The Indonesian Journal of Computer Science
www.ijcs.net
Volume 14, Issue 4, August 2025
https://doi.org/10.33022/ijcs.v14i4.4967

Automation and Selection Technique for Regression Testing: An Empirical Analysis

Muhammad Hilman[1], Wulan Mantiri[2]
muhammad.hilman@ui.ac.id[1], wulan.mantiri@ui.ac.id[2]
[1,2]Computer Systems Lab (CSL), Faculty of Computer Science, Universitas Indonesia

Article Information

Received : 6 Aug 2025
Revised : 11 Aug 2025
Accepted : 27 Aug 2025

Keywords

Software testing, regression testing, test automation, test execution, test result integration

Abstract

Software testing, particularly regression testing, is a process that is required when changes are made to the software or its environment to ensure that the software continues to perform as expected. Motivated by real industry needs, this study reports on the experience of transitioning from manual to automated regression testing in one of the mobile applications at a company that provides a lifestyle super app service in Indonesia. Prior to this study, regression testing was conducted manually, resulting in significant costs and inherent subjectivity. Test automation is then applied to the activities of test execution and test result integration as an effort to increase test productivity and efficiency. This study aims to find an efficient testing alternative by separating the flow that runs tests related to changes from the flow that runs all tests. Based on the analysis of the tested application, each flow has its trade-offs. The results show that test automation can provide benefits for regression testing, application releases, and software engineering flow. The framework presented in this paper aims to serve as a guideline for other industrial applications with similar specifications that are also considering implementing test automation.

A. Introduction

Software testing is a structured process designed to ensure that an application performs as expected. When new features are added or existing ones are modified, introducing new functionality, a specific type of testing known as regression testing is carried out to verify that these changes do not negatively impact previously developed and tested functionalities [1][2]. Typically, regression testing involves re-executing all existing test cases across both modified and unmodified parts of the codebase [3][4]. As new functionalities are introduced, additional test cases must be created and executed, resulting in increased testing costs that are proportional to the growing complexity and size of the software.

In practice, software testing—particularly regression testing—represents one of the most significant cost drivers in software development projects. Studies have shown that testing can consume up to 50% of the total project budget [5][6][7][8]. These costs encompass not only financial expenditures but also human resources, labor, time, and numerous other factors. One of the key reasons regression testing can be so costly is that it is often performed manually. In manual regression testing, the quality assurance (QA) team runs all test cases on the latest version of the software and reports any issues encountered. Over time, many of these test cases become repetitive and trivial, as features that haven't been changed or are unrelated to recent updates are still tested, even though their results remain unchanged from previous test cycles. Additionally, manual regression testing is inherently subjective and highly susceptible to human error [9][10][11].

One company that has encountered significant challenges with manual regression testing is *The Company*, a technology firm offering a lifestyle super app service in Indonesia. Their partner application, AM, is a relatively new product, and its entire testing process is still conducted manually by the quality assurance (QA) team. With a total of 1,046 test cases, each regression testing cycle for the AM app is allocated five days. However, issues often arise during testing that require considerable time to resolve before they can be retested. This has become a frequent bottleneck, causing the regression testing process to exceed its estimated timeline and resulting in delays to the application's release schedule.

Since the need for regression testing extends beyond *The Company*. For the entire software industry, it is essential to find more efficient ways to accomplish this. One of the most effective solutions for reducing testing costs is the combination of test automation and the use of appropriate regression testing techniques [7][8]. A variety of testing methods have been developed to support and simplify the automation of testing within the software development lifecycle.

A case study of *The Company* serves as the foundation for this research, where real-world industry demands drive the investigation. The study takes an exploratory approach, aiming to understand the processes behind test automation, uncover new insights, and ultimately improve the productivity and efficiency of existing testing workflows. To identify the most effective combination of approaches, the research involves experimenting with various technical methods for transitioning from manual to automated testing. Observations will be made to assess whether the benefits gained are proportional to the effort required, using a cost-benefit analysis. The outcome of this study is expected to provide a practical

guideline for implementing test automation, which other applications with similar characteristics can adapt to.

B. Research Method

The research process followed in this study adapted the stages outlined by the authors in [12][13][14]. The process consisted of seven stages: problem formulation, literature review, instrument design, implementation, data collection, data analysis, and drawing conclusions and recommendations. The first stage was problem formulation, which began with identifying the research topic. The topic was selected within the scope of the AM application at *The Company* by observing the current situation and examining the issues experienced by the application. The problem was formulated through a review of related prior studies, analysis of relevant literature, and exploration of potential research opportunities that offered value to both *The Company* and the wider industry. This study analyzed the rationale for automating regression testing, identified areas suitable for automation, and examined the cost-benefit implications for *The Company* and the industry as a whole.

The second stage was the literature review. Building on the context established in the first stage, this step involved collecting and thoroughly studying relevant concepts, theories, and prior research. Various references on software testing and test automation were explored and used as the foundation for understanding. The results of this review served as key references for designing the research instruments and selecting evaluation metrics.

The third stage involved designing the research instruments and implementation plan. The instruments were selected based on the technical requirements for implementing automated testing at *The Company*. The design and planning process followed an iterative approach using RFC (Request for Comments) documents. Each RFC included a proposal supported by research and a comprehensive comparison of alternatives. Feedback and suggestions were provided through comments or meetings. An RFC document was considered complete once all feedback had been incorporated and the proposal received approval from all relevant stakeholders.

The fourth stage involved implementing and automating the interface testing. The initial implementation followed the previously designed research instruments. Additional test cases and improvements were introduced incrementally. Each test case was executed automatically, and its performance was recorded. These performance metrics were collected for further analysis.

