Efficiency of Regression Testing Strategies in CI/CD Environments

Dmytro Lukavenko [1*], Maksym Delembovskyi [2,†]

[1,2] *Kyiv National University of Construction and Architecture (KNUCA), 31 Povitroflotsky Avenue, Kyiv, 03037, Ukraine*

Abstract

Regression testing is a critical element of software quality assurance, especially in Continuous Integration and Delivery (CI/CD) environments. The main challenge of regression testing in CI/CD is selecting the optimal strategy for a specific project, considering timelines, available resources, and the quality requirements of the final product. This study provides a comprehensive analysis of the efficiency of various regression testing strategies, including full regression testing, incremental regression testing, test prioritization, and test minimization. The primary objective of the research is to assess the ability of each strategy to detect defects while minimizing resource expenditure in the context of rapid codebase changes and frequent releases. By calculating the Overall Efficiency Index and reviewing the literature, the paper offers recommendations for selecting the optimal testing strategy in CI/CD environments. The study's findings indicate that employing a combination of strategies can achieve a balance between defect detection efficiency and testing costs, which is crucial for maintaining high software quality in the dynamic conditions of CI/CD.

Keywords

regression testing, continuous integration (CI), continuous delivery (CD), test optimization, test prioritization, test minimization, incremental testing, testing efficiency, software quality assurance, test automation

1. Introduction

The relevance of CI/CD (Continuous Integration and Continuous Delivery/Deployment) [24] is increasing in the modern software development world, as these methodologies allow companies to release new product versions faster while ensuring high quality and stability. According to the **Accelerate State of DevOps Report 2021** [1], organizations that utilize CI/CD practices achieve a 3-4 times faster time-to-market for new features compared to those that do not adopt these methodologies. This enables companies to respond more quickly to market changes and customer needs. The same studies indicate that organizations implementing CI/CD reduce the number of defects reaching production by up to 50%. This decreases the cost of bug fixes and increases user satisfaction.

According to **GitLab's 2022 Global DevSecOps Survey** [2], 60% of teams that implemented CI/CD reported significant productivity gains. Automation of routine processes

*ITTAP'2024: 4th International Workshop on Information Technologies: Theoretical and Applied Problems, October 23-25, 2024, Ternopil, Ukraine, Opole, Poland

[1*] Corresponding author.

[†] These authors contributed equally.

dimaluk99@gmail.com (D.Lukavenko); delembovskyi.mm@knuba.edu.ua (Delembovskyi M.) 0009-0004-0834-9351 (D.Lukavenko); 0000-0002-6543-0701 (Delembovskyi M.)

and the reduction of manual work allow teams to focus on developing new features and innovations. **Gartner** predicts that by the end of 2025, over 70% of companies will use CI/CD to release software, maintaining a high update pace and ensuring continuous product development.

Regression testing plays a critically important role in the Continuous Integration and Delivery
(CI/CD) environment. In the context of CI/CD, where new code changes are continuously integrated into the shared codebase and frequently delivered to users, regression testing ensures the preservation of software functionality and stability. It is aimed at detecting defects that may arise after code changes are made. This prevents new errors from being introduced into the already tested code, which is crucial for maintaining system stability. Additionally, in CI/CD environments, rapid feedback is a necessity. Regression testing is automatically triggered after each code change, allowing developers to quickly identify potential issues. This ensures that new features do not disrupt existing functionality, maintaining high software quality even in a fastpaced development cycle.

By examining systematic literature reviews conducted on the topic of regression testing [3-5], it can be concluded that this is a well-researched area, with a significant number of methods proposed in the literature. Despite the extensive research in this field, the findings are not widely applied in practice. This can be attributed to several factors, such as differences in terminology, accessibility of research results, and the lack of empirical evaluation of methods in real-world conditions.

The results of this study will help understand which regression testing strategies provide the optimal balance between efficiency, defect detection, and resource expenditure, as well as how to best adapt testing to the needs of fast-paced CI/CD environments.

