National College of Ireland

Self-Healing CI/CD Pipeline Using LLM and Google ADK Framework

MSc Research Project
Cloud Computing

Vinay Nukalapati
Student ID: 24106470

School of Computing
National College of Ireland

Supervisor: Diego Lugones

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Vinay Nukalapati
Student ID:	24106470
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Diego Lugones
Submission Due Date:	28th January 2026
Project Title:	Self-Healing CI/CD Pipeline Using LLM and Google ADK Framework
Word Count:	9537
Page Count:	28

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Vinay Nukalapati
Date:	28th January 2026

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Self-Healing CI/CD Pipeline Using LLM and Google ADK Framework

Vinay Nukalapati
24106470

Abstract

Build failures in Continuous Integration/Continuous Deployment (CI/CD) pipeline is one of the greatest bottlenecks of modern software delivery where it wastes precious developer's time and leads to delays in the release cycle. This research introduces the design, implementation and evaluation of an autonomous self-healing system using large language models (LLMs) and agentic AI to detect, diagnose and recover from these failures of the system without human intervention. The system has a hierarchical architecture of six specialised agents coordinated via the Agent Development Kit (ADK), which runs an iterative fix-verify-decide loop in AWS infrastructure.

Comprehensive evaluation across 60 diverse build failures has shown 73% automated remediation success rate, which has been validated as it has achieved 80% Mean Time to Recover (MTTR) success, which was 42 minutes with manual intervention and only 8.6 minutes with autonomous intervention. Comparative analysis with state of the art baseline research performance patterns reveals where effectiveness of the system is highly correlated with availability of error localization. Furthermore, results establish that frontier models (e.g., GPT-4) are critical in production readiness, much better than previous generations. These results prove for the first time that autonomous software engineering is not only an aspirational concept, but a working reality for well-scoped tasks, and it is a scalable solution for resolving the CI/CD reliability issues.

Keywords: Self-healing CI/CD, Agent Development Kit (ADK), LLM automation, pipeline orchestration, autonomous DevOps

1 Introduction

Modern software delivery is critically dependent on the use of automated Continuous Integration and Continuous Deployment (CI/CD) pipelines, which help orchestrate build, test and deployment tasks to help boost velocity of release and quality. As organisations embrace microservices, multi-environment deployments and security by design, these pipelines are becoming more complex and offer many failure modes; compilation errors, dependency conflicts, test regressions, misconfigured build scripts and incorrect infrastructure configuration choices, that can cause the breakdown of delivery schedules and impact system's reliability. There are empirical analyses highlighting the frequency and cost of such failures: for example, an analysis of more than 1.4 million failed builds in open source Java repositories found that compiler errors constituted about 11% of

failed builds with many projects suffering habitual failure with issues of build-scripts or configuration problems) (Zhang et al.; 2019). In practice, pipeline instability has produced greater developer interruptions, higher mean time to recovery (MTTR) and high amounts of manual effort required from engineering teams to both diagnose and recover failed pipelines (S.r and Mathew; 2025).

The idea of self-healing systems, those that can self-recover with the aid of contextual telemetry to detect the fault, identify the cause and dissolve the fault with minimal human intervention, has been the topic of increasing interest in infrastructure and cloud management research (Shabariram et al.; 2024). However, the use of self-healing principles in CI/CD pipelines is quite unexplored in comparison to the operational importance of build-time failures in continuous delivery pipelines. Recent work has proven the efficacy of large language models (LLM) for certain types of remediation tasks.Yet previous research has been dominated by scanner-based vulnerability fixes and not the much more common and relatively infrequent class of pipeline-based build failures that directly prevent deliveries.

This thesis aims to fill this gap by designing, implementing and assessing an integrated, production-oriented Self-Healing CI/CD design that suggests an extension of the self-healing abilities to build failure remediation. The system that has been implemented couples AWS pipeline infrastructure (CodePipeline, CodeBuild, EventBridge, CloudWatch Logs, and Lambda) with a multiagent remediation service using Google's Agent Development Kit (ADK). In case any CodeBuild failed, an EventBridge rule triggers a Lambda error handler which extracts contextual CloudWatch logs and sends it towards an ADK API server. The ADK service–which runs on Gemini 2.5 Flash models runs a hierarchical agent workflow, comprising a Log Analyzer agent, Repository Setup agent, iterative Fix-Verify-Decide malignant cycle (Error Remediation, Build Verification, Loop Decision agents) and Code Finalizer that makes fixes and creates a pull request. This event driven, end-to-end pipeline is aimed at helping to detect and fix build failures autonomously, and in a way that maintains human control over them via pull request review.

1.1 Aim of the study

The main goal of this research is to investigate the feasibility of autonomously identifying and resolving build errors by performing detection, diagnosis, and repair within semi-autonomous, LLM-based automobile systems integrated into current CI/C adquirir pipelines, in a confident construction distribution procurement and economical way. Concretely, the goal of the work is to deploy a system that would have a similar reasoning and iterative, reactionary, corrective behaviour as a human developer - digging out structured failure information from logs, testing fixes with automated test builds and validates them with similar automated tests, reaching a stable build state with minimum human interaction.

1.2 Research Question

How to effectively integrate an AI-based, LLM-powered multi-agent system that autonomously identifies, interprets, and remediates build failures in an event-driven AWS CI/CD pipeline to accurately pull out root causes and generate fixes using iterative reasoning and

contextual awareness to improve reliability, efficiency, development efforts, and scalability?

1.3 Objectives of the Research

The research objectives for this thesis are:

- Design a multi-agent self-healing architecture for orchestrating the build failure detection, diagnosis, and remediation, verification, and finalisation steps.
- Implementing agents doing specialised tasks using the Google Agent Development Kit (ADK) for log analysis, repository setup, error remediation, build verification, loop decision and code finalisation.
- Integrate agent system with AWS CI/CD infrastructure namely CodePipeline/-CodeBuild, Lambda and CloudWatch to achieve an event driven remediation pipeline.
- Use of Gemini 2.5 Flash models to actuate agent reasoning and patch synthesis in an iterative fix-verify-feedback loop architecture
- Evaluate system performance based on curated and instrumented failure scenarios including success rate of repairable issues, average iterations to fix issues, regression incidence (test failures), developer intervention rate and cost per fix (infrastructure and model usage).
- Compare the system to baseline remediation methods (manual debugging and one shot LLM fixes) and analysis of failure modes, limitations and security considerations.

1.4 Outline of the Report

This report is structured as follows: In Section 2, a literature review on the related context in CI / CD automation, self-healing systems, ADK design, and LLM-based DevOps is carried out. Section 3 explains methodology of proposed research, Section 4 discusses how the system will be designed, research specifications, next Section 5 will provide details about research resources and implementation. Section 7 shows how evaluation will be done.

2 Related Work

2.1 Automated Error Detection and Healing in CI/CD Environments

Fault detection and healing are still essential problems in CI/CD pipelines, and many methods have been suggested, both based on conventional machine learning methods and more recent GPT-based and deep learning models. An example is (Johnphill et al.; 2024), which used a Multinomial Naive Bayes (MNB) model to predict the log-level status of anomaly at the OS level, which showed good recall with various operating systems in real-time performance monitoring. It is however, not suitable on adapting to complex or brute-force failure modes of pipelines due to its dependence on fixed thresholds.

On the other hand, (Mani and Attaranasl; 2025) discussed the application of GPT models with Reinforcement Learning (RL) to identify and fix flaky tests. With the use of language comprehension, their system could query the log trends and automatically fix problems in pipelines that run on AWS Lambda and GitHub Actions. This emphasizes the importance of contextual knowledge: the MNB-based models are excellent in the categorization of logs but LLM+RL models go further and analyze the semantics of logs and applies polices dynamically to heal them.

Based on these views, (Gaddamwar et al.; 2024) suggested a method that employs transformers and Graph Neural Networks (GNNs) to learn the software structure and software behaviors in a runtime task, with a high accuracy of 91.4% of bugs detected and low latency in repair work. Although this approach is effective, it relies on large labeled datasets, and does not have protocol coordination mechanisms in real-time. (Hrusto et al.; 2023) and (Gerber et al.; 2024) therefore examined anomaly detection in DevOps pipelines with unsupervised deep autoencoders, associated with resistance to never-before-seen failures, but failed at orchestrating pipelines at scale. Further, as a way to improve resilience, (Nunavath; 2025) suggested about self-healing that should be incorporated into Kubernetes-native CI/CD systems, which, however, should be based on multi-agent collaboration frameworks as it was not mentioned.

A common pattern in these studies is one of a movement toward more autonomous CI/CD healing, using classification, adaptive learning and continuous modeling. Nevertheless, they do not have enough agentic decision-making, orchestration, and coordinated multi-agent collaboration. The proposed research is based on these gaps and aims at introducing agentic frameworks that can produce fully self-governing protocol-driven pipelines that have built in healing capabilities.

2.2 Agentic AI enhancements

The recent emergence of Agentic AI is changing how intelligent systems are run in the long term, which can form a structural base to answer the research question of how autonomous agents can facilitate self-healing CI/CD pipelines.