The fifth stage involved data collection, which was conducted concurrently with the implementation stage. The collected data included performance results from both general interface testing and the specific automation of regression testing. While performance measurement was carried out automatically, data compilation into a structured and readable format was handled manually.

The sixth stage was data analysis. The analysis focused on the costs and benefits of the implemented testing automation based on the collected data. It also included evaluations using relevant metrics. Observations from the implementation of alternative approaches and their outcomes were also documented.

The study concluded by drawing conclusions and providing recommendations. These were based on the findings and analysis from the earlier

stages. The conclusions addressed the initial research questions and offered a clear summary of the study's results. The recommendations, based on practical insights gained during the research process, aimed to support further efforts related to interface testing automation, particularly in the context of regression testing.

Instruments

The technologies used in this research included TypeScript, JavaScript, and Kotlin. TypeScript served as the primary programming language for developing the application's interface using the React Native library. JavaScript was used to write configuration scripts and tasks not directly related to interface development. Meanwhile, Kotlin supported the development of modules that required integration with Android hardware. Source code was managed using Git on the GitHub platform, specifically within a repository named BN. All pull requests to the BN repository were integrated with a Continuous Integration (CI) pipeline through Jenkins.

Interface testing for the AM application was conducted using Jest and the Testing Library. Jest provides built-in support for regression test selection (RTS) through the onlyChanged and changedSince configurations, enabling efficient file-level test execution. This feature represents a lightweight and commonly adopted approach in JavaScript-based industry projects [15][16]. In addition to Jest, tests adhered to standardized conventions from Testing Library, a framework prominent in the JavaScript development community.

The design of the testing implementation was limited to interface testing without external dependencies such as real data from web APIs. To enable independent testing of the AM application, this study utilized Nock, an HTTP interceptor library that intercepts HTTP calls and returns customizable responses. Nock was selected because it allows for manipulating API responses while preserving the actual application implementation.

The structure of test cases followed a hierarchical model starting at the project level. A project could contain multiple test suites, usually grouped by functional requirements. Each test suite could include one or more test groups, typically mapped to application modules. Test groups could be nested recursively and contain multiple test cases. By default, test cases followed TestRail configurations, which included type, priority, estimated time, test information, and automation category. These configurations were optional but helpful in filtering or organizing test cases according to specific testing requirements.

The structure for executing tests, referred to as executable tests, also began at the project level, which had access to Test Milestones, Test Plans, and Test Runs. Test Milestones were used to track progress and release goals in parallel, particularly when multiple targets were managed simultaneously. In the AM application, each release was aligned with a single Test Milestone, which could be subdivided into Test Sub-Milestones to manage timelines for specific sub-services, such as the back office, API, or the AM application itself. Test Plans were used to manage multiple Test Runs grouped by type. These groupings, called Test Entries, consisted of Test Runs that shared the same name but differed in configuration, such as testing methods or target systems. Each Test Run contained test groups with test cases that inherited attributes from their definitions, along with test-specific IDs and result-related data such as execution status and comments.

Finally, TestRail provided an API for integrating third-party applications or tools. In this research, the TestRail API was used to automate the submission of test results from automated testing and to create Test Plans and Test Runs through scripting. This integration supported a more efficient, maintainable, and scalable testing workflow throughout the project.

Evaluation Metrics

The performance evaluation of the implementation focused on three quantitative metrics: code coverage, test automation productivity, and a cost-benefit analysis. These metrics provided measurable insights, while a qualitative evaluation was conducted to analyze the potential benefits of test automation.

To track the progress of interface testing, code coverage was measured to indicate the comprehensiveness and effectiveness of the executed tests [17][18]. Code coverage typically includes four criteria: statement coverage, branch coverage, function coverage, and line coverage. These provide visibility into which statements were executed, which conditional paths (e.g., IF–THEN–ELSE or DO WHILE) were traversed, and which functions were called. The number of executable lines run during testing. This research used Jest's built-in code coverage reporting, which presents these metrics alongside intuitive textual and tabular summaries. The implementation was designed to collect coverage data from all files related to the interface, including those integrating with Redux.

In evaluating test automation productivity, two metrics were used: automated test case coverage and test design productivity [9][10][11]. Automated test case coverage measures the proportion of manual tests that have been automated, calculated using Equation 1

$$cova = \frac{tca}{tcm+tca} \times 100,\ 0 \leq cova \leq 100 \quad (1)$$

Here, *tca* and *tcm* represent the number of automated and manual test cases, respectively, and *cova* denotes the percentage of automated test coverage. Test design productivity measures the number of test cases created per unit time (e.g., person-hours) using Equation 2

$$productivity_x = \frac{tc_x}{t} \quad (2)$$

The data for both metrics were recorded incrementally, and the productivity values at different stages were compared to observe trends in automation. Test case data were sourced from TestRail, and productivity was calculated in person-hours.

To assess the impact of automation compared to manual testing in regression testing, this study considered the perspective of the authors in [7], who argued that manual and automated testing are fundamentally different processes, each uncovering different types of errors. Therefore, direct comparisons based on cost or number of defects may lack meaning. In line with this, the evaluation focused only on the time-based cost analysis and qualitative insights.

The cost of regression testing was modeled using fixed and variable components, defined as V_x and W_x for group *x*. The formula for total cost is described in Equation 3

$$cost_x = V_x + tc_x \times W_x \quad (3)$$

Fixed costs refer to the initial implementation time, whereas variable costs increase with the number of test cases. For automation, this expands into Equation 4

$$cost_a = V_a + tca \times W_a = V_a + tca \times (d_a + e_a + r_a) = V_a + D_a + E_a + R_a \quad (4)$$

Here, V_a represents one-time automation implementation cost; W_a includes the cumulative time spent on writing (d_a), executing (e_a), and integrating (r_a) automated tests. Their corresponding uppercase variables (D_a, E_a, R_a) represent the total time across all automated tests.

