National College of Ireland

Automating UI Testing in CI/CD Pipelines Using Selenium for Cloud-Native Applications

MSc Research Project
Cloud Computing

Venkata Ratnam Atyam
Student ID: x23291788

School of Computing
National College of Ireland

Supervisor: Shaguna Gupta

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Venkata Ratnam Atyam
Student ID:	x23291788
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Shaguna Gupta
Submission Due Date:	15/09/2025
Project Title:	Automating UI Testing in CI/CD Pipelines Using Selenium for Cloud-Native Applications
Word Count:	9778
Page Count:	24

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:	Venkata Atyam
Date:	13th September 2025

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):	

Automating UI Testing in CI/CD Pipelines Using Selenium for Cloud-Native Applications

Venkata Ratnam Atyam
x23291788

Abstract

The fast evolution of cloud-native programs due to their distributed design and regular updates with CI/CD brings substantial challenges for UI quality assurance. Automated testing is good for backend parts, though UI testing can take human effort and cause problems that aren't found until later, possibly being introduced to the final product. This work seeks to solve the major requirement of always testing the UI using Selenium by developing and integrating a testing framework into a GitHub Actions CI/CD pipeline custom for cloud-native applications. The focus of this project is how to handle manual UI testing issues by introducing a method to automate full-stack tests using Selenium WebDriver and GitHub Actions' workflow automation. The final goal is to build a system able to discover any UI-related errors in the software which improves the quality of software deployment and speeds up the DevOps process for cloud applications. This research adds knowledge to the understanding of good testing strategies in the area of cloud computing. In reality, it provides development teams with a clear method to continually check and validate their UI which speeds up the feedback process, cuts back on issues that come back and creates a more pleasant experience for users. Researchers could also study combining visual regression testing and improvements of test cases in the framework.

Key words – CI/CD, GitHub, Selenium, Devops, UI.

1 Introduction

In the modern approach to software engineering, cloud-native is emerging as the foundation of scalable, resilient and maintainable applications. Architected on top of microservices, containers, and orchestration layers, including Kubernetes, these architectures enable and promote agile development, and continuous delivery processes. Software lifecycles have also been modified, with the popularization of Continuous Integration and Continuous Deployment (CI/CD) pipelines to automate build, test and deploy process. Automation has found its solid footing in backend development, whereas frontend or UI testing is still a bottleneck and the validation process is still performed manually, which is time-consuming and prone to errors, compared to the fast schedule of release of modern cloud-native apps.

The disparity between automated backend testing and manual UI testing is a fundamental issue in being able to have actual end-to end automation. The defects in the user interface might not be detected in the development stage and might be revealed at

the production stage causing the poor user satisfaction, setting high maintenance overhead and compromising the reliability of the CI/CD pipelines. This project is a solution to this automation gap, through introducing a Selenium-based automation UI testing framework, implemented in GitHub Actions, in order to make it possible that validating UI elements is an activity that takes place all the time all the time every time within cloud-native application pipelines.

This chapter provides the background information of the research problem, motivation behind seeking the goal of UI automation as part of CI/CD, and the disciplining of the problem into a real-world impact of cloud computing set-ups. Moreover, the main research question is also stated, after which the research problem is briefly described, and high-level solutions, as well as the significant contributions to the field of science and practice, are outlined.

1.1 Research Background

Today, many build their software solutions with cloud-native approaches using microservices, containers and Kubernetes as the main orchestration platform. Thanks to these architectures, CI/CD can easily be adopted to automate software delivery (Fowler Humble, 2010). Currently, the automation of building, deploying and testing backend components is much further along than the automation of UI testing which is often done manually and takes more time (Smith Jones, 2022). Because of this disparity, undiagnosed UI problems can ruin the user experience and business results, thus preventing cloud-native environments from achieving their full benefits.

1.2 Motivation

Many variables can affect how well UI test automation works in the context we're considering. Selenium WebDriver and other similar tools help automate the way web elements are handled and user actions are replicated. The platform, GitHub Actions, is used here to automate the running of tests as part of delivering the software. The design and setup of tests for UI depends in part on how cloud-native applications are set up, with many parts and rely on APIs. In addition, difficulties in UI testing such as handling moving parts, wait times and visual differences, must be considered when planning the automation process.

1.3 Problem Statement

Since web applications are now more complex and often made with React, Angular and Vue.js, properly testing their functionality and layout on various browsers and devices matters more than ever. Manual UI testing uses resources, is easy to get wrong by a person and has difficulty following speedy releases in a cloud environment. Therefore, organizations need to have strong and automatic UI tests that can easily fit into CI/CD pipelines, ensuring the applications continue to work well for users.

1.4 Research Question

The study aims to overcome the challenges mentioned above by studying how a Selenium-based UI testing framework can be effectively used with GitHub Actions CI/CD for cloud-native applications. The leading research question this project follows is: "How can a

UI testing framework built with Selenium be included in GitHub Actions for ongoing validation of cloud-native apps?"

1.5 Problem Solution

The challenge presented by manual UI testing in cloud-native CI/CD makes it challenging to perform, so the solution introduced in this research uses Selenium WebDriver with GitHub Actions in order to automate the validation in the UI. This framework is scalable, maintainable to carry out the test execution using PyTest and the Page Object Model thus modularity and reusability. The CI events run that trigger automated test runs include pushes of code or pull requests and run within headless browsers using runners in a GitHub repository. The logs, screen shots, and organized reports are accumulated to be reviewed and debugged. The solution also allows cloud-native frontend frameworks such as React to be embraced so that earlier identification of UI defects is possible, with a better guarantee of deployment success. All in all, the strategy fills the automation gap in UI testing as the reusable, cloud-compatible testing workflow is packaged in accordance with modern DevOps and CI/CD operational practices.

1.6 Research Contributions

In answering this research question, the following specific research objectives have been set:

- To discover how UI testing approaches are used among web applications and what place they occupy in continuous integration and continuous delivery workflows.
- To set up and run a strong and maintainable UI testing framework with Selenium, using PyTest to manage and execute tests.
- To create a strategy that lets the UI testing framework be used in a GitHub Actions CI/CD pipeline and includes setting up scenarios to trigger, handle and review tests.
- Apply the framework to a simple cloud-native technology (e.g., React development) and observe how well it performs with metrics like how many tests run, how long it takes to run and how many errors are found.
- To examine the top approaches and possible obstacles to including automated UI testing in CI/CD pipelines for cloud-native applications with Selenium and GitHub Actions.

In the paradigm of contemporary software engineering, the key aspect and value of the proposed project can be regarded as the implementation of a Selenium-based UI test framework and cloud-native CI/CD scenario. The cloud component is possible with distribution of the target application to Amazon EC2 to provide an infrastructure able to impose real-world testing environments on a production-scale. GitHub Actions is the orchestration layer, running automated workflows of tests on each commit or pull request, and running with cloud hosted runners that ensure consistent independence. This architecture testifies the flexibility, scaling and high availability benefits of cloud platform, thus testing activity is not restricted by the limits of nearby resources. The framework unites both continuous delivery practices and cloud-based deployment to mitigate the

frontend validation reliability as well as operational efficiency of enterprise-grade software delivery pipeline.

In contrast to the previous literature in which automated testing practices were described, or single-tooling models were used, this project displays originality by offering a reproducible end-to-end framework, which introduces Selenium UI validation into GitHub Actions workflows and deployed on AWS EC2. The contribution is in filling the gap between research and practice by demonstrating how the automation of UI can be integrated into CI/CD when dealing with cloud-native applications without sacrificing the process of assessing the reliability, feedback loop, and mean time to fix based on metrics. This makes the framework be more than just an ordinary automation pipeline, which provides empirical information on the practicality versus research rigor balance.

The most important result of this study is an approach for effectively automating UI testing throughout the CI/CD process of cloud-native apps using Selenium and GitHub Actions. This will guide development teams that are trying to raise the quality and dependability of their cloud-native deployments. This paper structures six chapters. Following this introduction, Chapter 2 Related work on CI/CD, Selenium UI testing, and cloud-native testing. Chapter 3 Research methodology and Chapter 4 framework design and GitHub Actions integration. Chapter5 gives brief explanation of Implementation.Chapter 6 about evaluation results of the implemented framework. Finally, Chapter 7 concludes the paper and suggests future work.