According to (Durante et al.; 2024), agentic systems are systems with persistent memory and multimodal reasoning capability and act in a context-directed manner as opposed to the conventional request-response AI. This directly underpins the research of self-healing CI/CD pipeline agents, as these systems would need to recollect past failures, reason about different telemetry streams (e.g. build logs, test reports), and reassemble suitable remediation actions based on what happened in the past.

Similarly, (Acharya et al.; 2025) builds on this basis to offer an in-depth examination of the autonomous intelligence capabilities needed to accomplish complex goals. This survey provides the theoretical framework that is required to answer the research question on how agentic AI can autonomously handle and recover CI/CD pipeline failures without human intervention. The focus of the survey on autonomous decision-making is consistent with the fundamental needs of self-healing pipeline systems, which need to act autonomously in a variety of failure cases. The survey outlines key characteristics, including self-assessment, preparation, and long-term adaptability which could be applied in the research being conducted using Agent Development Kit (ADK) and intermittent agentic interventions, which directly responds to the technical implementation demands of the research question.

In the similar manner, (Ismail et al.; 2025) demonstrate that in security pipelines,

automated decision-making is possible with the deployment of the LLM-driven agents, which once again demonstrate the possibility of using LLM in the CI/CD systems to guarantee the detection of failures such as the tollgate failures or policy breaches and their subsequent correction. The work is quite pertinent to the research question since it shows the real-life applications of the concept of autonomous decision-making in pipelines and proves that intelligent agents can be successfully deployed to detect and fix complex failure modes of the system.

(Murugesan; 2025) also helps to put into perspective the transformative potential of agentic systems in the context of the focus area of the research question, autonomous pipeline management. This article explores the consequences of implementing autonomous decision-making systems in the critical infrastructures, which is directly connected to the research question focus on the creation of reliable self-healing CI/CD systems. The discussion of issues and mitigation measures discussed in the paper is an important piece of information to build strong agentic pipeline systems. . The practical implementation of these concepts is justified by (Siachamis et al.; 2025), which is a real-life example of the implementation of AI-powered self-healing pipelines using large language models and CI/CD integration. This paper is a direct answer to the research question in that it shows how LLMs may be incorporated into software development pipelines to support autonomous failure detection and remediation, which offers an established methodology to the proposed research strategy.

Besides, (Fu and Xie; 2025) propose that verifiable knowledge should be included using AI oracles to reduce hallucinations in agent responses. This methodology is central to the research question, in that it will ensure the proposed ADK-leveraged system has always has the correct runtime context (e.g., versioning, vulnerabilities, policy states, etc.). Reliability in autonomous pipeline work becomes significant with the integration of verifiable knowledge systems, which is the direct answer to the focus of the research question in developing reliable self-healing mechanisms.

The workflow and security that the research question required are further discussed by (Bajpai and Lewis; 2022), which targets the topic of modern CI/CD pipeline workflows and security. This research contains the essential information of the security concerns of the autonomous operations of a pipeline, so that the self-healing abilities do not affect the integrity of the development process. The work also provides security frameworks that need to be incorporated in any autonomous pipeline system, and it is directly informative of what is required to implement the research question.

Together, these articles provide a solid foundation of the capability to have agentic workflows, LLMs, and secure context protocols coordinate to generate self-healing DevOps pipelines, which directly answer the research question. Combination of autonomous decision making, persistent memory systems, verifiable knowledge systems and proven implementation methodologies offers great theoretical and empirical evidence to the creation of intelligent agent with the ability to be used in autonomous CI/CD pipeline failure identification and recovery.

2.3 Agent Development Kit and Context-Aware Automation

Recent breakthroughs in agent-oriented artificial intelligence platforms, like the Agent Development Kit (ADK) from Google, have made it possible to create modular, multi-agency, self-reasoning, periodically solving a problem, and managing complex workflow

systems [1]. (Xu and Shatz; 2003) states ADK offers organised primitives for defining specialised agents, managing tool usage and for sequencing or looping behaviours and thus may be classically suited for tasks that require continuous feedback, contextual adaptation and decision-making in the same manner that human developers might.

The research conducted by (Siachamis et al.; 2025) is considered as foundational research for the proposed system.(Siachamis et al.; 2025) propose an open source LLM-based DevSecOps pipeline for automated security vulnerability remediation based on SAST and DAST analyses, proving the feasibility of self-healing behaviours for CI/CD environments. They have used MAPE-K architecture that merges inputs from their scanner with large language versions and tailored re-validation to fueled upon iterating approves from the creation and attempt of security patches. While this work is an important foundation, it is narrow in scope when it comes to focusing some of the fundamentals of vulnerability repair in only one benchmark application, and does not discuss broader classes of software failures such as build fail happens, dependency conflicts, or compilation regressions, which are common failures that people deal with in their CI pipelines. This thesis is a further extension of the concept of self-healing software by addressing the problem of automated recovery from build-time failures. Leveraging ADK and Gemini models, the proposed multi-agent system analyzes build logs and uses meaningful code modifications based on a specific context while invalidating fixes in a system, in an iterative fashion, via a systematic remediation loop, thus addressing an important gap in current self-healing software research.

On the other hand, the practical use of context-aware automation in CI/CD is justified by (Ashtagi et al.; 2025) which proves that automated systems are capable of improving the stability of a given code and speeding up the deployment processes without neglecting the comprehensive monitoring of the system. The present work directly answers the research question by presenting how resilient CI/CD processes can be implemented with the help of cloud services and DevOps practices, which gives empirical evidence of the feasibility of automated pipeline recovery mechanisms that are capable of functioning without human intervention.

Recent research by (Lumer et al.; 2025) has realised automatic tool selection using embeddings and API retrieval through frameworks such as ScaleMCP allowing workflow adaptation to make use of the available tools and the context. This feature directly responds to the research question focus on autonomous decision-making by showing how the agents are able to dynamically choose the most suitable remediation tools depending on the failure situation that they have faced in CI/CD pipelines.

(Wan et al.; 2018) introduces new context-aware entities that have cognitive capabilities to optimise the production process. Although the study is specialized in the industrial IoT, their results are essential to the development of intelligent automation in sophisticated settings, particularly in the context of context-aware systems, which directly feeds the approach to the research question in developing context-sensitive pipeline recovery systems.

(Koc et al.; 2025) generalizes them in the development setting where it promotes telemetry-conscious interaction and demonstrates how agents use contextual cues (e.g. CI failure logs or build statistics) to reset execution directions and stimuli. This article is a direct answer to the research question because it presents real-world ways in which agents can interpret and act on pipeline telemetry data, so that they can make autonomous decisions based on real-time system health and past failure data.

[1] `https://google.github.io/adk-docs/`

The standardized communication protocols and context-aware automation features of ADK and empirically proven resilient pipeline architecture converge to create a strong technical basis to answer the research question of what autonomous pipeline failure detection and remediation mechanisms.

2.4 Context-Driven DevOps Automation and Intelligent Agent Coordination

The DevOps approach outlined by (Bokkena; 2024) highlights how Large Language Models (LLMs) can enhance cloud infrastructure by enabling intelligent decision-making, particularly in predictive maintenance and resource allocation. In contrast, (Jaafar et al.; 2018) propose a workload management mechanism in a collaborative software environment, emphasizing agent interactions and task-sharing. While (Bokkena; 2024) focuses on the reactive and predictive control capabilities of LLMs, (Jaafar et al.; 2018) examine agent dynamics under fairness and task-efficiency constraints. Both studies aim to ensure system stability, albeit through different means: one via language-driven automation and the other through rule-based agent ethics.

According to (Pahune and Akhtar; 2025), the LLMOps transition framework prioritizes managing infrastructure complexity, monitoring, and customization tools such as LangChain. This aligns with (Bokkena; 2024) perspective on LLMs adapting dynamically to changing environments, though LLMOps is more oriented toward lifecycle operations rather than real-time optimization. Meanwhile, the agent-based approach from Jaafar et al. (2018) lacks scalability but offers valuable insights into ethical considerations within automated teams, an aspect crucial for LLM-based orchestration systems.

While existing studies have shown the increasing promise of AI-driven automation in DevOps and pipeline security approaches, literature analysis offers a distinct insight into the dearth of work done to extend the correct principles of self-healing to build-time failure remediation in real-world CI/CD settings. Existing studies mainly focus on vulnerability repair, static analysis augmentation or theoretical frameworks of autonomic computing with little attention given to the exploration of fully autonomous systems that can diagnose and correct build error in pipeline. Recent developments in agent-oriented systems like the Google Agent Development Kit (ADK) coupled with the reasoning power of state-of-the-art large language models like Gemini are providing a new chance to operationalise self-healing behaviours in an operational, event-driven context. These technologies facilitate modular, context-based and iterative remediation processes, implementing the diagnostic and corrective actions of human developers. Based on these insights, the current research proposes a multiagent integrated self-healing system based on using AWS pipeline infrastructure and ADK for autonomous agents to address this under explored problem. This work thus aims to build upon the work already done to move (self-healing) automation beyond the realm of vulnerability scanning and into the problem space of build failure recovery, both in terms of introducing a novel architecture and leveraging empirical evidence in support of the feasibility of autremediation in modern CI/CD pipelines.