Two assumptions were made in calculating automation cost: first, the initial cost (V_a) was limited to coding activities and excluded administrative tasks, RFC writing, and meetings; second, the required libraries for execution were assumed to be pre-installed in the CI environment. Installation time was excluded to avoid skewed results, as it involved unrelated external dependencies.

For manual testing, the assumption was that the application and test environment were fully set up before testing. Hence, the only cost incurred was the time required for execution. This simplified cost model, using $V_m = 0$ and $W_m = e_m$ is described in Equation 5

$$cost_m = V_m + tcm \times W_m = 0 + tcm \times e_m = E_m \quad (5)$$

This approach replaced per-test W_x values with total execution time for all test cases, accommodating real-world conditions where each test may have different priorities and costs. As the authors in [7] noted, many real-world projects do not treat all test cases equally—some are more critical due to their likelihood of detecting defects or their impact on the system.

C. Results and Discussion

The evaluation of interface test automation was based on data collected across 20 phases of test additions and implementation refinement. Each phase contributed data on code coverage metrics from Jest, automation productivity, and time-based execution costs.

Table 1. Summary of Testing Performance Across 20 Phases

Phase	Code Coverage (%)	Test Case Coverage (%)	Productivity	Total Pipeline Time (seconds)		
				Feature	All (first)	All (second)
1	1.62	1.33	1.6	62.48	29.49	13.28
2	5.11	3.34	2	49.63	29.16	14.8
3	8.53	5.42	2.08	51.14	36.24	16.95
4	8.2	7.17	2.63	71.4	46.16	20.6
5	8.85	8.76	4.75	80.52	52.6	23.5
6	10.92	10.51	5.25	60.22	58.33	23.14
7	12.31	12.18	3.17	66.22	48.16	23.55
8	13.15	13.18	2.4	52.75	55.95	24.38
9	19.11	14.6	1.06	81.27	57.58	24.87
10	19.47	14.01	-	79.37	56.62	26.65
11	20.95	15.68	2.5	46.65	61.28	27.76
12	30.57	16.6	2.75	81.54	72.81	31.07
13	32.83	17.6	3.25	84.37	65.2	27.09
14	35.12	18.68	3.25	85.86	68.24	29.82
15	36.5	19.93	3.75	49.34	65.03	28.68
16	38.68	21.35	4.25	43.1	67.4	28.39
17	39.87	22.69	4	70	63.75	31.71
18	41.07	23.94	4.29	49.53	67.71	35.84
19	42.24	25.1	4.67	55.14	66.27	31.66
20	43.57	26.77	5	60.59	73.52	33.24

Table 2. Regression Testing Automation Performance

Release		1	2	3	4	5
Phase		-	-	-	3	10
Code Coverage (%)	**Statements**	-	-	-	8.16	19.01
	Branch	-	-	-	8.39	19.55
	Functions	-	-	-	8.76	20.21
	Lines	-	-	-	8.24	18.96
Number of Automated Test Cases		-	-	-	65	168
Number of Test Cases		689	954	1051	1004	1199
Test Case Coverage (%)		-	-	-	6.47	14.01
$cost_a$	V_a **(hours)**	-	-	-	36	59
	D_a **(hours)**	-	-	-	34	43
	E_a **(seconds)**	-	-	-	32.78	48.29
	R_a **(seconds)**	-	-	-	3.05	5.97
	Total E_a, R_a **(seconds)**	-	-	-	37.89	57.15
$cost_m$	E_m **(days)**	5	5	5	5	5
	E_m **(hours)**	80	80	80	80	72
Total Time (hours)		80	80	80	150	174

Table 1 consolidates the most relevant and representative metrics for evaluating test automation performance. It includes the code coverage percentage, the TestRail test case coverage percentage, the calculated test design productivity, and the total execution time of both CI pipelines. These metrics provide a comprehensive view of the progress, efficiency, and cost implications of the automated testing implementation throughout the phases.

The data related to interface test automation metrics for regression testing covers the last five release cycles. These include three releases that employed manual regression testing and two releases that adopted automated regression testing. For each release, the data collected included the testing phase, during which regression testing was conducted, code coverage metrics from Jest, test case coverage, and the time-based costs of executing both manual and automated tests. The complete dataset is presented in Table 2.

Test case coverage data included the number of test cases executed automatically and the total number of test cases in the Test Run (consisting of both manual and automated tests). The automation testing time cost was divided into five components: initial implementation cost (in hours), test writing time (in person-hours), clean test execution time (in seconds), result integration time (in seconds), and total execution time (in seconds). For manual testing, only execution time was recorded, presented in two time units: the first row shows the time in days, and the second in person-hours, calculated using Equation 6

$$Execution\ Time = n\ days \times 8\ hours\ per\ day \times 2\ testers \quad (6)$$

The total time cost for both automated and manual testing was then aggregated and recorded in hours in the final row.

The implementation results demonstrated that automated testing had a noticeable impact on the development workflow of the AM application. Although the number of modules tested automatically remained limited due to time constraints and test selection, two key implications emerged when automated tests successfully detected minor oversights that could have led to critical failures.

The first implication was that automated testing reduced code delivery time by minimizing the need for manual regression testing. The second was that the development team made code changes with greater confidence, as the automated tests helped ensure that existing functionality remained intact. In other words, automated testing contributed to an increased level of confidence among developers in the reliability of their code.