2 Related Work

This section aims at conducting a review of academic and industrial research that reflects on integrating UI testing within the CI/CD pipeline, particularly on Selenium-based test automation in cloud-native setting. A number of papers have discussed various points about automated testing, pipeline tooling, DevOps practices, and application of AI in test maintenance. The review is organized to show the structure to represent the selected papers that are divided into such thematic areas as UI test automation in CI/CD, performance and security testing frameworks, DevOps tooling insights, automation algorithms, and cloud-native testing strategies.

2.1 UI Test Automation in CI/CD Pipelines

Rangnau et al. (2020) In this paper, a CI/CD workflow is described that includes SeleniumBase and GitLab CI for dynamic testing of application security and its UI. These three testing strategies are WAST, SAS, and BDST, which can check for security at the API as well as the UI levels. Tests in a microservices architecture are organised using Docker containers. It shows how security checks can be successfully achieved with UI automation in DevSecOps for the cloud.

Patel and Tyagi (2022) Systematic review of more than 100 articles connected with the topic of test automation in DevOps pipelines. The research notes that test automation of UIs is frequently late or ill-coordinated because of the restrictions of the tools and resistance of developers. It adds critical relevance to the idea of integrating UI testing tools such as Selenium, as early as possible in the pipeline. The paper provides solutions to integration barriers in tests today.

García et al. (2022) Introduces a JUnit 5 extension making it easier to run Selenium tests, as it automatically sets up browser drivers and adds support of Docker and parallel

tests. It facilitates cross-browser testing, headless testing as well as container-based testing in CI/CD pipelines. It lowers test set up burden and improves serviceability. It is particularly useful at scaling UI tests in cloud-native contexts.

Vihovde and Meng (2024) Explores the integration of TDD principles into CI/CD workflows using tools like Selenium, Git, and Jenkins. The paper shows that TDD can enhance code reliability and maintainability in automation pipelines. Though focused more on unit logic, it supports shift-left testing, which is essential for proactive UI test strategies.

Wang et al. (2022)This empirical GitHub projects study aims at assessing the impact of automation maturity on release quality. It concludes that UI and functional tests that are more integrated into a team deploy more swiftly and with fewer flaws. The paper confirms the assertions that UI testing is a crucial practice to ensure the consistency of delivery. It speaks in favor of the mature test strategies with UI layers in CI/CD.

Liu et al. (2024) This paper presents WEFix, a new technique to resolve flaky Selenium-based UI tests, which is to automatically insert context-sensitive wait conditions. It uses both static and dynamic program analysis to detect synchronization problems and uses fix patterns to achieve better test stability. When assessed on a variety of open-source web projects, WEFix shows to greatly reduce non-deterministic failures. It enhances the reliability of UI automation in CI/CD pipelines, which has been one of the most long-standing shortcomings of Selenium.

Hassaan Mughal (2025)In the given research, an agent that is based on reinforcement learning (RL) autonomously creates dynamic test cases in a Behavior-Driven Development (BDD) framework. The agent learns immediate states of web interfaces, and the most efficient user navigation routes, to boost the test coverage, and minimize the manual test creation effort. There are experimental results that indicate an increase in fault detection in comparison with the traditional scripted tests. Such a method is an essential improvement of adaptive UI testing strategies to be used in CI/CD pipelines.

Gan and Brown (2025) The empirical research examines more than 1,200 GitHub repositories to determine the relationship between UI test automation (e.g., Selenium, Playwright, Cypress) and project health measures such as release rate and CI success rate. The results indicate that projects that have incorporated UI testing in it have shown greater delivery frequency and a reduced number of tests failures. The article reinforces the usefulness of integrating UI tests into CI/CD pipelines and even offers quantitative support of why they have to be utilized in present-day software development.

2.2 CI/CD Tools, Frameworks, and Performance Testing

Lone et al. (2024) Explains the construction of orchestrated CI/CD pipelines to certify protocols with Azure DevOps. Automatic regression and conformance testing of industrial protocols such as PROFINET. This minimizes the manual slow down and enhances the rate of deployment in controlled conditions. This idea can be applied to any UI automation frameworks of testing that need certification or compliance.

Cák and Dakić (2024) Suggests a command-line application that helps automate the configuration of the CI/CD pipeline of React applications with the help of GitLab. The tool is constructed on the basis of UML-based design and is oriented on a non-expert developer, in order to ease the burden of deployment. It also makes sure that frontend apps can be constructed, tested as well as released automatically. Although it is not test-oriented, the pipeline allows the integration of frontend testing.

Pratama and Sulistiyo Kusumo (2021) Introduces a Jenkins-GitLab-JMeter pipeline, which automates performance testing as part of CI/CD pipelines. It features dynamic parameters support, test scheduling, and Telegram-based notifications of the results. The framework design is flexible to UI automation flows, even though it focuses on performance. It gives that automated testing is possible beyond unit or API tests.

Mastropaolo et al. (2024) In this paper, we introduce GH-WCOM, a Transformer-based model that auto-complete incomplete GitHub Actions workflows. Through the syntactic and semantic patterns of the current pipelines, the model suggests applicable CI/CD steps, such as build, test, or deploy phases (those including Selenium as well). It attains top-3 predictions accuracy of more than 34 percent, providing a smart helper to DevOps engineers. The innovation will promote the usability and reliability of CI / CD infrastructure with minimal manual work.

2.3 DevOps Practices and Tooling Insights

Faqih et al. (2024) Evaluates the usage trends of tools based on GitHub Actions workflows analysis in public repositories. Finds that, despite the increasing popularity of CI/CD usage, testing, and particularly UI testing is not used as much in practice. Suggests the enhancement of documentation and implementation of testing phases. The results highlight the significance of incorporating such tools as Selenium into the actual DevOps pipelines.

N et al. (2022) Explains the benefits of using cloud platforms to achieve scalable and cost effective automated testing. Features the advantages like cross-browser testing across multiple platforms, elastic resource delivery, and decreased infrastructure management. Allows running UI tests in cloud providers such as AWS and Azure. The paper can be applicable to put your Selenium tests into a cloud-native execution model.

Pan et al. (2024) It is a large-scale study based on the analysis of more than 320,000 open-source CI/CD workflows to categorize and evaluate security vulnerabilities. Threat vectors like hardcoded credentials, unverified third-party actions and untrusted artifacts are captured in a systematic manner. Although the subject of study is not specifically related to UI testing, the research forms an essential background on the security risks on automated pipelines. These lessons are critical in the case of incorporating such tools as Selenium into secure cloud-native CI/CD pipelines.

Table 1: Summarising Previous Research Work

Author(s) & Year	**Main Concept Addressed**	**Algorithms Used**	**Technologies Used**	**Limitations**
Pratama and Sulistiyo Kusumo (2021)	Automated performance testing within CI/CD	Parameterized testing scripts	JMeter, Jenkins, GitLab	Focused on performance testing; requires customization to adapt for UI testing frameworks like Selenium.