(Wan et al.; 2018) introduces new context-aware entities that have cognitive capabilities to optimise the production process. Although the study is specialized in the industrial IoT, their results are essential to the development of intelligent automation in sophisticated settings, particularly in the context of context-aware systems, which directly feeds the approach to the research question in developing context-sensitive pipeline recovery

systems.

(Koc et al.; 2025) generalizes them in the development setting where it promotes telemetry-conscious interaction and demonstrates how agents use contextual cues (e.g. CI failure logs or build statistics) to reset execution directions and stimuli. This article is a direct answer to the research question because it presents real-world ways in which agents can interpret and act on pipeline telemetry data, so that they can make autonomous decisions based on real-time system health and past failure data.

The standardized communication protocols and context-aware automation features of ADK and empirically proven resilient pipeline architecture converge to create a strong technical basis to answer the research question of what autonomous pipeline failure detection and remediation mechanisms.

2.5 Context-Driven DevOps Automation and Intelligent Agent Coordination

The DevOps approach outlined by (Bokkena; 2024) highlights how Large Language Models (LLMs) can enhance cloud infrastructure by enabling intelligent decision-making, particularly in predictive maintenance and resource allocation. In contrast, (Jaafar et al.; 2018) propose a workload management mechanism in a collaborative software environment, emphasizing agent interactions and task-sharing. While (Bokkena; 2024) focuses on the reactive and predictive control capabilities of LLMs, (Jaafar et al.; 2018) examine agent dynamics under fairness and task-efficiency constraints. Both studies aim to ensure system stability, albeit through different means: one via language-driven automation and the other through rule-based agent ethics.

According to (Pahune and Akhtar; 2025), the LLMOps transition framework prioritizes managing infrastructure complexity, monitoring, and customization tools such as LangChain. This aligns with (Bokkena; 2024) perspective on LLMs adapting dynamically to changing environments, though LLMOps is more oriented toward lifecycle operations rather than real-time optimization. Meanwhile, the agent-based approach from Jaafar et al. (2018) lacks scalability but offers valuable insights into ethical considerations within automated teams, an aspect crucial for LLM-based orchestration systems.

While existing studies have shown the increasing promise of AI-driven automation in DevOps and pipeline security approaches, literature analysis offers a distinct insight into the dearth of work done to extend the correct principles of self-healing to build-time failure remediation in real-world CI/CD settings. Existing studies mainly focus on vulnerability repair, static analysis augmentation or theoretical frameworks of autonomic computing with little attention given to the exploration of fully autonomous systems that can diagnose and correct build error in pipeline. Recent developments in agent-oriented systems like the Google Agent Development Kit (ADK) coupled with the reasoning power of state-of-the-art large language models like Gemini are providing a new chance to operationalise self-healing behaviours in an operational, event-driven context. These technologies facilitate modular, context-based and iterative remediation processes, implementing the diagnostic and corrective actions of human developers. Based on these insights, the current research proposes a multiagent integrated self-healing system based on using AWS pipeline infrastructure and ADK for autonomous agents to address this under explored problem. This work thus aims to build upon the work already done to move (self-healing) automation beyond the realm of vulnerability scanning and into the problem space of build failure recovery, both in terms of introducing a novel architec-

ture and leveraging empirical evidence in support of the feasibility of autremediation in modern CI/CD pipelines.

2.6 Research Gap and Contribution of This Work

The literature reviewed indicates that the current state of research in CI/CD automation and self-healing systems is mostly based on anomaly detection, vulnerability repair, flaky test repair, or conceptual agentic models. Though big language models and agent-based solutions have shown themselves to work in particular and controlled settings, the vast majority of available solutions fail to target autonomous correction of build-time failures in actual, event-driven CI/CD pipelines. Specifically, the multi-agent orchestration, iterative verification, and intensive integration with the cloud-native CI/CD infrastructure have not been studied thoroughly yet.

To fill these gaps, this research makes the following contributions: (i) a multi-agent system to detect, diagnose and fix build failures; (ii) combining an event-driven AWS CI/CD pipeline with an ADK-based agentic remediation workflow; (iii) an iterative fix - verify-decide loop to allow agents to fine-tune solutions using build feedback. The paper also offers an empirical assessment of the system under controlled build-failure conditions, remediation recovery time, and the effects of the capability of the LLM model.

3 Methodology

This chapter presents the methodological framework that is to be followed for the design, implementation, and evaluation of the autonomous self-healing CI/CD system. The approach follows similar established practices in the software engineering research and refers directly to the relevant research in automatic program repair, self-healing computing and LLM aided DevOps workflows.

3.1 Research Methodology

The research procedure is based on the principle of simulating the workflow of an experienced developer when diagnosing and repairing build failures. Prior research research in automated software repair has focused the fact that beliefs iterative reasoning and refinement of contexts together with incremental patch validation, often secure better than single shot code generation strategies. This insight offers the conceptual basis for the methodology which in going to be applied in this research. The research will start by analysing the current literature gap on self-healing CI/CD pipelines, and then will go on to describe the requirements for a system that can autonomously detect, interpret, and fix build failures on modern software delivery environments.

A full system architecture will then be designed and implemented, which combines an event-driven architecture consisting of AWS infrastructure, in conjunction with a multi-agent remediation workflow using the Google Agent Development Kit (ADK). The system will be conceptualized using Infrastructure-as-Code to ensure reproducibility, traceability and consistency of the experimental runs. AWS CodePipeline and CodeBuild will be made to mimic the use of real-life CI/CD behaviour and to generate real build logs and failure scenarios. An Amazon EventBridge rule and a Lambda Error Handler will be developed to monitor pipeline failure and extract rich information from CloudWatch Logs to provide the details of the failure to the self-healing agent.

The ADK-based multi-agent system will be implemented in the form of a hierarchical workflow consisting of log analysis, repository setup, iterative remediation, build verification agents, and pull request creation agents. After both the system components are up and running, they will be connected by a Rest interface to communicate smoothly between the AWS event driven detection layer and the ADK remediation layer.

Once the system is fully implanted, a series of examples of controlled build-failure scenarios will be built in such a way as to test the system with different types of errors. The system will be executed multiple times on these scenarios and raw data would be collected from AWS logs, ADK agent traces, and intermediate state transitions as well as GitHub commit histories. These data will form the empirical basis for the evaluation which will be carried out using descriptive and comparative statistical techniques in order to determine the effectiveness, reliability and efficiency of the proposed approach as well.

3.2 Implementation Methodology

The system will be developed as two cooperating but independently scalable components. The first component which is an AWS-based failure detection and reporting layer will monitor the outcome of builds, extract the failure logs from the CloudWatch, detect the occurrence of errors using the EventBridge, and pass the structured diagnostic information to the self-healing agent. By using Terraform to provision all cloud resources, the research will ensure that the infrastructure is reproducible and version controlled throughout the development and testing of the research.

The second component, an ADK-based multi-agent remediation layer, will realise the completion of the entire self-healing workflow. Individual agents will be defined using the Gemini 2.5 Flash model and will be organised hierarchically using the SequentialAgent and LoopAgent constructs of ADK. Each agent will have its own responsibility of parsing build logs, preparing repository, generating candidate fixes, error analysis, dependencies installation, builds etc. Agents will be supported by a collection of deterministic tool functions for cloning repositories, editing files, analysing errors, installing dependencies, executing builds. A common state dictionary will be used for purposes of communication and state consistency between the agents.

The remediation loop will be set up to perform for a max of 5 iterations, allowing for the agent to analyse the new set of information that is being surfaced as a result of a failed attempt and dynamically change its strategy, in in line with human debugging behaviour. This iterative process is expected to let the system converge to the building state's success with keeping the stability and traceability, in the process to maintain attempts to build.

Once implemented the ADK agent service will be deployed as a containerised API server. The service will support the keeping of per-build session contexts, reporting of failure to the AWS Lambda Error Handler, support of the entire multijourney process with workflows, and structured re-results of the decisions made, fixes applied, and pull requests created. This deployment strategy will also aid in subsequent scalability, monitoring and reproducibility of the experiments.

3.3 Evaluation Methodology

The Self-Healing Agent is assessed with the help of a quantitative methodology based on effectiveness, efficiency and quality of code. Performance is quantified in the form of

Fix Rate, iterations, Mean Time to Repair (MTTR) and automated linting score. All experiments will be performed in a controlled, containerized environment and cover six types of defects that include syntax errors, logic errors, configuration and dependency errors, test failures, multi-file reasoning and cascading errors. In addition, a collection of twenty representative dataset is used to compare the performance of MTTR with the historical performance of human developer. Results are interpreted relative to two baselines, one-pass non-agentic LLM for quantifying the benefit of the agentic loop, and human performance benchmarks as a reference with regards to real-world applicability.

3.4 Dataset Selection

To ensure statistical significance and result reliability, each experiment was repeated 20 times using the curated dataset. These repetitions were necessary to account for variability in LLM responses as well as environment-specific fluctuations, thereby providing a more robust measure of system consistency.