The analysis of interface test automation metrics and regression testing appeared in the following subsection. This analysis unpacked the data from the corresponding tables, described the observed outcomes, and discussed the implications drawn from the processed results.

Automated Testing Analysis

The data showed that code coverage generally increased as more modules were tested across nearly every phase. However, Phase 4 presented an exception, where a decline in coverage was observed—except for function coverage—despite the addition of three tested modules. This anomaly may have resulted from external factors, such as dynamic changes in the source code repository, which can affect overall code coverage. Significant additions or removals of untested code can either increase or decrease coverage, depending on the impact of the related modules. Modules containing relatively little code may have less influence compared to more complex modules with larger codebases.

Another anomaly appeared in Phase 12, where code coverage was split into two segments with a noticeable difference between them. The first segment showed a significant increase compared to Phase 10, similar to the jump from Phase 8 to Phase 9, likely due to the testing of complex modules such as the Home and Chat pages. The second segment marked a shift where the coverage collection configuration was adjusted to include files related to Redux. The observed increase aligned with expectations, as Redux code was also tested through integration tests. Before this change, Redux-related files were not considered valid or relevant to the interface and were thus excluded from coverage collection.

Over two months and 20 development phases, statement coverage increased from 0% to 43.57%. The central tendency of coverage change per phase was 1.39%, as measured by the median, rather than the mean, which was less suitable due to the presence of anomalies, such as the drop in Phase 4 (-0.33%) and the sharp rise in Phase 12 (9.62%).

Changes in the number of tests executed in Jest and reported to TestRail were documented to highlight the points of divergence between the two sources. Three such discrepancies were identified in Phases 4, 10, and 11. In Phase 4, the number of tests remained the same; however, an error occurred when the developer assigned duplicate TestRail test case IDs to two different tests. In Phase 10, the development and QA teams were finalizing which test cases fell within the scope of interface testing. As a result, several integration tests previously linked to TestRail were excluded by removing their test case IDs from the titles. However, the tests themselves were retained as unit tests in Jest. Phase 11 marked a shift in testing conventions, where it became acceptable for a single test to be associated with multiple TestRail test case IDs.

Observations on test-writing durations indicated that implementing automated tests initially required a significant adjustment period, although the process began to stabilize after Phase 4. Across all 20 phases, the average test-writing time was 6.3 hours, ranging from a minimum of 3 hours to a maximum of 16 hours. The initial phases had longer durations, primarily because the concept of automated testing and the testing strategy for the AM application were still unfamiliar to the development team. After the early phases, most durations stabilized between 3 and 8 hours, except for Phase 9.

In terms of productivity values, a monotonic increase occurred until Phase 7, after which a decline was observed due to the nature of the tested module. The module in Phase 7 required a deeper contextual understanding and the development of new test tools, both of which consumed additional time. This downward trend continued through Phase 9, where productivity reached its lowest point across all phases, even lower than the initial productivity in Phase 1. The module tested in Phase 9 was the Chat module, which required significantly different preparation not accounted for in the initial setup. Since the time spent preparing for Phase 9 was included in the test-writing duration, the productivity metric was significantly impacted.

The test case coverage was calculated by dividing the number of test cases executed by the total of 1,199 test cases. Over the two-month development period, test case coverage increased from 0% to 26.77%, with a single decline observed in Phase 10, as previously explained. Excluding Phase 10, the average change in test case coverage per phase was 1.45%, with a minimum of 0.92% and a maximum of 2.09%. Unlike code coverage, changes in test case coverage per phase were expected to increase linearly, since the test execution workload was deliberately distributed evenly across all phases.

The average time spent on fetching a branch from Git was 29.25 seconds, with a minimum of 20 seconds and a maximum of 38 seconds. The time required for the first branch fetch varied consistently, whereas the second branch fetch consistently took only two seconds. This discrepancy was likely due to the second fetch having prior access to the Git repository, facilitated by the initial fetch, which resulted in a shorter and more stable execution time.

The total test execution time depended on the number of tests run, with fewer tests corresponding to shorter durations. In contrast, the total pipeline duration was influenced by two unrelated factors: the number of tests and the Git fetch time, leading to inconsistent patterns. The difference between total and net test execution time stemmed from the teardown process within the Jest testing environment. An anomaly occurred in Phase 17, where this difference exceeded 11 seconds, while the typical range for teardown time differences in the Feature pipeline was between 2.4 and 3.3 seconds.

The first test run showed a trend of increasing duration in line with growing code coverage. The second test run also showed an increase, albeit at a slower rate. A substantial difference was observed between the first test run and the subsequent ones. The investigation revealed that Jest applied automatic caching to previously executed tests, affecting all pipelines. Thus, the first run consistently required the longest execution time. By Phase 20, the first run took approximately 73 seconds,

while subsequent runs took 33 seconds. For both runs, the difference between total and net execution time ranged from 1.5 to 3.2 seconds.