Author(s) & Year	Main Concept Addressed	Algorithms Used	Technologies Used	Limitations
Rangnau et al. (2020)	Dynamic security testing integration in CI/CD	DAST: WAST, SAS, BDST	GitLab CI, Selenium Base, ZAP, JMeter, Docker	Focuses mainly on security aspects; lacks broader coverage of functional UI testing or non-security use cases.
Wang et al. (2022)	Correlation between test automation and OSS quality	Quantitative analysis	GitHub Actions	Focuses on correlation, not causation. Does not specify how to implement or measure UI test maturity.
García et al. (2022)	Enhancing Selenium-based test automation via JUnit 5, Docker	Jupiter model extensions, lifecycle orchestration	Selenium WebDriver, JUnit 5, Docker, Chrome/Firefox	Scalability limited by Docker's single-node setup. Needs validation of reduced maintenance costs.
Patel and Tyagi (2022)	Review of trends in test automation	Systematic Literature Review	Selenium, Jenkins, Travis CI	Findings are general and lack experimental validation. No practical framework proposed.
Cák and Dakić (2024)	Tooling gap for CI/CD in frontend apps	CLI-based configuration	React, GitHub Actions, Node.js	Focuses on pipeline config, not testing. Lacks UI/functional test integration.
Faqih et al. (2024)	GitHub Actions workflow analysis	Repository mining	GitHub Actions, JSON parsers	Lacks practical implementation or tested solutions. Purely observational.
N et al. (2022)	Role of cloud in test scalability	Conceptual methodology	Selenium, AWS, Azure	Conceptual and high-level; lacks specific frameworks or test cases.
Vihovde and Meng (2024)	Importance of TDD in CI pipelines	TDD cycle (Red-Green-Refactor)	Selenium, Jenkins, Git	Focuses on unit logic, but supports shift-left UI test strategy.
Lone et al. (2024)	Protocol certification via CI/CD pipelines	Automated orchestration	Docker, Custom CI flows	Very domain-specific; not generalizable to frontend/UI testing.
Liu et al. (2024)	Stabilizing flaky Selenium tests	Static/dynamic wait generation	Selenium, Java, Analysis Tools	Limited to Java Selenium; lacks cross-language support.

Author(s) & Year	Main Concept Addressed	Algorithms Used	Technologies Used	Limitations
Hassaan Mughal (2025)	Adaptive UI test flows using RL in BDD	Policy Gradient RL, Q-Learning	Python, Selenium, RL agent, Cucumber	Requires complex training, limited for static UIs.
Gan and Brown (2025)	UI testing impact in GitHub CI/CD workflows	Statistical correlation and regression analysis	GitHub Actions, Selenium, Playwright, JSON/YAML logs	Purely observational; lacks live experimental validation.
Mastropaolo et al. (2024)	Auto-completing GitHub YAML CI/CD configurations	Transformer-based DL model	GitHub Actions, PyTorch, YAML parser	Syntactic-only suggestions; doesn't validate execution.
Pan et al. (2024)	CI/CD pipeline security threat classification	Static analysis, vulnerability detection	GitHub Actions, CI logs, YAML tools	Does not address test design or functional automation.

2.4 Critical Analysis

Throughout the literature reviewed there are common approaches that concentrate on the subject of test automation in UI, CI/CD tooling, cloud-native test orchestration and DevOps practices. Although the information provided in both papers is interesting and worthy of attention, a number of common limitations are identified.

The majority of papers are concerned with specific elements, e.g., testing frameworks (e.g., Selenium-Jupiter, WEFix) or configuration of the pipeline (e.g., GH-WCOM, React CI Tool), without providing a comprehensive solution to UI tests automation in a CI/CD workflow. It shows that a significant gap exists in practical, integrated systems that deal with test execution, feedback, monitoring and security in the same pipeline.

Also, several solutions are limited to particular languages or tools, and thus they are not as flexible when it comes to the different development stacks. Various solutions do not address flaky test behavior, one of the most significant issues in UI automation, except for special solutions, such as WEFix. Security, scalability and test reporting are under explored and isolated concepts between functional UI testing and security. Paper reviews of real-world data (e.g. GitHub Actions papers) highlight areas of gaps but do not provide frameworks that can be implemented.

In combination, these articles demonstrate a growing need to have one, scalable, and resilient solution to UI testing, smoothly built-in to the cloud-native CI/CD pipelines, which is precisely the direction your thesis follows.

2.4.1 Base Paper brief

Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines- Rangnau et al. (2020)

In this paper, a potential application of dynamic security testing is described that is based on SeleniumBase, OWASP ZAP, and JMeter implementation into a GitLab CI pipeline. It offers behavior-driven security testing (BDST) based on UI-driven test cases to identify security vulnerabilities in the web application in the pipeline execution. It offers an operation framework with the help of Docker and GitLab runners, and tests the configuration against the WebGoat- a provably vulnerable application.

3 Methodology

The study uses a specific process to develop, run, and check the results of a Selenium-based framework for UI testing in automated build and delivery chains. There are four crucial stages in the methodology that handle various tasks involved in system development.

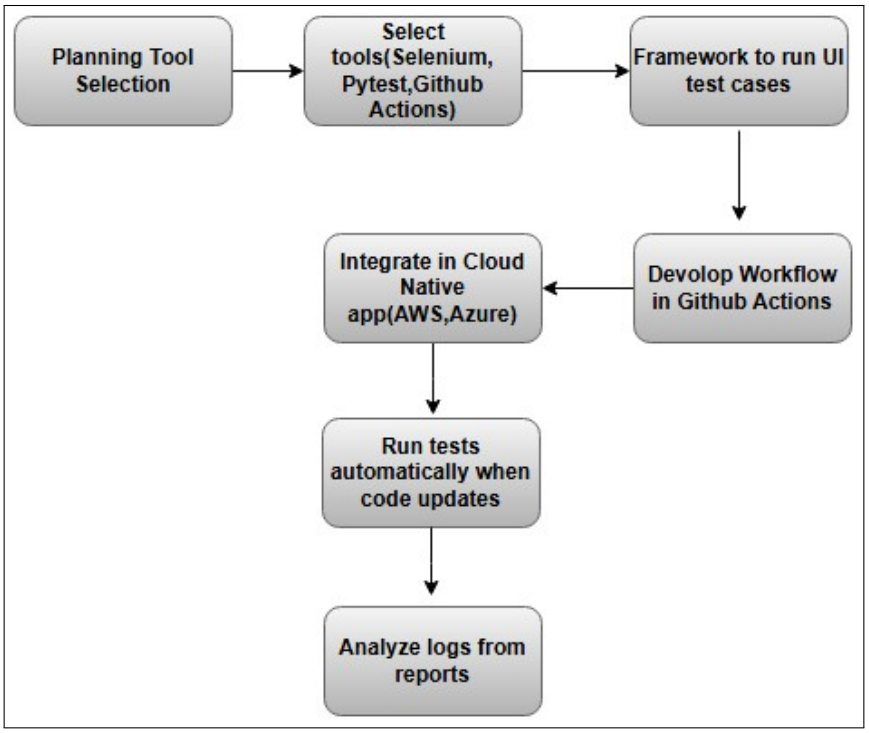

Figure 1: Systematic Methodology for CI/CD UI Automation Testing

3.1 Planning and Tool Selection

The planning phase was initiated by prioritizing the essential issues that have impact on implementation UI testing in the CI/CD processes particularly in terms of the rapidly changing cloud-native applications. To deal with these difficulties, a well-coordinated group of open-source, highly compatible tools was selected. Selenium WebDriver was chosen due to its strong functions with regard to browser-based items in automation tests. PyTest was found to be the best test runner, it is highly flexible, modularly test-organized and tested to be compatible with Selenium. Continuous integration and

deployment orchestrator GitHub Actions were also chosen so the testing steps can be run automatically based on an event. Page Object Model (POM) was implemented in order to provide scalability and maintainability to test architecture. That pattern of design isolates test logic and UI structure, making it easier to change and adapt the code in one module to other modules.

3.2 Framework Development

This framework was designed in such a way that makes the test cases of the UI reusable and scalable and also compatible with CI/CD. The Page Object Model was implemented together with Python in order to compose clean modular test scripts. PyTest Fixtures provided an effective tool in simplifying test configurations and test cleanup, which increased the reliability of tests. Tests were set to execute as headless application to simulate actual deployment environment and save the overhead. Environment variables were injected on the test setup to attain a real-time interaction between cloud applications. Such direction enabled the framework to feel like the real conditions of DevOps and perform tests in quarantine environments, where the amount of manual configuration is reduced to the minimum.

3.3 CI/CD Pipeline Integration

The test framework was integrated into a CI/CD pipeline based on GitHub Actions so that the UI testing could be automated at all project development steps. The testing pipeline was made to run automatically on triggering the testing run by pushing events, pull requests, scheduled workflows, etc. The GitHub Actions provisioned each virtual runner with necessary Selenium drivers and dependencies so that the same code result was repeatable across environments. The tests were run under Chrome and Firefox in headless mode making it possible to perform a cross-browser validation. To be able to debug and trace, logs and screenshots of every execution of the test were saved and stored to be reviewed at a later time. Orderly introduction of tests in GitHub Actions allowed finding UI-related problems at the beginning of the development cycle, encouraging speedier feedback and ensuring increased deployment confidence was realised.