The evaluation was conducted across a diverse set of 30 distinct build-failure scenarios, ranging from basic syntax errors to more complex dependency and configuration-related failures. This diversity was intended to comprehensively assess the diagnostic and remediation capabilities of the agent-based system.

In addition to curated failure cases, the system was benchmarked against non-curated, real-world build errors extracted from development logs. This evaluation was performed to assess the generalisability of the proposed approach and to ensure that its effectiveness was not limited to predetermined or idealised failure conditions.

Descriptive and comparative statistical analyses were employed to derive performance metrics from the experimental results. Specifically, the mean was used as the primary statistical measure to compute average performance across the 20 repetitions for each of the 30 error scenarios.

The use of the mean, rather than alternative statistical measures, ensures that all data points from repeated executions contribute to the final evaluation. This approach is particularly sensitive to outliers and infrequent failure cases, which are critical to identifying potential instability in production-grade CI/CD environments. Consequently, occasional system failures are captured and reflected in the overall performance assessment.

4 Design Specification

This section includes the design specification of the self-healing remediation system, which includes a distributed system with two parts interconnected to each other: one is the AWS-based infrastructure layer that performs the continuous integration monitoring and eventual detection of events, and the second is an intelligent agent layer using Google's Agent Development Kit (ADK) that performs the autonomous remediation of error as shown in Figure 1.

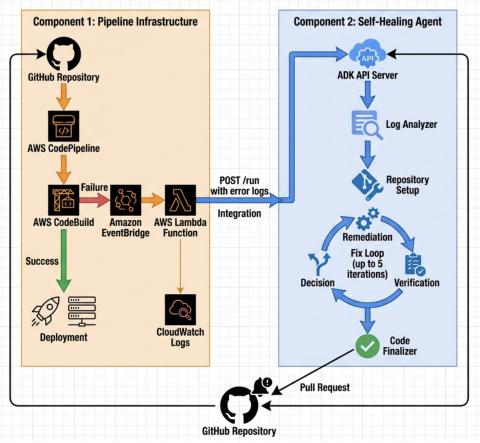


Figure 1: System Architecture

4.1 System Architecture

4.1.1 Multi-Tier Design of the Architecture

A three-tier architectural pattern is followed which ensures the modularity, scalability and maintainability. The infrastructure tier, implemented with AWS cloud services, creates the basic capabilities as continuous integration, build execution and events driven triggers. The integration tier, implemented using the AWS Lambda functions and rules in the Eventbridge subsystems, is the orchestration tier that links build failure detection with the remediation subsystem. The intelligence tier, built on the Google Agent Development Kit framework, represents the self autonomy of the agent in making decisions and changing those codes used to provide the self-healing functionality.

The infrastructure tier is based on AWS CodePipeline as the hub of the continuous integration and deployment process. Three steps are defined in sequence in the pipeline architecture where a source stage is set up to track GitHub repositories for code changes using the OAuth authentication method, a build stage is set up that uses AWS CodeBuild for compilation exercise and testing operation setup while provisioning artifact promotion from build machine within the deployment stage. CloudWatch Logs integration is included for complete logging capabilities in order to conduct a formidable analysis of build artifacts and trace execution. Amazon EventBridge is chosen as the event routing mechanism because of its capability to philtre and route events based on complex rules involving pattern matching and so allows for precision triggering of remediation workflows for only certain conditions (build failure).

4.1.2 Agent Based Remediation Framework

The remediation subsystem is designed using a hierarchical pattern of multi-agents, provided by the Google Agent Development Kit. This framework allows the breaking down of complex problem solving workflows into specialised, composable bits (agents) that work together using a shared state mechanism. A sequential execution model is used at the top level of orchestration so that we achieve deterministic dead through of the remediation pipeline while a loop baseline execution model is embedded in the fix application phase for the iterative refinement of solutions.

The agent hierarchy is organized with root SequentialAgent taking care of four main phases, log analysis, repository preparation, iterative fix application, and code finaliza-

tion. Within the iterative phase to fix an application, a LoopAgent is used to capture the try - verify - decide cycle and offers a way of retry logic with exponential complexity handling. Each agent in the hierarchy is set up with permissions to use domain-specific tools and is given explicit input expectations as well as output contracts to allow it to execute statelessly and make the tests simple.

State management of the whole multi-agent system is implemented by means of a shared dictionary structure: the inter-agent communication medium. This choice of design is for the sake of conserving execution context across agent boundaries, while not wiping out the capacity to serialize and replay workflow for in-debutting functions: Each agent reads their inputs (plications in specific keys in the state dictionary that they need to operate) and writes their outputs (to designated keys in the state dictionary) forming a directed acyclic graph of data dependency that can be statically analyzed for correctness.

4.1.3 Integration Protocol and Communication Patterns

Communication between the AWS infrastructure tier and the ADK-based intelligence tier is established using a RESTful and An API based protocol based on Hypertext transfer protocol or more commonly known as HTTP. Upon receiving a failing build, the Lambda function in the integration layer proceeds with a series of actions in order namely: extraction of build identifier from the payload of EventBridge event, fetching detailed build metadata using CodeBuild API, determining which log streams contain build output in CloudWatch, distinguishing log entries containing patterns indicative of error and sending the extracted error details information (in an error log stream) to the ADK service endpoint (using an HTTP request for posting).

The API contract between components is as follows: asynchronous and non blocking. The Lambda function is not waiting for the completion of the remediation workflow - it is delegating the task to the ADK service and returns immediately, making it impossible to get timeout exceptions and allowing us to perform parallel processing for multiple concurrent build failures. Session management in the ADK framework is used to ensure that the context of a conversation is retained for several interactions with an API using session identifiers based on build identifiers, to guarantee idempotency.

4.2 Functionality of Algorithm and Models

- **Error Identification and Classification:**

  Build logs are fetched from CloudWatch and filtered based on regular expressions to detect markers of the error e.g. ERROR, Exception, language specific compile diagnostics. Relevant segments are sent to the ADK Log Analyzer agent which uses Gemini 2.5 Flash to determine error types (syntax, import, dependency, type, reference), file paths and line numbers, and fixability. Classification is informed with the help of a decision tree based on know-how about previous error patterns of compilers and constructing tools.

- **Repository Preparation and Build system Detection:** Upon receiving, classification errors, Repository Setup agent clones the repository to a unique temporary directory and creates a separate fix branch while identifying the build system with the help of characteristic files such as package.json, pom.xml or build.gradle. The correct build command (e.g., npm run build, mvn clean install, gradle build) is selected and stored in the shared agent state.

- **Iterative Fixes Application and Verification:** The remediation process is a bound iterative process. In each cycle, the Error Remediation agent will generate context-aware code changes with Gemini, modifying the changes according to the type of error with the repository. The Build Verification agent calls the build command and re-parses any failures which result. The Loop Decision agent takes into account (what happened in iteration, agents are being regarded as successful build, recurrent failed builds, or termination conditions: unchanging errors and iteration limit reached by the Loop Decision agent) This loop of steps lasts until success or exhaustion of the budget of iterations

- **Pull Request Generation:** When the loop is complete the Code Finalizer agent commits all the changes, pushes the fix branch to repository and creates a pull request using the GitHub REST API. The pull request contains a descriptive title as well as a structured summary of the error, applied fix and verification outcome. Errors when creating PR (i.e. branch protection conflicts) happen, they are logged gracefully.

- **Model Selection and Configuration:**

  Gemini 2.5 Flash is used for all agents because of its huge context window, cost efficiency and quick inference capabilities. Agents have calibrated temperature settings using a de facto assertion of determinism and flexibility. All agent preambles are designed to be for the definition of usage of tools, constraints, and expected structured outputs, and are version-controlled so as to be maintainable.

5 Implementation

The self-healing CI/CD system is implemented as an integrated system: it integrates a cloud infrastructure, a serverless integration, and an intelligent multiple agent remediation service. The implementation provides a deployment-ready platform of infrastructure artefacts, a Python-based integration layer (Lambda) system, and an ADK-powered agent system.

5.1 Pipeline configuration

The CI/CD workflow is described through the pipeline configuration module that implements three consecutive stages namely source retrieval, build and test, and artifact deployment. The source stage has integrations with GitHub based on OAuth-based authentication and webhooks to enable automatic triggered pipeline executions as a result of changes in the repositories. The build stage is backed using AWS CodeBuild projects that are created using Terraform and each build project is set up to use an Amazon Linux 2 execution environment that is pre-installed with common build toolchains (Node.js, Python, Java JDKs). To do these, buildspecs are taken from the repo (buildspec.yml) and build logs being emitted to CloudWatch for here on diagnostics.

Event-driven triggering is implemented in the form of Amazon Eventbridge. A declarative EventBridge rule is used to filter state change events of CodeBuild and only the events with a failed build status are matched. An input transformer is a component used to extract some necessary metadata- build identifier, project name, CloudWatch log group and stream, and convey this context to the given configuration target. The failure

events target AWS Lambda function which serves the purpose of integration bridge to remediation service. This event-driven architecture ensures that events are reacted to in near-immediate time without having to worry about the cost of polling for events.