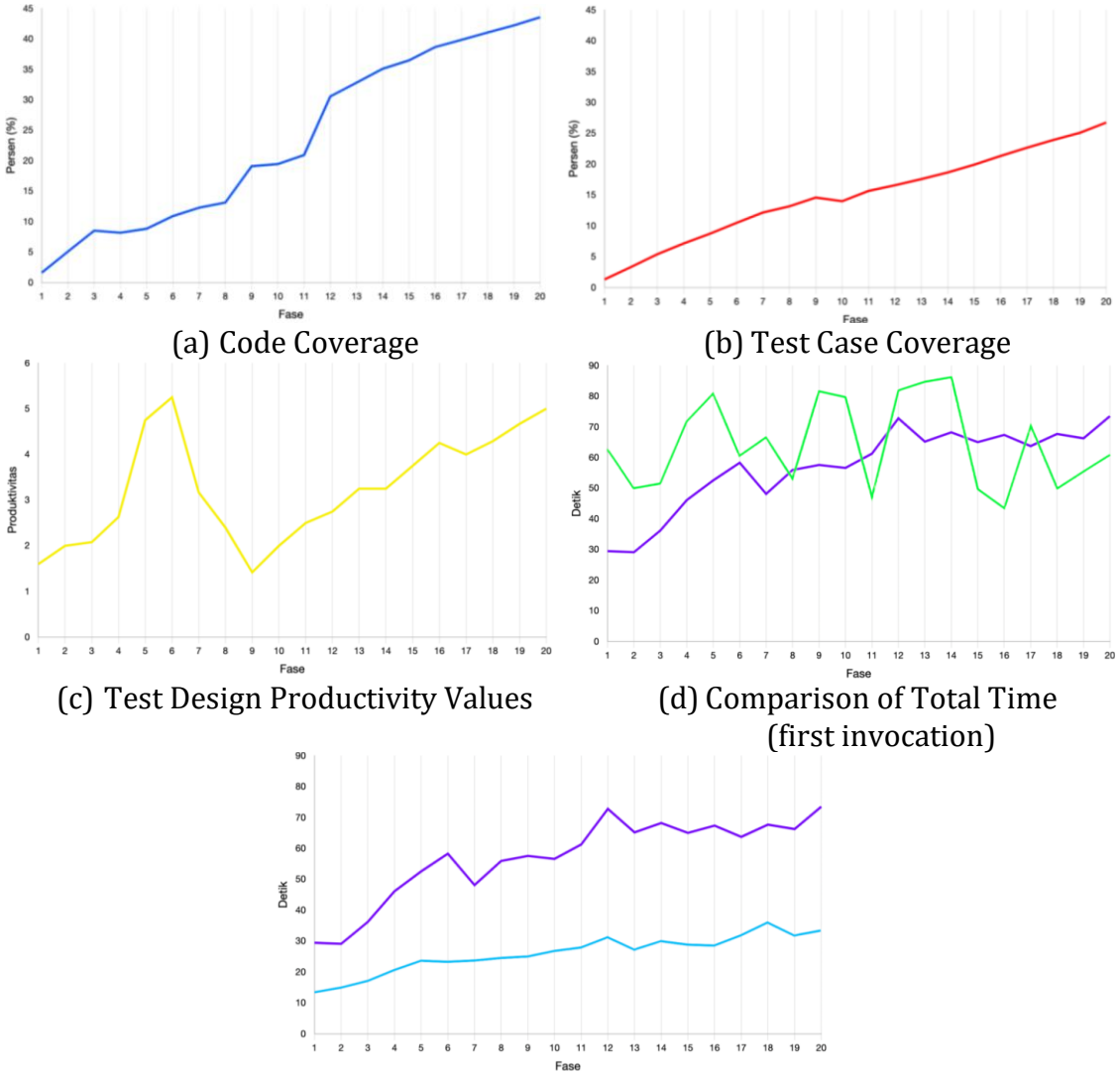
Observations across both pipelines were made using three total time comparisons during the first executions: (1) similar number of tests, (2) moderately different numbers, and (3) significantly different numbers. The first comparison, conducted in Phase 2, involved both pipelines running 40 tests. Although test execution times were similar, the All pipeline outperformed the Feature pipeline in total time, contrary to expectations, since the Feature pipeline included additional steps to compare the source and target Git branches. This indicated that the Feature pipeline was less efficient when handling changes involving core components or a large number of tests.

The second comparison, in Phase 3, showed that the Feature pipeline ran 26 tests and the All pipeline ran 65 tests. Despite the Feature pipeline having a shorter test duration, its total time remained longer than the All pipeline, implying that the test count difference must be significant enough to offset the overhead. The third comparison, seen in Phase 8, involved the Feature pipeline running 14 tests and the All pipeline running 159 tests. In this case, the Feature pipeline was 3 seconds faster than the All pipeline, even after accounting for Git-related overhead. The difference of 145 tests proved sufficient to compensate for the additional Git time, suggesting that the Feature pipeline offered tangible benefits at a larger scale.

To verify this assumption, Phases 15 to 20 were analyzed. In all these phases, the Feature pipeline consistently executed far fewer tests (14–28) than the All pipeline (245–327). Results showed that the total time in the Feature pipeline was consistently lower, aligning with expectations, except in Phase 17. The anomaly in Phase 17 was attributed to unusually high Git fetch time and extended teardown time, which added 11 seconds. Overall, these findings indicate that the Feature pipeline is optimal for minimizing CI pipeline duration when developing features that do not involve core components or large-scale code changes.

To provide a holistic view of the metrics, Figure 1 presents five visualizations summarizing the testing performance from Table 1: (a) code coverage, (b) test case coverage, (c) test design productivity values, (d) a comparison of total time between the Feature pipeline and the All pipeline based on first invocation, and (e) a comparison of total time in the All pipeline between first and second invocations. In Table 1, it can be observed that code coverage and test case coverage overlapped during Phases 5 to 8. From Phase 9 onwards, the percentages began to diverge, showing an even wider gap by Phase 12. The trend in code coverage growth depended on the complexity of newly tested modules, while the increase in test case coverage followed a linear pattern, as seen in Figure 1b. The analysis suggested that high code coverage does not imply high test case coverage, and vice versa.

In Figure 1c, test design productivity values fluctuated over time, with the highest recorded at 5.25 tests per hour. Three main factors were identified as influencing productivity. The first factor was the availability of reusable test tools for the target module. The second was whether the development team already had sufficient context about the implementation being tested. The third was the team's learning curve. Developers were observed to complete testing more efficiently when they were already familiar with the tools and had tested similar test cases related to the code implementation.