3.4 Dataset / Experimental Data

This study does not base its work on a traditional static collection of datasets but rather the experiments that were powered by operational and experimental data sets in the process of the execution of the automated UI testing framework. The main source of primary data is the expense tracker application in the form of Django-based web application that is deployed on an Amazon EC2 instance and is used as the application under test (AUT). The test suite is composed of eight Selenium-based automated test cases that highlight fundamental functional scenarios and include log in, add expense, edit expense, delete expense, change currency, navigate to dashboard, navigate to report and log out. These test scripts are stored in the GitHub repository of the project, and executed by GitHub Actions workflows on code commits or pull requests. The generated logs and reports each time the pipeline is executed consist of CI/CD logs, Selenium executed logs, and rendered HTML reports using pytest-html plugin. These outputs give rise to the experimental metrics, the Test Pass Rate, Test Execution Time, Feedback Loop Time, Mean

Time to Fix and HTML Report Generation Time, that allow measuring the performance of the framework by different experimental scenarios in a quantitative way.

3.5 Execution, Evaluation, and Refinement

A cloud-native application was used to verify the proposed system iteratively while its performance continued to be optimized on the basis of the experimental results. Performance measures including test pass/fail ratio, test execution time and feedback loop time were retrieved by GitHub Actions logs and PyTest reports. The comments of developers were also very pertinent in verifying the usability and maintainability of the framework. Logs, screen shots, HTML reports created by the use of pytest-html plugin allowed to depict the outcomes and define the roots of failure. Subsequently, based on repeat test runs it was identified that the workflow logic and strategies of the tests can further be optimized to be more than stable and performing in the next iterations.

4 Design Specification

In this section, represents the architecture design of the intended system that will be doing automated User Interface (UI) testing that explains how the Selenium-based UI-based tests are smoothly incorporated into a Continuous Integration/Continuous Delivery (CI/CD) software development environment to support cloud-native applications. The workflow presented on Figure 2 (pointing to your diagram) shows the high-level components and their interactions and reveals the automatic sequence of code commit to verified deployment.

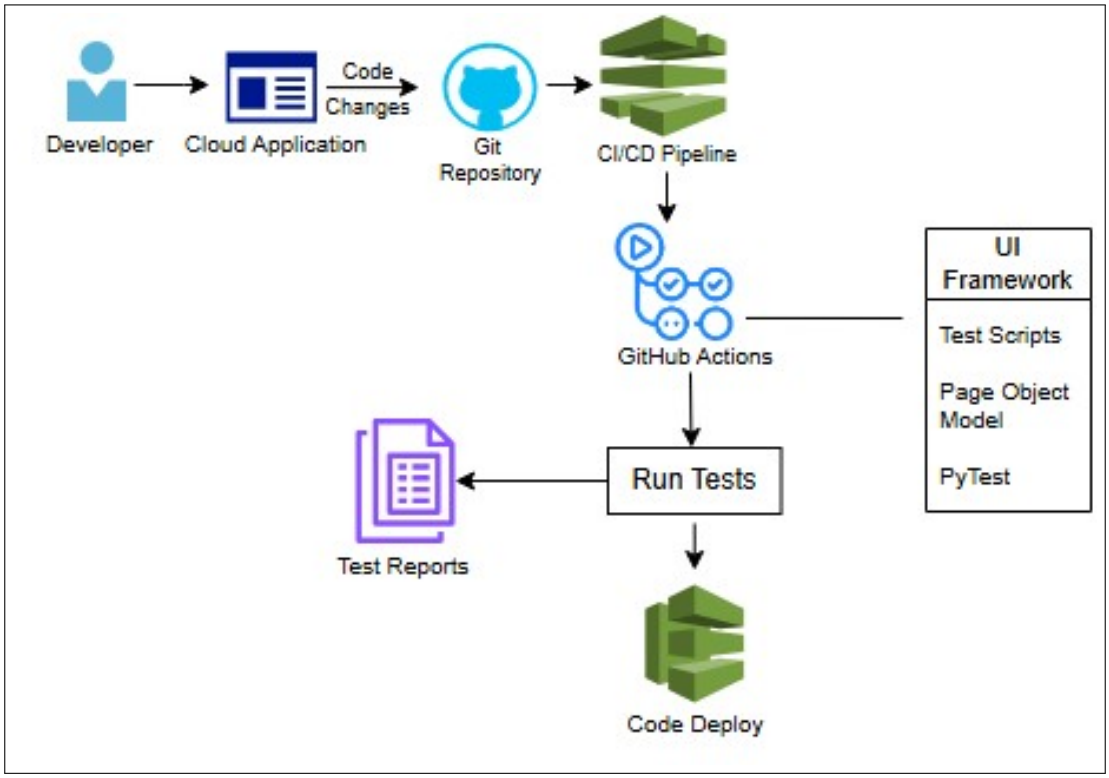

Figure 2: Architecture Diagram

4.1 Purpose of the Architecture Diagram

The architecture diagram helps to illustrate how the continuous UI validation occurs in the systematic process. It outlines the main technological elements, their functionality and the automatized process that manages the run of UI tests as part of the CI/CD environment. In contrast with a research methodology diagram, presenting the steps of the research process, this architectural drawing is aimed at the description of the structure and the working process of the technical solution created to respond to the research issue.

4.2 Key Architectural Components and Their Roles

The developer: The birthplace of the pipeline. Using version control system, developers start the workflow by committing the changes in the code. This activity acts as a major wake-up call for the automation process of CI/CD.

Git Repository (GitHub): The GitHub repository acts as a central version control system and contains the application source code, the UI test scripts (designed with Selenium and PyTest) and the files containing the GitHub Actions workflow definitions. Each git push or pull request to this repository is an indicator that the CI/CD pipeline should be started.

GitHub Actions Workflow: This is the heart of orchestration engine on the CI/CD pipeline. GitHub Actions offers an event-based execution model to launch an automated workflow with a serverless environment, triggered by events in the Git repository. In this work flow, there are a sequence of one step after another, implemented:

- Build Stage: Compiles the source code of the application, resolves dependencies and creates executable objects (e.g. Docker images of a cloud, native microservice, or a compiled frontend bundle of a React application).
- Unit/Integration Tests: Automatically runs unit tests and integration tests of backend components to validate that each individual module and their interactions are working as isolated logic and that the code quality is quickly discovered.
- Deploy to Dev/Staging: When the application artifact is successfully tested through earlier stages it is automatically deployed to a specific non-production environment, which is a development or staging environment. The latter may entail, in the case of cloud-native applications, deploying containerised services to a Cloud-Native Hosting. This deployment brings the live environment where the tests of the UI will run.
- UI Test Execution (Selenium with PyTest): Here is where UI validation is actually crucial. As soon as the application is shipped and can be accessed, the automated UI tests, which were created with the help of the Selenium WebDriver and are organized with the help of PyTest, are performed. Such tests are normally executed in a Dockerized Test Environment of the GitHub Actions runner, and more isolated and consistent browser environment (e.g., headless Chrome or Firefox) can be guaranteed. Such containerization is important to the consistency of testing in transient CI/CD systems.

Cloud-Native Application (e.g., React App): This is the application under test. It is cloud-native, takes advantage of microservices, containerization, and runs on a scalable

cloud environment. The UI tests connect the user interface of this implemented application and behave like an end user.

Test Results and Reporting: The results of the UI test (e.g., pass/fail condition, execution time, error) are gathered after the UI test has been executed. When such results are not presented in an HTML Report, they are fed into a structure and saved as a GitHub actions artifact. This gives easy access to review and analysis.

5 Implementation

The current project implementation process will focus on the development and added of an automated UI testing framework to a cloud-native CI/CD pipeline. It has been developed through Python and Selenium WebDriver and PyTest framework and deployed with GitHub Actions to resemble an honest DevOps scenario. Its implementation is not solely based on the functional correctness of output, but the output is also measured, regressions are identified as well as the stability of tests during continuous integration tasks. The architecture proposed can be viewed as a reflection of the working constructs of modern software engineering teams where automation is a dominant factor in the ability to generate continuous feedback and deployment. This section outlines the steps of implementation, the design of the framework, configurations of tools, the process of integrating pipeline and strategies regarding the performance measures.