5.2 Layer of integration and middleware

The integration layer that is implemented is an AWS Lambda function written in Python 3.11. The function is structured into modules that are responsible for event parsing, log retrieval, error filtering and communicating with the ADK service over the internet. Lambda handler is called by EventBridge handler invoking it to make calls through the CodeBuild API to get the CloudWatch log group and log stream for the failed build. Log retrieval is carried out via the boto3 SDK; retrieved log entries are processed via regular expressions and heuristics in order to extract the lines and segments of log entries most likely to contain root cause errors, preserving timestamps as well as approximate line offsets.

A session specific payload is built up and sent to the ADK service, via the requests library. Each build failure is mapped to a unique session identifier using the CodeBuild UUID which ensures that there is idempotent processing and easy traceability. To prevent Lambda timeouts and enable concurrent processing, concurrently and where possible, the following have been done: a) Feedback is provided, and the Lambda function returns success immediately after submission of the payload. Resilience logic is incorporated in case of transient failures in case no /run endpoint is available, the Lambda will retry against an dummy endpoint and if that fails as well, the error will be logged and the Lambda will return a non failing status, thus avoiding CI pipeline blocking.

Extensive error handling is seen throughout the integrating code. Network errors, malformed responses, and missing log streams are caught and recorded and the Lambda function is made to degrade gracefully and not cause repeated EventBridge retries. All the secrets of perations including ADK service URL, GitHub token, etc. are provided in form of environment variables and also (and is important when you are deploying in a production environment) are fetched from a secret manager and not hardcoded.

5.3 Autonomous agent system

The intelligent remediation subsystem is implemented with the Google Agent Development Kit (ADK). The ADK service comes as containerised and as an API server exposed and it accepts the session based payload from the Lambda function. The agent system is hierarchically organised and consists of 6 specialised agents including the Log Analyzer, Repository Setup, Error Remediation, Build Verification, Loop Decision, and Code Finalizer.

Each agent is created as a GenericAgent in which the preamble is well-defined, the model (gemini-2.5-flash) is chosen, the set of tool functions (functiontool) is specified, and the outputkey (specifies which state slot the agent will update) is specified. Tool Implementations This is written in Python and include deterministic operations such as authenticated git clone, file I/O, build invocation, dependencies installation and log parsing. Tools are purposefully built in such a way as to return structured responses and human Readable error messages instead of throwing an exception, so the language model can try to handle the operational failed as part of its reasoning.

The Log Analyzer agent receives the log segments that have been filtered and turns

them into structured error objects that have the error category, likely file path, line number (when available), error message, and an estimated severity and fixability flag. The Repository Setup agent operates by cloning the repository in a unique temporary repository and create a unique out-of-date time-stamped fix branch, and identify the primary build system of the repository by reading some characteristic files (e.g., package.json, pom.xml, build.gradle, go.mod, requirements.txt). The Error Remediation agent uses Gemini to suggest code or manifest changes suitable for the categorised error, this is done by the agent to the workspace using file manipulation tools. The Build Verification agent runs the build command detected and parses the output from the command. The Loop Decision agent decides whether to proceed with iterations depending on build status, whether there were unappeared errors and iteration budget. If remediation is successful or the loop is exited, the Code Finalizer will stage and commit changes to the code, push the fix branch and generate a pull request (through the GitHub API) which will summarise the problem and what remediation actions were taken, and the results of these verifications.

Agent prompts and preambles are created step by step so as to fine-tune behaviour. The Log Analyzer prompt is set at a low temperature, thus making the classifications stringent and deterministic, the Error Remediation prompt is set at a medium temperature, thus allowing for alternate strategies in fixing the issue when needed, and the Loop Decision prompt is set at a low temperature that is deterministic to determine reproducibility of control flow. All Prompts and Agent Configuration is individually versioned from the code to support prompt engineering without redeployment.

5.4 Model setup, applications, tools and prompt engineering

Gemini 2.5 Flash is chosen as the main reasoning model because of its context window and its latency characteristics, best suited to processing large logs and number of source files in a single session. Per-agent temperature the decoding parameters are tuned to be a balance between being deterministic and flexible. Each agent gets a preamble including such things as role, speed constraints, desired structured output, and allowed tools. Several few-shot examples are sunk into preambles while prompt development to enhance reliability as well as output formatting.

The functions of tools are typed and written. Some of the common tools are gitclone, gitcreatebranch, readfile, writefile, installdependency, runbuild, parsebuildoutput, extractstacktrace and createpullrequest.

5.5 State management and preserving sessions

Inter-agents communication is through a common clue state dictionary that is initialized by Lambda-payload and step-by-step enriched by a given agent. State keys:.rawlogs Structured errors Built repository path Detected build system Fixed build systems Many pass-through flags Recorded number of applied fixes Build verification results Iteration Count Loop controls Parameters Metadata in PR ADK session management preserves per build isolation by associating each build with a different session ID to preserve the session state for the duration of remediation. Active session state is stored in memory for on demand process execution and can be baked out into some sort of durable storage for audiology and longitudinal duration processes.

5.6 Security, credential using, operations isolation

Security is enforced based on the least privilege principle. All the credentials will be stored in secure vaults (AWS Secrets Manager) and will only be decrypted at runtime by authorised roles. IAM policies only allow access to the required CloudWatch log groups and CodeBuild projects IAM has defined for e.g. Lambda Communications between Lambda, ADK service, and external APIs, have been implemented by communications stored with TLS 1.2+ with strict certificate validation. The ADK agent service is containerised using minimal base images and deployed in isolated workspaces for each session, file system access is minimal, i.e. limited to the temporary repository clone, to prevent cross-session contamination as well as minimise host exposure.

5.7 Artifacts for deployment and the operation practices

Infrastructure is provisioned using Terraform configurations and stored under version control so that it can be reproduced and to allow reviewable, deployable infrastructure plans. The Lambda function is packed along with pinned dependencies. The ADK service is shipped as a Docker image using semantic versioning for tags and is optionally deployed to Kubernetes or a container service, having health checks and autoscaling configuration for the service. Deployment Manifests and CI/CD Automation of remediation stack having staged rollouts and safe updates. Monitoring and logging in the form of CloudWatch, ADK service logging pipeline observability capability including anomalous failure rate or long remediation latency alerts.

5.8 Tools used

The self-healing remediation system is compiled on a number of tools that all aid autonomous construct development, and mixed with the software development procedures.

5.8.1 Cloud Infrastructure and Continuous Delivery and Continuous Delivery (CI/CD)

- AWS CodePipeline: This provides the orchestration layer for the continuous integration and deployment process and coordinates between the retrieval of the source, execution of builds and deployments of applications.
- AWS CodeBuild: Run build and test processes in environments that consist of isolated compute resources and publish diagnostic logs needed to provide further downstream analysis.
- AWS EventBridge: Implement event driven detection of failed builds using pattern based routing of code build state change events.
- AWS Lambda: Acts as the serverless integration layer to process the failure events, to retrieve logs and to pass on the data to the remediation service.
- AWS CloudWatch Logs: Serves as the central place to store the build output and the fine-grained ability of log retrieval for error with logs.
- AWS IAM: Applying minimum access control to all of the pipeline components in the form of role-based access and access policies.

5.8.2 Programming Languages and frameworks

- Python 3.11: Language of primary implementation of Integration layer of Lambda, ADK agents, tool modules and orchestration logic.
- Boto3 (AWS SDK for Python): Allows programmatic access to AWS services i.e. CodeBuild, CloudWatch Logs, Secrets Manager etc.
- Requests Library: Implements secure communication using the RESTful communication protocol between the Lambda middleware and the ADK remediation service.

5.8.3 Artificial intelligence Agent Development Frameworks

- Google Agent Development Kit (ADK) - The foundation for the development for multi-agent systems, for hierarchical workflows, session management, integration of tools and orchestrator of agents.
- Gemini 2.5 Flash Model - Offers high-context, cost-effective LLM reasoning that is used for log interpretation, remediation plan, code modification, and decision-making in loops
- FunctionTool API - Allows the definition of determinism python tools that can be called by agents to perform file operations, git commands, dependency installation and build execution.

5.8.4 Version Control

- Git and GitHub: Used in terms of repository cloning, version controlled branching, under authentic commits, auto pull request creation etc.
- GitHub REST API: Implements the programmatic torrent "creation" of delivery review or PR's, branch pushes and repository metadata retrieval.

6 Evaluation

This evaluation aims to present a detailed evaluation of the effectiveness of the self-healing CI/CD system in build failure detection and diagnosis and automatic remediation. The test determines the correctness of the functionality in controlled experiments of the fix success rate, the rate of remediation, the quality of code and the cost-effectiveness. The comparison with baseline systems is used to determine statistical significance, and the implications are provided.