(a) Code Coverage

(b) Test Case Coverage

(c) Test Design Productivity Values

(d) Comparison of Total Time (first invocation)

(e) Comparison of Total Time (second invocation)

Figure 1. Graphical Visualization of the Summary of Testing Performance

Figure 1d shows that the total timeline for the All pipeline tended to remain below the Feature pipeline in the earlier phases, but the opposite trend emerged toward the end. As previously explained, this shift primarily resulted from the difference in the number of tests executed by each pipeline. A significant gap (more than 145 tests) was often enough to offset the average additional Git time, allowing the Feature pipeline to outperform the All pipeline. Finally, Figure 1e shows that subsequent invocations—represented here by the second run—in the All pipeline were substantially faster than the first run due to caching mechanisms. The study also found that this pattern applied similarly to test execution times in the Feature pipeline.

Automated Regression Testing Analysis

The explanation begins by providing context for the manual regression testing practices employed prior to this study. As shown in Table 2, manual testing

during Releases 1 through 3 included only three metrics: total test cases, execution time in days, and total execution time in hours. Starting from the initial release of the application before Release 1, each regression testing cycle was allocated five working days for two testers, totaling 80 person-hours. These five days were typically divided into two phases: the first one to three days were used to test modules affected by new features, while the remaining days were allocated for regression testing of unaffected modules. The intended expectation was for the entire process—including testing all available cases, reporting bugs to the development team, and verifying the fixes—to fit within this five-day window.

As a result, the QA team needed to manage their time efficiently to ensure no test cases were skipped. A contributing factor to this efficiency was the increasing familiarity of the QA team with older test case workflows, allowing them to execute those tests more quickly. However, as the number of test cases grew and new features were continuously introduced, the QA team sometimes had to make compromises to meet the time constraints. This typically involved prioritizing test execution based on importance and complexity. Examples of test cases considered "skippable" included hard-to-reproduce error flows or older modules that had never failed and were assumed to be stable. In practice, this strategy proved unreliable and often resulted in delayed releases, as bugs were only discovered after testing.

This study introduced an experimental hybrid testing approach in Releases 4 and 5, utilizing the same five-day time benchmark for manual testing. The initial implementation took 36 hours over four weeks. Regression testing for Release 4 was conducted after Phase 3, with code coverage recorded at approximately 8% and test case coverage at 6.47%. A total of 65 test cases were written, requiring 34 hours of test authoring. According to the QA team, these 65 test cases did not significantly impact their manual testing workload; therefore, manual testing still required 80 hours. The total execution time for automated regression testing on these 65 test cases was 37.89 seconds, comprising 32.775 seconds for test execution and 5.14 seconds for result integration. Overall, the total time spent on testing in Release 4 was approximately (36 + 34 + 80) hours + 37.89 seconds, or around 150 hours.

In comparison, the total time for Release 5 increased to 174 hours. This was attributed to significant increases in initial implementation and test writing time, which rose to 59 hours and 43 hours, respectively, over a five-week development period. The initial implementation was extended to incorporate feedback that had not yet been addressed in Release 4 and was refined to meet project requirements fully. Test authoring time also remained relatively high, as new tests continued to be added during the migration to automated testing. Despite this, manual testing time was reduced to 72 hours. The QA team reported that one tester completed their portion of testing a day earlier because automated tests covered some of their assigned features. This indicated a noticeable impact of test automation by Release 5, with test case coverage reaching 14%. The total time for automated regression testing of 168 test cases was 57.15 seconds, including 48.295 seconds for execution and 5.97 seconds for result integration.

To support a long-term analysis, this study projected the performance of regression testing if the next application release (Release 6) were conducted after Phase 20, as shown in Table 3. The projection drew on data from the regression testing automation experiment conducted in Phase 20 and the estimated duration

of manual testing provided by the quality assurance team. No further implementation effort occurred beyond Phase 20, so the implementation cost was assumed to be zero.

The execution time of automated regression testing was measured directly from the test run in Phase 20, under the same conditions and implementation as those of the previous releases. The total execution time was 78.61 seconds, comprising 70.453 seconds for test execution and 8.08 seconds for result integration. After the regression testing, optimization was applied by converting arrays to sets for element lookups. As a result, the result integration time was reduced to 4.38 seconds for 321 tests, making it even faster than the result integration time recorded in Release 5.

According to estimates from the quality assurance team, the features already covered by automated tests reduced manual testing time by two to three working days for one tester. The testing allocation was adjusted to five days for tester one and three days for tester two, totaling eight workdays or 64 person-hours. Based on this projection, the total testing time dropped significantly to 106.5 hours.

Three long-term outcomes emerged from the application of automated testing. First, the duration required to write new tests consistently decreased over time and eventually stabilized at a lower threshold. This occurred because, once tests were automated, no additional effort was needed aside from maintenance, and the number of new tests only needed to accommodate newly added features.

Second, the execution time of automated tests remained within a minute's range, even as code coverage increased. In Phase 20, 43% code coverage required only 70 seconds (approximately one minute) to execute. Since the AM application contained no use cases involving excessive iteration that could affect runtime, the increase in code coverage was expected to remain within the same time range. This indicated that the execution time of automated tests could be considered negligible compared to the total regression testing duration, which was measured in hours.