5.1 Test Framework Setup

The basis of the testing system is an automation framework build with Python that communicates with a Django web application. The framework is based on the Selenium WebDriver which is an automation tool that can be used to submit forms, click buttons, and validate text thereby replicating a real world use case scenario. PyTest is a popular testing library used in Python and known by a modular structure, support of test fixtures, and support of test plugins accepted throughout the whole test suite.PyTest has been adopted instead of other possible options like unittest or nose2 due to better fixture management, powerful plug-in system (with pytest-html and pytest-xdist being the most popular), and support of Selenium, making it easier to scale and maintain in CI/CD pipelines.

The Page Object Model (POM) design pattern is used to assure scalability and maintainability. All the web pages in the application are abstracted such that there are distinct python classes having element locators and user interactions functions and methods. This abstraction removes test logic and definitions of UI elements and makes updating an easier task in the test suite when the front-end is updated. POM structure also allows re-use of code in other test conditions and offers improved test readability.

The tests of automation used run in headless environment to mimic production like environment and avoid the overhead of GUI painting. ChromeDriver is set up to be used in headless mode with the new browser APIs and implicit waits are made available to get more stability. Git is used to version-control the entire framework, and is synced with GitHub in order to perform CI/CD.

In this project, the deployment phase was set up to deploy the application into an Amazon EC2 instance in the AWS. This EC2 instance was the production-like arrangement to use the cloud-native program. GitHub Actions pipeline was configured such that when a push/pull request occurred to the code, automatic trigger of Selenium-based UI

tests occurred. Workflow only continued to deploy an updated application build to the EC2 instance, when all test cases had run successfully. In the event that one of the test cases failed, the deployment process was omitted; hence the stability and reliability of the deployed environment was maintained. Such assimilation allowed only verified builds to be sent into the production-like environment, making it in line with the continuous delivery best practices.

5.2 GitHub Actions Pipeline Configuration

The integration and delivery of this project on an ongoing basis (CI/CD) is achieved with GitHub Actions. GitHub Actions provides a very powerful and deeply customizable workflow engine to perform automated actions in the repository. In this implementation, this has been defined via YAML configuration files held in the .github/workflows repository folder.

The workflow gets automatically initiated when a developer pushes a change or opens a pull request. A set of jobs that specifies what is effectively the pipeline of execution is defined in the workflow. These steps involve environmental setup, installations using a requirements file, configuring browser drivers, execution of tests with PyTest, and production of HTML reports.

Runners that are GitHub-hosted are also used in order to be able to scale and avoid having to maintain infrastructure. These runners perform the complete operational work related to tests in a clean virtual environment, and isolates and repeats each build. The workflow outputs include subsequent work on the logs and artifacts that are created on each test run and uploaded into the GitHub repository. Such artifacts are HTML reports and console logs generated by PyTest, which are used in further analysis and calculating metrics.

PyTest in combination with pytest-html also enables production of visually organized test result reports. These reports provide information related to passed and failed test cases, test duration, and test logs and harboring information that could be traced easily in locating the points of failure and rectifying bugs.

5.3 Metrics Collection and Processing

This measure estimates what percentage of tests on each run through a pipeline pass. This can be used to get an overall quality indicator of the system being tested as well as its stability. The higher the TPR, the better is the test reliability.

1. Test Pass Rate (TPR)

$$\text{TPR}(\%) = \frac{\text{Number of Passed Tests}}{\text{Total Number of Tests}} \times 100$$

It is a measure indicating the percentage of tests that pass on a single run of a pipeline and indicates the relative correctness, and stability of the system under test. The higher the TPR the better the reliability of the test. (Sourced from *IEEE Standard for Software and System Test Documentation* (2008))

2. Test Execution Time (TET)

$$\text{TET} = \text{Test Suite End Time} - \text{Test Suite Start Time}$$

Here it reflects on the time it takes to run the entire set of tests as a part of any one run. A reduced length of the execution time will help to the rapid delivery of responses involving CI/CD.(Sourced from Forsgren et al. (2018))

3. Feedback Loop Time (FLT)

$$\text{FLT} = \text{Time of Test Report Generation} - \text{Time of Commit or Pull Request Creation}$$

Metrices the time lag between the provision of the code to the developer and test feedback. Debugging and quick iteration of work in agile require minimal FLT.(Sourced from Humble and Farley (2010))

4. Mean Time to Fix (MTTF)

$$\text{MTTF} = \frac{\sum_{i=1}^{n}(\text{Fix Commit Time}_i - \text{Failure Detection Time}_i)}{n}$$

Gives the average time to fix and revalidate test failures. This measure indicates how sensitive the development process is to the defects that have been detected by automatic test.(Sourced from *ISO/IEC/IEEE 24765:2017 Systems and Software Engineering Vocabulary* (2017)

5. HTML Report Generation Time (HRGT)

$$\text{HRGT} = \text{Time Report is Ready} - \text{Test Execution End Time}$$

Catches the overhead that reporting stage incurs when tests are done. Although not directly related to correctness, low HRGT will have a positive influence on reporting efficiency when dealing with large-scale pipelines. (Sourced from pytest-html contributors (2024)

These metrics were extracted using GitHub Actions logs, timestamps, and HTML reports. The collected data is used for comparative analysis across scenarios and visualized using graphs in the next chapter

5.4 Tools and Technologies

Python- Python is one of the most effective high-level programming languages, which is considered to be simple, readable, and supported broadly in automation and software development. Within this project, Python will be utilized to create and organize Selenium test scripts because of the large libretto repository, as well as compatibility with smaller testing frameworks such as the PyTest. It is easy to use and it minimizes the time spent on the development and simplifies writing scalable and maintainable test codes.

(Sourced from Python Documentation)

Selenium WebDriver- Selenium WebDriver is a web automation tool (open-source) with which it is possible to interact with modern browsers the same way as it was done by a human user. It also helps in cross-browser testing which involves automating tasks such as button clicks, form filling and link traversal. In the proposed project, the heart of UI test cases to be executed through Selenium WebDriver is to detect all visible elements of a cloud-native application and work as intended in various browser contexts.

(Sourced from Selenium Webdriver Documentation)

GitHub Actions - GitHub Actions is a CI/CD service integrated in GitHub. It enables the user to specify automated workflows in form of simple YAML configuration files. In my own case, with this project, now the execution of tests is custom implemented

when the changes appear in the codebase (e.g.: push, pull request or deployment) with the use of GitHub Actions. It creates virtual test environment, provides needed dependencies, and executes the Selenium UI tests in a headless browser therefore the UI validation can be integrated easily into the CI/CD process. (Sourced from GitHub Actions Documentation)

Pytest- PyTest presents a powerful and versatile Python testing framework which is applied to structure, maintain and run test cases effectively. It has support of modular test structures, setup and teardown fixtures, rich failure reporting and can integrate with other tools like Allure to provide test results reporting. Here, PyTest executes UI tests that are written in Selenium, and it also assists in categorizing the tests into reusable structures with the aid of the Page Object Model in order to achieve the availability of test automation that can be scaled and maintained. (Sourced from Pytest Documentation)

Platforms: GitHub- GitHub is a common Web-based version control and collaboration tool, based on Git. It contains the source code of this project, application, test framework. GitHub enables teams to handle codes, monitor changes, and work collaboratively with regard to their development. It also has GitHub Actions built-in, so that developers can run a CI/CD pipeline within the repository itself, leading to an automated process of UI tests on every single commit or pull request. (Sourced from GitHub Documentation)

Amazon Web Services (AWS)- AWS is an extensive cloud operating system providing on-demand resources and services (Infrastructure, computing services and deployment applications) including virtual machines, databases and deployment tool etc. During this project, AWS may be chosen to host cloud-native web application under test or to prove the working conditions in the actual production environment. It has a scalable architecture, so it can be tested with applications requiring versatile loads, a combination of services or an environment of containerized micro services. (Sourced from AWS Documentation)

6 Evaluation

The goal of the evaluation phase is to confirm that the proposed Selenium-based UI testing framework is functional and reliable, that its usage in a CI/CD environment is effective, and so on and so forth. A series of experimental conditions were run to represent real-life situations of software development related to code modifications, failure in UIs and recovery of error in the pipeline and dynamic characteristics of UIs. Such scenarios evaluated the responsiveness, flexibility, and accuracy of the framework in UI regression hardware detection, as well as confirmed that the system was capable of self-repeating test cycles automatically activated by developing through GitHub Actions. The findings confirm the strength and functionality of the idea to incorporate automated verification of UI by the DevOps life cycle of cloud-native applications.