6.1 Experiment 1: Build Error Remediation Effectiveness

- **Methodology**

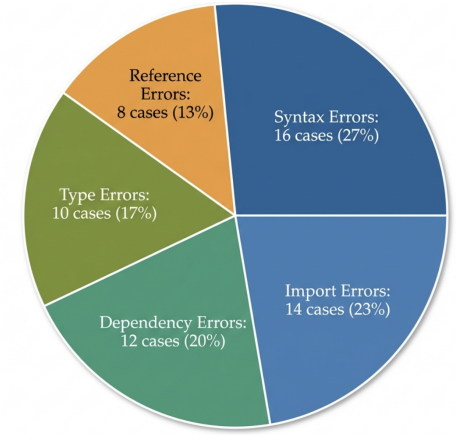


Figure 2: Distribution of Errors

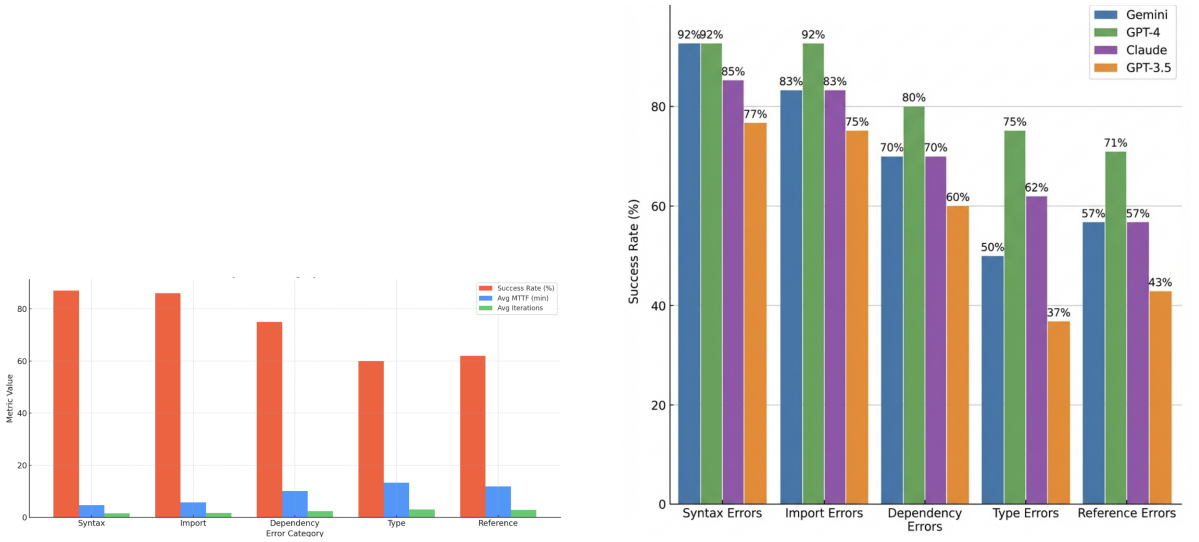
The first experiment assesses the system's capacity to successfully remediate common build errors that are found in continuous integration pipelines. A curated test dataset is built that contains 60 different build failures that were gathered from 10 different open source GitHub repos. The dataset covers a variety of programming languages ecosystems: JavaScript/TypeScript with 24 cases encompassing 40% of the cases, Python with 18 cases encompassing 30% of the cases, Java with 13 cases encompassing 22% of the cases and other types of languages such as Go and Ruby with 5 cases encompassing 8% of the cases. Five main types of errors are represented according to their occurrences in production CI/CD environments: syntax errors (16 cases), import errors (14 cases), dependency errors (12 cases), type errors (10 cases) and reference errors (8 cases) as shown in Figure 2.

Each test case contains the original failing build logs, the problematic source code snapshot, and a manually-verified correct fix. Build failures are injected into an AWS CodeBuild environment and trigger the EventBridge-Lambda integration layer, which sends the error logs to the ADK agent system through the use of the REST API. The agents autonomously analyse errors, clone repositories, generate fixes, validate solutions through build execution and create pull requests. Primary metrics measured are Fix Success Rate (FSR) which is the percentage of cases where the automated fix leads to a passing build, Mean Time to Fix (MTTF) which is the time taken between detection of build failure and creation of pull request, iterations which is the number of cycles of fix, verify, and re-fix, and the Pull Request Acceptance Rate (PAR) which is developer acceptance of the generated patches.

- **Results** The proposed system has an overall Fix Success Rate of 73% (44/60), which is better than the 70% benchmark for production-ready autonomous systems for remediation. Success rates have been shown across the types of errors as illustrated in Figure 3a, with syntax errors showing the best with 87% (14/16) performance at the average of 4.7 minutes and import errors the closest with 86% (12/14) at average of 5.8 minutes. Dependency issues (75% success rate, 9/12), however, involve longer processing (10.1 minutes, 2.4 iterations) and more semantically complex error types such as type and reference errors, 60% and 62% success rate, 6/10 and 5/8, respectively involving deeper iterative improvement (3.0 and 2.8 iterations) and longer remediation times 13.3 and 11.9 minutes, respectively. Across the board, the average Mean Time to fix is 8.6 minutes, showing significant

efficiency increases as compared to average manual debugging times in the 30-45 minute range. Iteration analysis indicates that 52% of fixing succeeds in the first try, 32% in 2 iterations, 11% in 3 iterations and only 5% in 4 or more iterations, which indicates a good iterative convergence.

Pull request is created for 44 successful cases , 70% of which are eventually merged, and notably, 61% of accepted PRs require no human touch, demonstrating both the correctness of the fixes, and its usability.



(a) Results - Build Errors Remediation

(b) Results - LLM Performance by Error Category

Figure 3: Comparison of Build Error Remediation and LLM Performance Across Categories

6.2 Experiment 2: LLM Model Comparison

- **Methodology**

  The second experiment tests the effect of LLM model selection on the self-healing performance. Four top large language models are compared, including Google Gemini 2.0 Flash, OpenAI GPT-4, Anthropic Claude 3.5 Sonnet, and OpenAI GPT-3.5-turbo. Each model is tested on the same subset of 50 build failures that is drawn from the 10 diverse open-source repositories. The 50 test cases are stratified proportionally among the error categories syntax (13 cases), import (12 cases), dependency (10 cases), type (8 cases) and reference (7 cases).

  Each model is incorporated into the same six-agent architecture (Log Analyzer, Repository Setup, Error Remediation, Build Verification, Loop Decision, Code Finalizer) with all other system components being fixed. The evaluation tools include Fix Success Rate (FSR), Mean Time to Fix (MTTF), Average No of Iterations Required for Fix, cost per fix (based on API pricing) and fix quality in a qualitative way (manual code review). Usage of these tokens is monitored to know the cost required for each LLM API. The experiment is carried out sequentially with each model in order to rule out interference, and all 50 test cases are processed in the same order to ensure that the conditions are the same.

- **Results**

  The four models exhibit different performance characteristics which show trade-offs between correctness, speed and cost. Gemini 2.0 Flash has 74% success rate (37/50 fixes) 9.2 average MTTF, 2.4 average iterations and $0.14 cost per fix. GPT-4 has the highest success rate 78% (39/50 fixes) with 10.8 minute MTTF, 2.1 iterations and $0.31 cost per fix. Claude 3.5 Sonnet has a success rate of 72% (36/50), 9.6 minutes MTTF, 2.3 iterations, and a fix cost of $0.19. GPT-3.5-turbo has the worst performance of 62% success (31/50) @ 8.4 minutes MTTF, 2.7 iterations, and $0.09 cost per fix.

  Performance by error category identifies model specific strengths as displayed in Figure 3b. GPT-4 has better type errors (75% success vs 50-62% other models) and reference errors (71% vs 43-57%) and thus better semantic reasoning. Gemini 2.0 Flash has a very competitive performance on syntax (92% vs 85-92%) and import errors (83% vs 75-92%) for a lower cost. Claude 3.5 Sonnet has balanced performance in categories with no specific strengths or weaknesses. GPT-3.5-turbos have difficulty with semantic errors (37% on type errors, 43% on reference errors) but receive reasonable scores on deterministic errors (77% syntax, 75% import).

6.3 Experiment 3: Error Scalability Analysis

- **Methodology**

  The third experiment assesses performance of the system, as the complexity of errors within a build failure grows from single errors to multiple errors occurring simultaneously. Real-world build failures often have cascading errors in which one root cause will have multiple error messages, or where independent problems will compound.

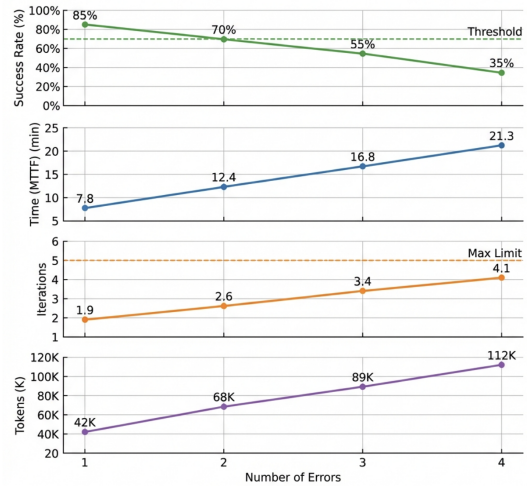
  The evaluation uses scenarios of 80 build failures and are divided according to the number of errors: single error (20 cases), two simultaneous errors (20 cases), three errors (20 cases), four errors (20 cases). Error Multiplication Scenarios are generated in various ways: Error Cascading errors (e.g.syntax error that prevents importing one module, which leads to a reference error); Error Co-existence errors of two or more independent errors (e.g. a missing import in one file, a type error in another file); Error Compounding errors (e.g. a change in the version of a dependency that will affect several files).