Table 3. Projected Performance of Regression Testing Automation

Code Coverage (%)	**Statements**	43.57
	Branch	39.1
	Functions	38.7
	Lines	43.63
Number of Automated Test Cases		321
Number of Test Cases		1199
Test Case Coverage (%)		26.77
$cost_a$	V_a **(hours)**	0
	D_a **(hours)**	42.5
	E_a **(seconds)**	70.45
	R_a **(seconds)**	6.14
	Total E_a, R_a **(seconds)**	4.38
$cost_m$	E_m **(days)**	5
	E_m **(hours)**	64
Total Time (hours)		106.5

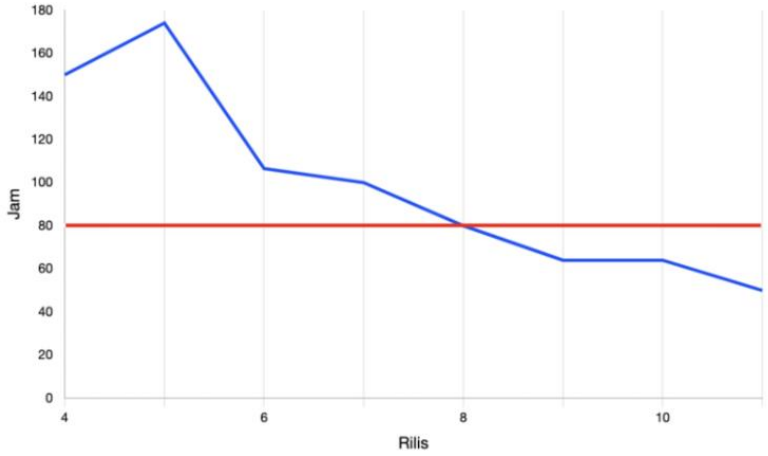


Figure 2. Illustration: Total Regression Testing Time in the Long Term

Third, the duration of manual testing steadily declined as the coverage of automated testing increased. Although the total regression testing time could not be definitively concluded in this study, it was expected to fall within one of two outcome scenarios.

The first scenario represented an ideal condition in which the total regression testing time stabilized at less than five days, as illustrated in Figure 2. This scenario assumed minimal changes were introduced to existing features, and if any changes did occur, they were not breaking changes. Automated tests that replaced manual testing for these features remained relevant without requiring intensive maintenance. As a result, the time contribution from automated testing was minimal or could be disregarded in the overall time calculation.

The remaining time could then be used to reduce either the time allocation or the resources required, for instance, by shortening the regression testing schedule to three days or assigning only one tester to perform the regression testing. This allowed the testing team's resources to be redirected toward more meaningful tasks, rather than being focused solely on regression testing.

The second scenario is one in which the total time remains unchanged, as illustrated in Figure 3. The assumption in this scenario is the worst-case situation where development not only adds new functionality with each release but also modifies existing functionality. In this case, tests become out of sync with the product and require updates. The duration for writing tests remains high due to the constant addition of features and intensive maintenance in each release.

At first glance, the impact of test automation may not appear significant because the total time is “about the same” as manual testing. However, the second scenario offers a different outcome than previous manual testing, where the five-day window can now accommodate the entire regression testing process without requiring compromises. With time made available from automated testing, regression testing can complete all test cases earlier, allowing bugs to be identified and fixed before the deadline. All the benefits presented in the second scenario also apply to the first scenario.

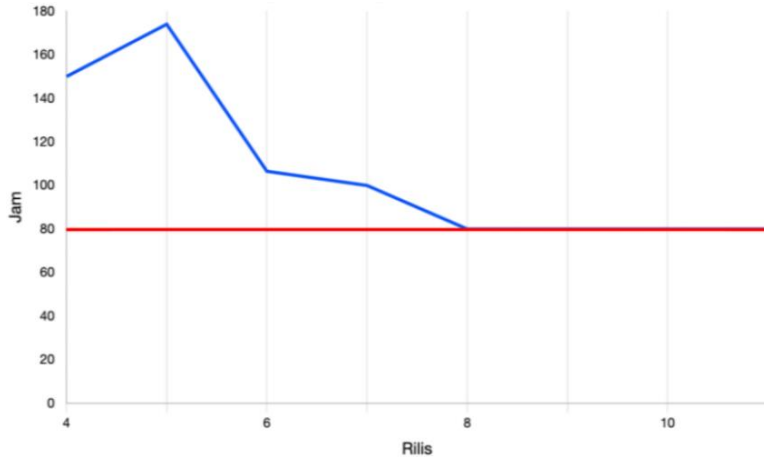


Figure 3. Illustration: Long-Term Outcomes of Test Automation

The implication of the two scenarios above is that test automation can reduce the risk of application release delays caused by regression testing. Even though the number of features and tests will increase as development progresses, the regression testing workload becomes more predictable and does not require compromise. Future application releases can be carried out more regularly, with shorter intervals compared to the previous release intervals that could stretch from one to two months.

D. Conclusion

Motivated by real-world industrial needs, this study reports the experience of transitioning from manual regression testing to automated regression testing for one of the applications at PT. XYZ. Before the study, regression testing was conducted manually, which was subjective and costly. Test automation was then implemented as a means to enhance testing productivity and efficiency. This study automated the test execution phase and integrated the results for the test analysis phase. The preparation and implementation framework of test automation described in this paper is expected to serve as a guideline for other industrial applications with similar specifications that also intend to adopt test automation.

Over two months, this study developed more than 300 automated test cases and executed regression testing for two application releases. The performance of the automated testing was observed based on metrics such as code coverage, productivity, and time-cost analysis. The results were then used to analyze the costs and benefits of regression testing using the hybrid testing approach.