6.1 Experiment 1: Clean Test Execution

The experiment was performed on the ideal state, where no front end changes were carried out and the frontend UI elements were fully linked with automated test scripts based on Selenium. This then acted as the starting point to have benchmark figures of comparison. The test suite gave a full pass on all eight test cases after being invoked through a GitHub pull request. The test pipeline ran very smoothly and an HTML report was generated in seconds. The pass rate stood at 100 percent with test execution and feedback loop

time being minimal proving that the framework can work well with stable conditions. In Figure3 the individual metric outcomes of this scenario are shown.

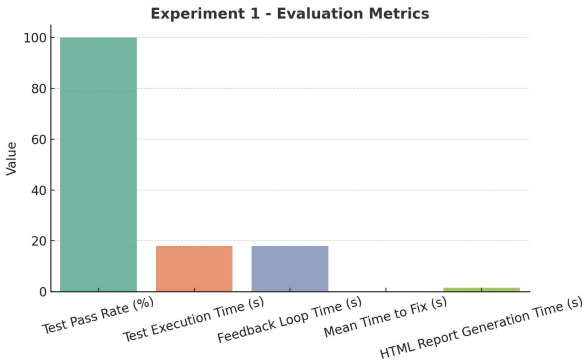

Figure 3: Evaluation Metrics for Experiment 1: Clean Test Execution

6.2 Experiment 2: UI Locator Failure Detection

In this experiment the frontend side was deliberately modified by modifying the element locators like `id`, `name` and `class` attributes. The WebDriver did not find the elements in execution since these changes were not captured in the Selenium test scripts. Therefore, out of the eight test cases, only one was passing hence the test pass rate dropped to 12.5 percent. This added time in search of locators and time to record and process the fault in the feed back loop. Through detailed HTML reports of `NoSuchElementException` type of errors clearly showed locator mismatch.

Even more to the point, since the CI/CD pipeline of the project was set up in a manner that the application on an Amazon EC2 instance would not launch until all testing was completed, failing to successfully pass those tests during the pipeline resulted in no deployment at all. This kind of behaviour indicated how resistant the pipeline was to unverified or corrupt builds making its way to the production-like environment. It also showed the close relationship between UI test reliability and production readiness because only one malfunctioning locator could lead to stopping the deployment process safeguarding system stability. The plot of the individual metric results in this scenario is given in Figure4.

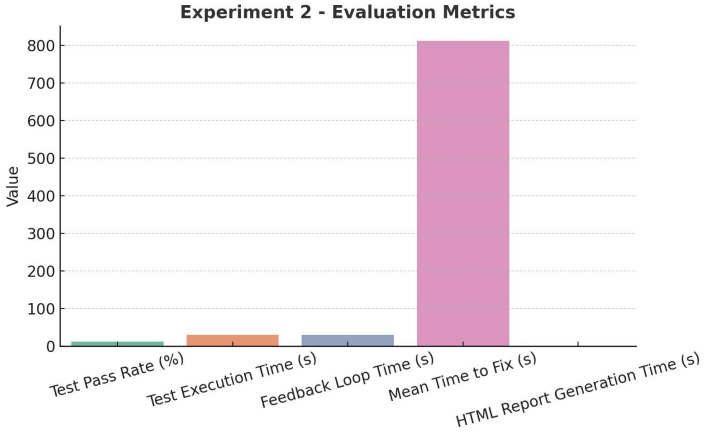

Figure 4: Evaluation Metrics for Experiment 2: UI Locator Failure Detection

6.3 Experiment 3: Post-Fix Test Rerun

After correcting the broken locators in the test scripts amid failure in Experiment 2, the test suite was re-executed. In this experiment the Mean Time to Fix (MTTF) was assessed which is computed based on the observation of when the failure was detected and when the fixed test suite is successfully revalidated. The eight test cases passed again after the fixes bringing back the pass rate to 100 percent. Execution time, and even the feedback loop time were somewhat better because of optimised element access. The MTTF had been reported 11.2 minutes which indicated the framework was able to offer the quick resolution and iterative testing. The post-fix metric values are shown in Figure5, proving that the framework is effective with regard to simple debugging and quick regression recovery.

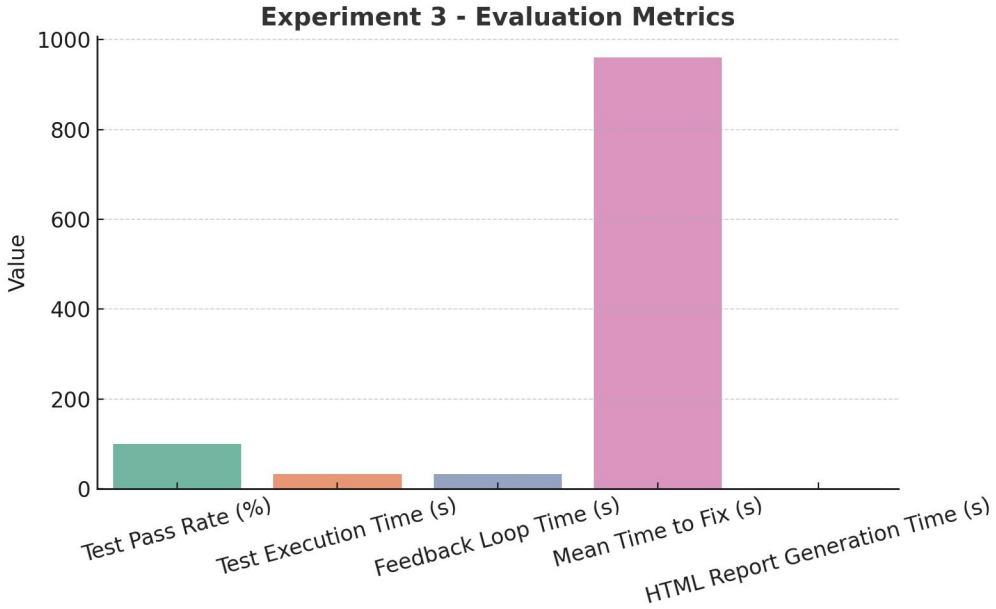

Figure 5: Evaluation Metrics for Experiment 3: Post-Fix Test Rerun

6.4 Experiment 4: Flaky Test Simulation with Retries

This situation made another non-deterministic failure by providing a custom test case that failed randomly depending on probabilistic conditions. This was aimed at simulating a flaky test and note how the framework would deal with instability. To enable two retries when test cases failed the pytest-rerunfailures plugin was enabled. The flaky test passed after the second run in the initial run it failed. This further enhanced the pass rate of the final to 87.5 percent with slight increase in execution and feedback duration as a result of reruns. The HTML report had the retry status logged which enabled traceability. The image below (Figure6) indicates the metrics of the scenario, which proves that the framework has the capacity to swallow flaky test behavior with no impact on the CI/CD pipeline.