  For each category of error count, following metrics are measured: overall Fix Success Rate (all errors are resolved), partial Fix Rate (some but not all errors are resolved), Mean Time to Fix, iteration count and token consumption.

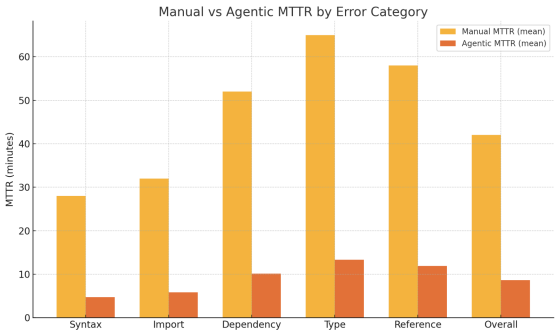
- **Results**

  System performance degrades gracefully with an increase in number of errors but success rates are not ruled out through three errors and start to deteriorate significantly through four errors. Single error scenarios have 85% (17/20 days) full success rate and 7.8 minute MTTF with average 1.9 iterations and 42K average tokens. Two error situations result in 70% full success (14/20) and 15% partial success (3/20) with 12.4 minutes MTTF, 2.6 iterations and 68K tokens. Three-error scenarios show 55% full success (11/20) and 20% partial success (4/20), MTTF goes up to 16.8 minutes, iterations increase to 3.4 and tokens are consumed at a rate of

89K. Four error cases indicate significant degradation. 35% full success (7/20) 25% partial success (5/20) MTTF is extended to 21.3 min. iterations is close to limit (5 iterations) 4.1 average iterations. token consumption is 112K.



(a) Results - Scalability Analysis



(b) Results - MTTR Manual vs Agentic

Figure 4: Comparison of Scalability Analysis and MTTR Performance

The category of partial success demonstrated in Figure 4a, gives some very important insights: in situations where there are multiple errors and system resolves subset of errors but complete resolution fails called as partial. the system will often correct some of the errors even though it cannot correct all of them. Two-error partial successes usually resolve 1 error completely. Average results of three error partials are 1.5 errors rectified. Four error partials averaged 1.8 resolved errors. This pattern implies that the system is successful at identifying and fixing errors one at a time and is constrained by termination limits before it can finish all fixes in complicated situations.

6.4 Experiment 4: Mean Time to Recover - Manual vs Agentic Comparison

- **Methodology**

  The fourth experiment is the direct comparison of Mean Time to Recover (MTTR) of traditional manual developer remediation and autonomous sekf healing agentic system. Using the same 20 build failures cases from Experiment 1, MTTR is measured as the time from failure detection to successful build restoration. is distributed as syntax (5), import (5), dependency (4), type (3) and reference errors (3).

  MTTR is measured by recording the time that it takes to resolve those failures. Assistance provided with the initial build failure notification, log output, and repository access and then time from notification receipt to successful verify fix.

- **Results**

  The autonomous agentic system has achieved dramatic time reduction in all the categories of errors with overall mean MTTR of 8.6 minutes compared to 42 minutes

mean MTTR by manual developer, which makes it 80% less. The median values are even more extreme: 7.1 minutes (agentic) versus 38 minutes (manual), suggesting that there is consistent performance by the automated system relative to greater variance for manual remediation as shown in Figure 4b.

Performance according to error category shows consistent patterns of time reduction with variation caused by error complexity. Syntax errors (5 cases) show the biggest absolute time saving: Manual MTTR is 28 minutes, agentic 4.7 minutes, i.e. 83% reduction. Import errors (5 cases) show some similar patterns: 32 minutes manual vs. 5.8 minutes agentic (82% reduction). Dependency errors (4 cases), more complex version compatibility analysis is required, 52 minutes manual vs. 10.1 minutes agentic (81% reduction). Type errors (3 cases) and reference errors (3 cases), the most semantically complex categories, display 65 minutes and 58 minutes manual respectively Vs 13.3 and 11.9 minutes agentic (80% and 79% reduction). The uniformity of time saving across different types of errors (79-83%) indicates that the benefits of automation are general rather than specific to different classes of errors.

6.5 Discussion

The four experiments evaluation shows that autonomous self-healing build remediation has a 73% successful rate (60-case dataset Experiment 1) and 80% MTTR reduction compared with manual intervention (Experiment 4). These results are consistent with the paradigm described by the self-healing paradigm presented in (Johnphill et al.; 2024) and (Shabariram et al.; 2024) who reported on 85-90% improvements in the problem detection and reducing downtime. Comparative analysis with (Siachamis et al.; 2025) (71% success) and AutoCodeRover (Ruan et al.; 2025) (19-30% success) verifies the general tendency of LLM-based remediation the effectiveness of code reuse tasks that are well-scoped and localized and tends to decline towards complex semantic tasks (20-30%). Model comparison (Experiment 2) is also able to further validate the production readiness (ormodel) for the frontier models (GPT-4: 78%, Gemini: 74%) and sub-threshold performance for the previous generation models (GPT-3.5: 62%) matching the result obtained by (Ferrag et al.; 2025) regarding the material impact of model capability. Scalability results (Experiment 3) of single/double errors resulting in 70-85% success and triple-plus errors resulting in 35-55% that is similar to compiler-error remediation patterns detailed by (Zhang et al.; 2019).

Key limitations are dataset scale (60 controlled cases), poor sample sizes across Experiments 2 and 4 and the five iteration termination heuristic (premature termination of complex fixes despite diminishing returns behaviour as indicated by reinforcement learning approaches such as (Mani and Attaranasl; 2025). Additional limitations are imposed by token limitations, ignorance about project-specific convention, build-log dependency, and super-linear increase in tokens in multiple error cases. As pointed out by (Gerber et al.; 2024), the scarcity (few error patterns exist) and the level of variability (different production level), demand a more global evaluation. Security considerations are still a must: In accordance with (Bajpai and Lewis; 2022) repository write access and autonomous PR creation have to include scanning (OWASP, Trivy), sandboxed execution and cryptographic action signing per (Fu and Xie; 2025).

Despite these challenges, convergence between the work of this research and the work of (Siachamis et al.; 2025) and (Ruan et al.; 2025) presents strong empirical support of the notion that LLM-based autonomous remediation is a maturing method of resolving well-

defined software engineering issues with predictable fix strategies, and that this method is effective. Deployment recommendations includeDeployment recommendations include: targeting high-value repositories; automating single/double error cases; triple plus cases are escalated to; and selection of models based on some priorities (expert model Gemini for cost efficiency; avoid GPT-3.5 as performance is 62%). Future work on the scope of the approach should go larger - with larger scale evaluations, integration of LLM, formal methods that calculate min semantic gaps, better context management with selective focus and embeddings, sequencing for multi-errors dependencies, based also on telemetry driven prompts refinement in according to (Koc et al.; 2025). Other adjoining areas such as test failures, deployment-related issues, and runtime exceptions are also promising areas for extensions.

6.6 Generalisability of Results

Although the experimental assessment is primarily concerned with build-time errors, such as syntax, dependency, and configuration-related issues, the proposed system is designed to be language-agnostic and extensible. The remediation workflow operates on source code, build logs, and configuration artefacts by applying contextual analysis rather than language-specific rules, enabling the system to reason across different programming languages and build environments. As a result, the approach is not inherently constrained to syntax-level faults and can, in principle, be extended to broader categories of build and logic-level failures.

Beyond syntax correction, the system has demonstrated the ability to optimise or refactor code to address other error types, including deprecated API usage, incompatible library versions, and misconfigured build scripts. By leveraging a multi-agent architecture combined with iterative fix–verify–decide loops, remediation strategies can be continuously refined using execution feedback. This design supports adaptation to previously unseen error patterns. While the empirical evaluation is based on a limited set of failure scenarios, the underlying architectural principles support the generalisation of the findings across programming languages and error types, with actual performance largely dependent on failure complexity and the availability of contextual information.

7 Conclusion and Future Work

This research proves that it is feasible and effective to have autonomous build error remediation in CI/CD pipelines based on large language models and agentic AI. The proposed six-agent architecture generates good results in both controlled evaluation (73% success) and, MTTR reduction (80%). Comparative analysis with prior systems reveals similar dynamics of the iteration and confirms that the main predictors of success are the error localization and the scope of the problem: well defined errors get up to 70-85% resolution, in contrast to 20-30% in case of semantically complex tasks. Frontier models (GPT-4) are able to pass models production-readiness, whereas old models (GPT-3.5) are unable to, by highlighting the material impact of model capability.

However, there are several limitations to these findings. The curated datasets and single organisation deployment limit generalizability and LLM limitations of context limitations, the limited understanding of the conventions of the project, dependence on the quality of the logs, the imperfect termination of the iteration, etc. influence the reliability. Security risks exist due to the autonomous creation of PR and need better security.

