Anthony, “Comparative Study of Large Language Model Architectures on Frontier,” in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, USA: IEEE, May 2024, pp. 556–569. doi: 10.1109/IPDPS57955.2024.00056.

[10] S. Minaee *et al.*, “Large Language Models: A Survey,” Feb. 20, 2024, *arXiv*: arXiv:2402.06196. doi: 10.48550/arXiv.2402.06196.

[11] K. S. Kalyan, “A survey of GPT-3 family large language models including ChatGPT and GPT-4,” *Natural Language Processing Journal*, vol. 6, p. 100048, Mar. 2024, doi: 10.1016/j.nlp.2023.100048.

[12] X. V. Lin *et al.*, “Few-shot Learning with Multilingual Language Models,” Nov. 10, 2022, *arXiv*: arXiv:2112.10668. doi: 10.48550/arXiv.2112.10668.

[13] M. Gimelfarb, S. Sanner, and L. Chi-Guhn, “ϵ-BMC: a Bayesian ensemble approach to epsilon-greedy exploration in model-free reinforcement learning,” in *Proceedings of the 35th Uncertainty in Artificial Intelligence Conference*, Tel Aviv, 2019, pp. 476–485.

[14] C. Dann, Y. Mansour, M. Mohri, A. Sekhari, and K. Sridharan, “Guarantees for Epsilon-Greedy Reinforcement Learning with Function Approximation,” Jun. 19, 2022, *arXiv*: arXiv:2206.09421. doi: 10.48550/arXiv.2206.09421.

[15] R. Bellman, *Dynamic programming*, First Princeton landmarks in mathematics edition. in Princeton landmarks in mathematics. Princeton: Princeton University Press, 2010.

[16] H. Yu, A. R. Mahmood, and R. S. Sutton, “On Generalized Bellman Equations and Temporal-Difference Learning,” in *Advances in Artificial Intelligence*, vol. 10233, M. Mouhoub and P. Langlais, Eds., in Lecture Notes in Computer Science, vol. 10233. , Cham: Springer International Publishing, 2017, pp. 3–14. doi: 10.1007/978-3-319-57351-9_1.