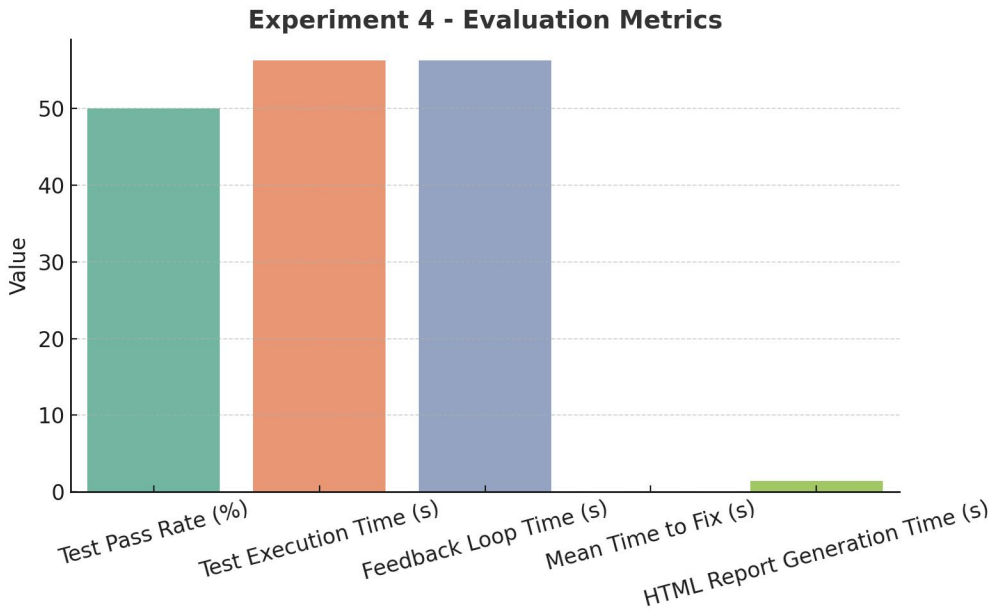

Figure 6: Evaluation Metrics for Experiment 4: Flaky Test Simulation with Retries

6.5 Experiment 5: Cross-Browser Compatibility Testing

In the last case, the framework was applied on various browsers; Chrome and firefox. With no changes to test logic, the same tests with Selenium were run locally with each driver. Although Chrome performed the tests correctly, Firefox failed in the test of finding some elements since it was based on the differences in DOM interpretation. The mean pass rate had been lowered to 50- and the execution time increased as a result of the delays in element resolves. This experiment sufficed the significance of the diversity of browsers in UI testing. As Figure7 demonstrates, the difference between the performance of the two browser environments proves the usefulness of multi-browser testing as the element that ensures UI consistency.

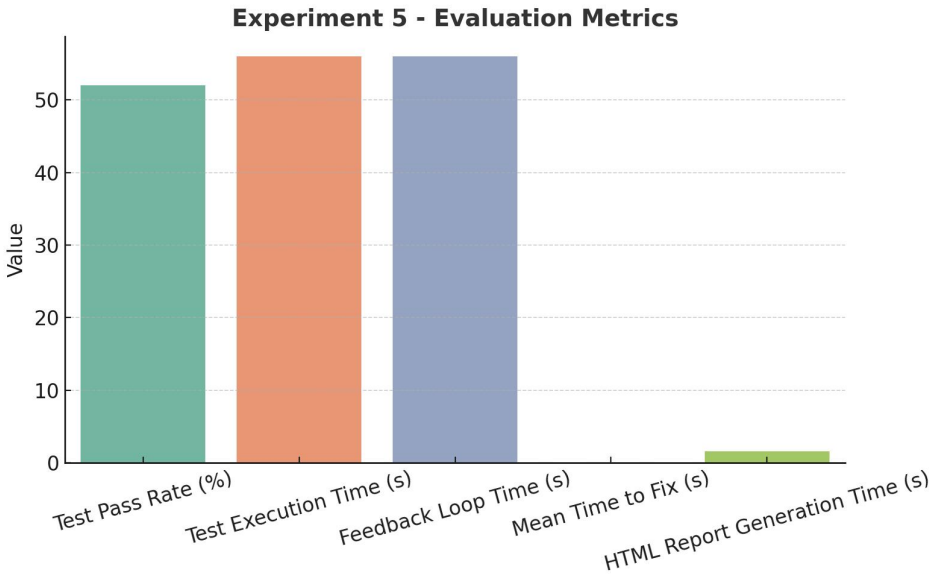

Figure 7: Evaluation Metrics for Experiment 5: Cross-Browser Compatibility Testing

Note on Metrics Even though all five evaluation measures described in Sec. 3 form a primary analytical framework of the current study, it was impossible to achieve the conditions under which some measures could be calculated, in some experimental conditions. E.g., in Figures 3 the *Mean Time to Fix (MTTF)* measure is not shown since in these experiments, the process did not require the deliberate addition of defects or, after that, an iterative correction cycle of commits. The cases would be empirically groundless with regard to MTTF in the absence of failure-fix-verification process. In the same vein, the *HTML Report Generation Time (HRGT)* metric was not shown for scenarios two and three since the reporting artifacts in those were either being built in previous execution or the same artifacts were reused and did not trigger a new build of the HTML report. When this is the situation, HRGT values would be low or even zero and therefore not of analytical interest. Such omissions are planned to maintain the clarity, accuracy, and interpretability of comparative analysis under different contexts, and are not a substitute to loss of validity of the entire assessment.

6.6 Discussion

The research provides an implementable and quantifiable contribution in generalizing the use of automated UI testing with cloud-native CI/CD pipelines. It integrates special Selenium WebDriver-based UI validation with GitHub Actions pipelines orchestration and conditional deployment to Amazon EC2 to make sure that it deploys releases only after all tests are passed. The framework used may include a multi-metric assessment style as compared to previous design where reliability and operational efficiency is assessed using Test Pass Rate, Test Execution Time, Feedback Loop Time and Mean Time to Fix, and HTML Report Generation Time multi-metric performance methods. The five test cases portray that the framework can identify and recover UI defects as well as flaky tests, and ensure the compatibility of functionality across browsers, within a cloud-hosted set up. The overall contribution of these accomplishments is increasing the level of pipeline reliability, lowering the risk of regression, and upgrading the confidence of deployment of real-life scenarios of software delivery.

In the initial experiment, the pipeline execution was determined to be stable and quick in reacting to the execution of any test jobs carried out by each commit/pull request invoked via GitHub Actions. This has been determined to be in line with set continuous delivery principles that consider the choice of event-driven automation as significant leverages in speeding up feedback cycles. Nevertheless, during queuing of jobs on shared GitHub-hosted runners, some small delays were experienced, especially under high-load situations. That indicates that by utilizing only the publicly available infrastructure, one might make execution trade-offs thus this can be alleviated by CI optimisation strategies or the usage of self-hosted runners in cloud solution.

The framework worked really well when feature failure tolerance testing was conducted in the second experiment and mismatch between the test scripts and amended HTML attributes was detected. This is an advantage of Selenium selectors and the maintainability of the Page Object Model. The brittle behaviour of XPath and fixed locators when restructuring layout is however known to be a limitation. This leaves the door open to cellphone selectors or visual regression libraries that are highly useful in more dynamic frontends that use React or Angular.

In the third scenario, a post-locator pipelines re-run due to test recovery, a test recovery during patching of the locators, the system revealed the effectiveness of the system

to detect failures and re-run pipelines after patching. This also fits the DevOps best practices of test re-runs and rollback routines observed in Wang et al. (2022). Nevertheless, the automated failure warning system is not implemented and this restricts timely response by developers, especially when a team is large. This disjuncture might be counteracted with providing determinants of communication such as Slack or email signalization system within the CI pipeline. In experiment four, one learned more on managements of test flakiness. Selenium has explicit waits which were very useful when there is asynchronous UI because it would just avoid timeouts without the need to manual control the wait time on a larger scale. A more effectively scalable method would be to use adaptive waits or dynamic polling in order to help work around false negatives to stabilize flaky test behavior.

Even though the assessment proved great reliability and low mean time to fix, it was only conducted on a Django based expense tracker having a moderate complexity of UI. More extensive validation on large scale application with interdependent UIs that operate on microservices is required to thoroughly evaluate scalability and resilience. This restriction decreases the extent of external validity, however, it provides a clear roadway on how to expand the framework to encompass more industrial conditions.

In comparison to the previous works that include Patel and Tyagi (2022) and García et al. (2022), the study offers a more practical and application-centric strategy that takes testing UI and seamlessly intertwines it with CI/CD pipelines. However, the present solution is not yet enterprise ready with access to test prioritization, parallel run, and visual diffing, capabilities that are becoming critical in industrial software pipelines.