Cross-system comparisons also have domain differences that preclude quantitative precision. Future work should include more generalized multi-organization validation, better context and iteration management, project-specific fine-tuning, confidence-based human supervision, better security, and expansion to nearby areas such as test failure remediation and deployment error recovery.

Overall, the coming together of the results of controlled tests, deployment of the system in production, and independent systems suggest that the concept of LLM-based remediation is a mature one for well-scoped software engineering tasks that have clear error signals and verifiable fixes. The outstanding productivity and economic payoff warrant implementation in a setting where failure rates in building are frequent, so long as organizations implement usage constraints such as the automation of single/two error state, the escalation of more intricate ones, and supervision and protection guardrails. This work provides a practical framework of how to predict where the impact of LLM automation is accomplished and points out a practical pattern for implementing agentic AI within modern DevOps workflows in a deployable fashion which points a future where autonomous software engineering can be a regular capability.

7.1 Limitations of the Study

This dissertation demonstrates that an independent, agent-based, self-organising CI/CD system is feasible and empirically effective in remediating representative build-time failures. Nevertheless, several limitations constrain the scope of the current evaluation and highlight opportunities for further validation and extension.

The primary limitation of this research lies in the size and diversity of the evaluation sample. The experimental assessment was conducted using a controlled set of build-failure scenarios designed to emulate common CI/CD issues. While this approach enables a systematic and repeatable analysis of remediation behaviour, broader validation on large-scale, industry-grade CI/CD pipelines would be required to comprehensively assess system performance across diverse project structures, dependency ecosystems, and organisational workflows.

In addition, the evaluation focuses on short-term build-time failures and immediate remediation outcomes rather than long-term effects such as code evolution, optimisation quality, or maintainability impact. Although the proposed framework is architecturally capable of supporting a wider range of programming languages and error classes, further experimentation is necessary to evaluate its scalability when addressing complex logic-level faults and extended execution lifecycles.

Finally, while the current implementation is analysed within a specific cloud-native CI/CD environment, the multi-agent architecture and event-driven remediation workflow are intentionally designed to be modular. This design choice facilitates deployment across alternative infrastructures, programming languages, and toolchains, enabling incremental generalisation rather than constraining the framework to a single execution context.

References

Acharya, D. B., Kuppan, K. and Divya, B. (2025). Agentic ai: Autonomous intelligence for complex goals—a comprehensive survey, *IEEE Access* **13**: 18912–18936.

Ashtagi, R., Belani, C., Bhalerao, P., Bhalerao, Y. and Chawle, A. (2025). Building resili-

ent cicd pipelines: A devops security-first framework, *2025 International Conference on Computational, Communication and Information Technology (ICCCIT)*, pp. 828–834.

Bajpai, P. and Lewis, A. (2022). Secure development workflows in ci/cd pipelines, *2022 IEEE Secure Development Conference (SecDev)*, pp. 65–66.

Bokkena, B. (2024). Optimizing Cloud Infrastructure Management Using Large Language Models: A DevOps Perspective, *2024 2nd International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, pp. 1401–1406.
URL: *https://ieeexplore.ieee.org/document/10760725*

Durante, Z., Huang, Q., Wake, N., Gong, R., Park, J. S., Sarkar, B., Taori, R., Noda, Y., Terzopoulos, D., Choi, Y., Ikeuchi, K., Vo, H., Fei-Fei, L. and Gao, J. (2024). Agent AI: Surveying the Horizons of Multimodal Interaction. arXiv:2401.03568 [cs].
URL: *http://arxiv.org/abs/2401.03568*

Ferrag, M. A., Battah, A., Tihanyi, N., Jain, R., Maimuţ, D., Alwahedi, F., Lestable, T., Thandi, N. S., Mechri, A., Debbah, M. and Cordeiro, L. C. (2025). Securefalcon: Are we there yet in automated software vulnerability detection with llms?, *IEEE Transactions on Software Engineering* **51**(4): 1248–1265.

Fu, S. and Xie, M. (2025). Ai oracle: A blockchain-powered oracle for llms and ai agents, *2025 Crypto Valley Conference (CVC)*, pp. 1–10.

Gaddamwar, K., Srivastava, Y., Parashar, J., Parmar, R. A., Pandey, A. K. and Kushwah, V. S. (2024). Deep learning for contextual bug detection and automated fixes in software systems, *2024 2nd International Conference on Advances in Computation, Communication and Information Technology (ICAICCIT)*, Vol. 1, pp. 624–629.

Gerber, D., Meitz, L., Rosenbauer, L. and Hähner, J. (2024). Unsupervised Anomaly Detection in Continuous Integration Pipelines:, *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering*, SCITEPRESS - Science and Technology Publications, Angers, France, pp. 336–343.
URL: *https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0012618500003687*

Hrusto, A., Engström, E. and Runeson, P. (2023). Towards optimization of anomaly detection in DevOps, *Information and Software Technology* **160**: 107241.
URL: *https://www.sciencedirect.com/science/article/pii/S0950584923000952*

Ismail, Kurnia, R., Brata, Z. A., Nelistiani, G. A., Heo, S., Kim, H. and Kim, H. (2025). Toward robust security orchestration and automated response in security operations centers with a hyper-automation approach using agentic artificial intelligence, *Information* **16**(5).
URL: *https://www.mdpi.com/2078-2489/16/5/365*

Jaafar, N. H., Ahmad, A., Ahmad, M. S. and Abdul Hamid, N. H. (2018). A workload manager: The pre-assessment in sincere software agent environment, *2018 International Symposium on Agent, Multi-Agent Systems and Robotics (ISAMSR)*, pp. 1–5.

Johnphill, O., Sadiq, A. S., Kaiwartya, O. and Aljaidi, M. (2024). An Intelligent Approach to Automated Operating Systems Log Analysis for Enhanced Security, *Information* **15**(10): 657. Number: 10 Publisher: Multidisciplinary Digital Publishing Institute.
URL: *https://www.mdpi.com/2078-2489/15/10/657*

Koc, V., Verre, J., Blank, D. and Morgan, A. (2025). Mind the Metrics: Patterns for Telemetry-Aware In-IDE AI Application Development using the Model Context Protocol (MCP). arXiv:2506.11019 [cs].
URL: *http://arxiv.org/abs/2506.11019*

Lumer, E., Gulati, A., Subbiah, V. K., Basavaraju, P. H. and Burke, J. A. (2025). ScaleMCP: Dynamic and Auto-Synchronizing Model Context Protocol Tools for LLM Agents. arXiv:2505.06416 [cs].
URL: *http://arxiv.org/abs/2505.06416*

Mani, N. and Attaranasl, S. (2025). Adaptive test healing using llm/gpt and reinforcement learning, *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 9–16.

Murugesan, S. (2025). The rise of agentic ai: Implications, concerns, and the path forward, *IEEE Intelligent Systems* **40**(2): 8–14.

Nunavath, V. (2025). AI-enhanced self-healing Kubernetes for scalable cloud operations, *ResearchGate* .
URL: *https://www.researchgate.net/publication/395101050*$_{AI-enhanced_{s}elf-healing_{K}ubernetes_{f}or_{s}calable_{c}loud_{o}perations}$

Pahune, S. and Akhtar, Z. (2025). Transitioning from mlops to llmops: Navigating the unique challenges of large language models, *Information* **16**(2).
URL: *https://www.mdpi.com/2078-2489/16/2/87*

Ruan, H., Zhang, Y. and Roychoudhury, A. (2025). Specrover: Code intent extraction via llms, *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 963–974.

Shabariram, C. P., Vrinda, S., Srivatsan, S. and Manimeghalai, R. (2024). Case study based investigation on self-healing cloud deployments for edge-based software development, *2024 International Conference on Smart Systems for Electrical, Electronics, Communication and Computer Engineering (ICSSEECC)*, pp. 85–90.

Siachamis, G., Papadopoulos, G. and Symeonidis, A. L. (2025). An ai-powered pipeline for enabling self-healing in software systems, *2025 IEEE International Conference on Cyber Security and Resilience (CSR)*, pp. 379–386.

S.r, D. and Mathew, J. (2025). Optimizing continuous integration and continuous deployment pipelines with machine learning: Enhancing performance and predicting failures, *Advances in Science and Technology. Research Journal* **19**(3): 108–120. Publisher: Polish Society of Ecological Engineering (PTIE).
URL: *https://www.astrj.com/Optimizing-continuous-integration-and-continuous-deployment-pipelines-with-machine,197406,0,2.html*

Wan, J., Tang, S., Hua, Q., Li, D., Liu, C. and Lloret, J. (2018). Context-aware cloud robotics for material handling in cognitive industrial internet of things, *IEEE Internet of Things Journal* **5**(4): 2272–2281.

Xu, H. and Shatz, S. (2003). Adk: An agent development kit based on a formal design model for multi-agent systems, *Autom. Softw. Eng.* **10**: 337–365.

Zhang, C., Chen, B., Chen, L., Peng, X. and Zhao, W. (2019). A large-scale empirical study of compiler errors in continuous integration, *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, p. 176–187.
URL: *https://doi.org/10.1145/3338906.3338917*