Experiments were conducted to find efficient alternatives for test execution by splitting the CI pipeline into two types: the Feature pipeline, which runs only tests related to recent changes (or a subset of tests), and the All pipeline, which runs all tests. The analysis showed that the Feature pipeline is not suitable for changes that involve core components or impact multiple tests simultaneously. However, it is appropriate when the number of tests run is significantly smaller than the total test count. Based on the tested application, the Feature pipeline only outperforms the All pipeline in execution time if the test count difference exceeds 145. This is because the Feature pipeline requires additional time to compare the source and target branches in Git.

The findings of this study demonstrate that test automation has a tangible impact on three aspects: regression testing, application releases, and the application development workflow. First, the effect of automation was already evident in time allocation (person-hours) at just 14% test case coverage, or 168 tests. In the long-term analysis, test automation is projected to reduce the overall time required for regression testing, or at least stabilize the total time, even as the application grows and the number of tests increases. Second, automation can reduce the risk of release delays caused by regression testing. As a result, application releases can be scheduled more frequently and regularly. Lastly, automation can reduce the need for manual regression testing for every code change while still ensuring the quality of the developed product.

One limitation of this study is the short duration available for data collection. The data was collected over two months, during which only two application releases occurred. Therefore, the impact analysis of test automation was based on limited data. A long-term analysis was conducted theoretically, presenting abstract scenarios based on the testing team's domain knowledge and quantitative projections from real-world experiments. Additionally, this study was scoped to the AM application at PT. XYZ. Research in different contexts may yield different results.

Based on the research conducted, several suggestions can be made to improve future studies. First, the study could be conducted over a more extended period or applied to more application releases. The more data collected, the more concrete the analysis becomes, which could potentially differ from the projected scenarios. Second, future research could aim to implement end-to-end automation by generating test cases automatically during the test design phase. While this would broaden the scope and complexity of the study, it would also contribute significantly to both academic and industrial fields related to test automation, which is still a relatively new field. Lastly, future studies could enhance the test automation artifacts or processes by exploring the latest conventions and alternatives.

E. Acknowledgment

The author would like to express sincere gratitude to *The Company* for providing access to the necessary data and resources that made this research possible. Special thanks are also extended to the Computer Systems Laboratory (CSL) for their continuous support, guidance, and facilities throughout this study.

F. References

[1] Hasnain, M., Pasha, M. F., Ghani, I., & Jeong, S. R. “Functional requirement-based test case prioritization in regression testing: a systematic literature review,” *SN Computer Science*, vol. 2, issue 6, pp. 421. 2021.

[2] Akin, A., Sentürk, S., & Garousi, V. “Transitioning from Manual to Automated Software Regression Testing: Experience from the Banking Domain,” In Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC), pp. 591-597, 2018.

[3] Sawant, P. D. “Test Case Prioritization for Regression Testing Using Machine Learning.” In Proceedings of the 2024 IEEE International Conference on Artificial Intelligence Testing (AITest), pp. 152-153, 2024.

[4] Kandil, P., Moussa, S., & Badr, N. "A Study for Regression Testing Techniques and Tools," *International Journal of Soft Computing and Software Engineering*, vol. 5, pp. 64-84, 2015.
[5] Homès, B. Fundamentals of software testing. John Wiley & Sons, 2024.
[6] Corradini, D., Zampieri, A., Pasqua, M., Viglianisi, E., Dallago, M., & Ceccato, M. "Automated black-box testing of nominal and error scenarios in RESTful APIs," *Software Testing, Verification and Reliability*, vol. 32, issue 5, e1808, 2022.
[7] Ramler, R., & Wolfmaier, K. "Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost," In Proceedings of the 2006 International Workshop on Automation of Software Test, pp. 85-91, 2006.
[8] Rosero, R. H., Gómez, O. S., & Rodriguez, G. "15 Years of Software Regression Testing Techniques -- A Survey," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, pp. 675-689, 2016.
[9] Thant, K. S., & Tin, H. H. K. "The impact of manual and automatic testing on software testing efficiency and effectiveness," *Indian Journal of Science and Research*, vol. 3, issue 3, pp. 88-93, 2023.
[10] Haas, R., Nömmer, R., Juergens, E., & Apel, S. "Optimization of automated and manual software tests in industrial practice: A survey and historical analysis," *IEEE Transactions on Software Engineering*, vol. 50, issue 8, pp. 2005-2020, 2024
[11] Sharma, R. M. "Quantitative Analysis of Automation and Manual Testing," *International Journal of Engineering and Innovative Technology*, vol. 4, issue 1, pp. 252-257, 2014.
[12] Baltes, S., & Ralph, P. "Sampling in software engineering research: A critical review and guidelines," *Empirical Software Engineering*, vol. 27, issue 4, pp. 94, 2022.
[13] Guevara-Vega, C., Bernárdez, B., Durán, A., Quina-Mera, A., Cruz, M., & Ruiz-Cortés, A. "Empirical strategies in software engineering research: A literature survey," In Proceedings of the 2021 Second International Conference on Information Systems and Software Technologies (ICI2ST), pp. 120-127, 2021.
[14] Kothari, C. R. Research methodology: Methods and techniques (2nd ed.). New Age International (P) Limited, 2004.
[15] Wu, M., Dong, W., Zhao, Q., Pan, Z., & Hua, B. "An Empirical Study of Lightweight JavaScript Engines," In Proceedings of the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), pp. 413-422, 2023.
[16] Chen, Y. "NodeSRT: A Selective Regression Testing Tool for Node.js Application," In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 126-128, 2021.
[17] Bandyopadhyay, B. A Comprehensive Study on Code Coverage Analysis for Effective Test Development/Enhancement Methodology. American Journal of Science & Engineering, vol. 3, issue 2, pp. 31-36, 2022.
[18] O'Regan, G. Concise Guide to Software Engineering: From Fundamentals to Application Methods. Springer International Publishing, 2017.