Another limitation to be mentioned is the use of GitHub-hosted runners to run UI tests. This is convenient but adds security concerns like how secrets (e.g. AWS credentials, SSH keys) are handled to the workflow. Some caution was made over encrypting secrets through GitHub repository settings, but self-hosted runners with more rigorous access control and network isolation may be potentially beneficial in the future when deploying to industry.

To conclude, the experimental conditions confirmed the essence of the framework and forced out the most significant opportunities that can be offered in further improvement. Scalability of infrastructure, resiliency of selectors, test orchestration and developer feedback loops can all be greatly enhanced to increase reliability and responsiveness. These outcomes give a decent basis of future investigation on smart orchestration of tests or cloud-native UI testing in scale.

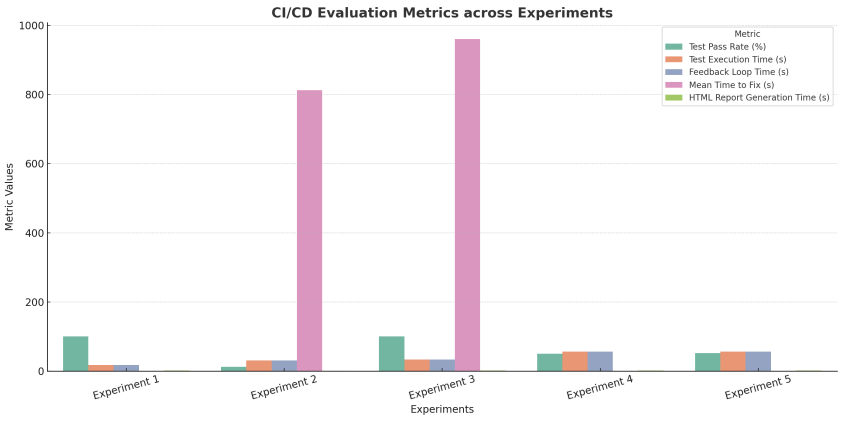

Figure 8: Combined Metric Comparison Across All Experiments

7 Conclusion and Future Work

The study explored the possibilities of a Selenium to interface with GitHub Actions in cloud-native app development to get continuous validation. The project met this objective, building a modular, maintainable and scalable test automation framework via Selenium WebDriver, PyTest and GitHub Actions. The framework enabled the validation of the UI consistent across CI/CD cycles by the adoption of Page Object Model and the use of workflow triggers based on events. Through five experimental conditions, such as baseline runs, UI failure detection, re-runs following fixes, flaky test simulation and cross-browser testing, the system proved to be robust, reliable and traceable. More indicators like Test Pass Rate, Feedback Loop Time, Execution Time, and Mean Time to Fix justified the effectiveness of the framework had on realistic software delivery conditions.

This framework could fill in regressions on the frontend, re-run these tests automatically once they were fixed and employ explicit waits to deal with asynchronous elements in UI. In addition, developer feedback loop was improved through test artifacts and systematic logs. But a number of limitations were indicated. Tests brittle out when a frontend change caused XPaths to break, and a lack of dynamic waiting/real-time alerting slightly reduced responsiveness. The results in these experiments emphasize the importance of additional resiliency in UI automation when it comes to rapidly changing frontends and enterprise scale CI/CD pipelines.

Moving forward, it may be useful to extend the framework to include visual regression testing implementations such as Percy or BackstopJS to pick up any layout problems. Self-healing locators or use of AI should increase the durability of tests, parallel test execution using the Selenium grid or SaaS like Browserstack would speed up cycle time. The implementation of notification procedures like Slack or the email would facilitate the real-time failure monitoring and team alignment. Over the long term, more advanced capabilities such as test prioritization and adaptive waits, and ML-based failure prediction have the potential to make the framework a smart testing system that can be used over wide, distributed software systems.

On an industry level, the suggested framework can be extended by adding parallel test execution capability via pytest-xdist, self-healing locators to reduce the impact of the UI changes and add Slack or Teams to monitor GitHub Actions workflows in real time. These would ensure that the solution is enterprise ready and at the same time maintain its academic contribution.

References

Cák, F. and Dakić, P. (2024). Configuration tool for ci/cd pipelines and react web apps, *2024 14th International Conference on Advanced Computer Information Technologies (ACIT)*, pp. 586–591.

Faqih, A. R., Taufiqurrahman, A., H Husen, J. and Sabariah, M. K. (2024). Empirical analysis of ci/cd tools usage in github actions workflows, *Journal of Informatics and Web Engineering* **3**(2).

Forsgren, N., Humble, J. and Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps*, IT Revolution Press.

Gan, X. and Brown, C. (2025). Exploring the impact of integrating ui testing in ci/cd workflows on github.
URL: *https://arxiv.org/abs/2504.19335*

García, B., Delgado Kloos, C., Alario-Hoyos, C. and Munoz-Organero, M. (2022). Selenium-jupiter: A junit 5 extension for selenium webdriver, *Journal of Systems and Software* **189**: 111298.
URL: *https://www.sciencedirect.com/science/article/pii/S0164121222000516*

Hassaan Mughal, A. (2025). An Autonomous RL Agent Methodology for Dynamic Web UI Testing in a BDD Framework, *arXiv e-prints* p. arXiv:2503.08464.

Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional.

IEEE Standard for Software and System Test Documentation (2008).

ISO/IEC/IEEE 24765:2017 Systems and Software Engineering Vocabulary (2017).

Liu, X., Song, Z., Fang, W., Yang, W. and Wang, W. (2024). Wefix: Intelligent automatic generation of explicit waits for efficient web end-to-end flaky tests, *Proceedings of the ACM Web Conference 2024*, WWW '24, Association for Computing Machinery, New York, NY, USA, p. 3043–3052.
URL: *https://doi.org/10.1145/3589334.3645628*

Lone, O., Stasiak, T. and Doran, H. D. (2024). Automated and orchestrated ci/cd pipelines in industrial protocol certification testing, *2024 IEEE 20th International Conference on Factory Communication Systems (WFCS)*, pp. 1–4.

Mastropaolo, A., Zampetti, F., Bavota, G. and Di Penta, M. (2024). Toward automatically completing github workflows, *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3597503.3623351*

N, V. S., Mohan Saini, L. and Mohan, H. (2022). Cloud computing in automation testing, *2022 International Conference on Edge Computing and Applications (ICECAA)*, pp. 31–36.

Pan, Z., Shen, W., Wang, X., Yang, Y., Chang, R., Liu, Y., Liu, C., Liu, Y. and Ren, K. (2024). Ambush from all sides: Understanding security threats in open-source software ci/cd pipelines, *IEEE Transactions on Dependable and Secure Computing* **21**(1): 403–418.

Patel, A. R. and Tyagi, S. (2022). The state of test automation in devops: A systematic literature review, *Proceedings of the 2022 Fourteenth International Conference on Contemporary Computing*, IC3-2022, Association for Computing Machinery, New York, NY, USA, p. 689–695.
URL: *https://doi.org/10.1145/3549206.3549321*

Pratama, M. R. and Sulistiyo Kusumo, D. (2021). Implementation of continuous integration and continuous delivery (ci/cd) on automatic performance testing, *2021 9th International Conference on Information and Communication Technology (ICoICT)*, pp. 230–235.

pytest-html contributors (2024). pytest-html documentation.
URL: *https://pytest-html.readthedocs.io/*

Rangnau, T., Buijtenen, R. v., Fransen, F. and Turkmen, F. (2020). Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines, *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 145–154.

Vihovde, E. H. and Meng, Q. (2024). Test-driven development: Ensuring code quality in integration with ci/cd, *2024 8th International Conference on Management Engineering, Software Engineering and Service Sciences (ICMSS)*, pp. 8–11.

Wang, Y., Mäntylä, M. V., Liu, Z. and Markkula, J. (2022). Test automation maturity improves product quality—quantitative study of open source projects using continuous integration, *Journal of Systems and Software* **188**: 111259.
URL: *https://www.sciencedirect.com/science/article/pii/S0164121222000280*