2. Main Research

Continuous testing and DevOps [6] are key elements of modern software development, enabling the integration of testing processes at all stages of the development lifecycle. The goal is to ensure rapid feedback, which helps accelerate the release of new versions without compromising quality. The architecture of the CI/CD model consists of three main components::

1. Preparation: In this phase, all necessary conditions and resources are provided to ensure the seamless operation of CI/CD processes. This includes conViguring the version control system (VCS), setting up development environments, conViguring the CI/CD infrastructure, and deVining testing strategies.
2. CI Pipeline (Continuous Integration Pipeline): At this stage, the code is built, regression testing is conducted, and code quality is analyzed. All test results are recorded, and in case of errors, developers are notiVied for corrections.
3. CD Pipeline (Continuous Delivery/Deployment Pipeline): The built and tested artifacts are automatically delivered to testing and staging environments, where deployment and additional testing are performed. After successfully passing the tests, the artifacts are deployed to production servers, followed by monitoring and testing to verify operational functionality.

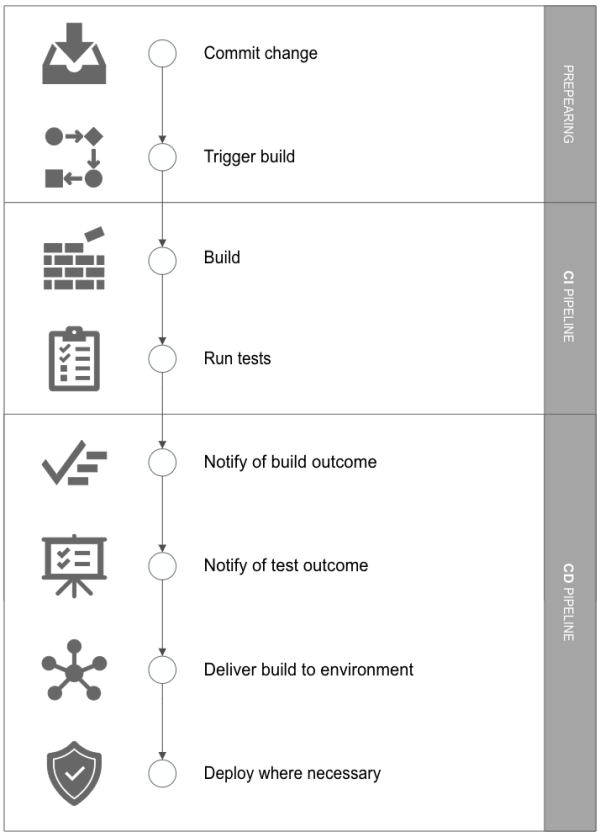


Figure 1: The Continuous Integration/Continuous Delivery architecture

Analyzing this information leads to the conclusion that testing at every stage of software development is critically necessary.

In this context, we compare full regression testing [6-8], incremental regression testing [9-10], test prioritization [11-12], and test minimization [13-14] based on execution time, defect detection, and impact on software quality. First, it is appropriate to clarify each of the methods.

Regression testing is the process of testing modiVied software to ensure its continuous quality. Typically, regression testing is conducted by reusing test scenarios developed during the testing of previous versions of the software system, as well as by creating new test scenarios to verify new functionality. As shown in Fig. 2, let D represent the original development that has been modiVied to create D', and let T represent the set of tests created for D. When transitioning from D to D', the program may regress, and its behavior may change. Regression testing is necessary to check for regressions in D'.

However, as the software grows, the number of test cases can become overwhelming, making the verification process too expensive, lengthy, and resource-intensive. Incremental regression testing selects test cases from T_{all} discarding those that do not test the modified code $T_{non\text{-}mod}$.

Test case prioritization arranges the cases in a sequence based on their importance or likelihood of detecting defects T_{all} to improve testing efficiency. Test minimization reduces the number of test cases to the minimal necessary set T_{all} by eliminating redundant checks. Full regression testing retains the maximum number of tests T_{all}, adding new test scenarios T_{new}.

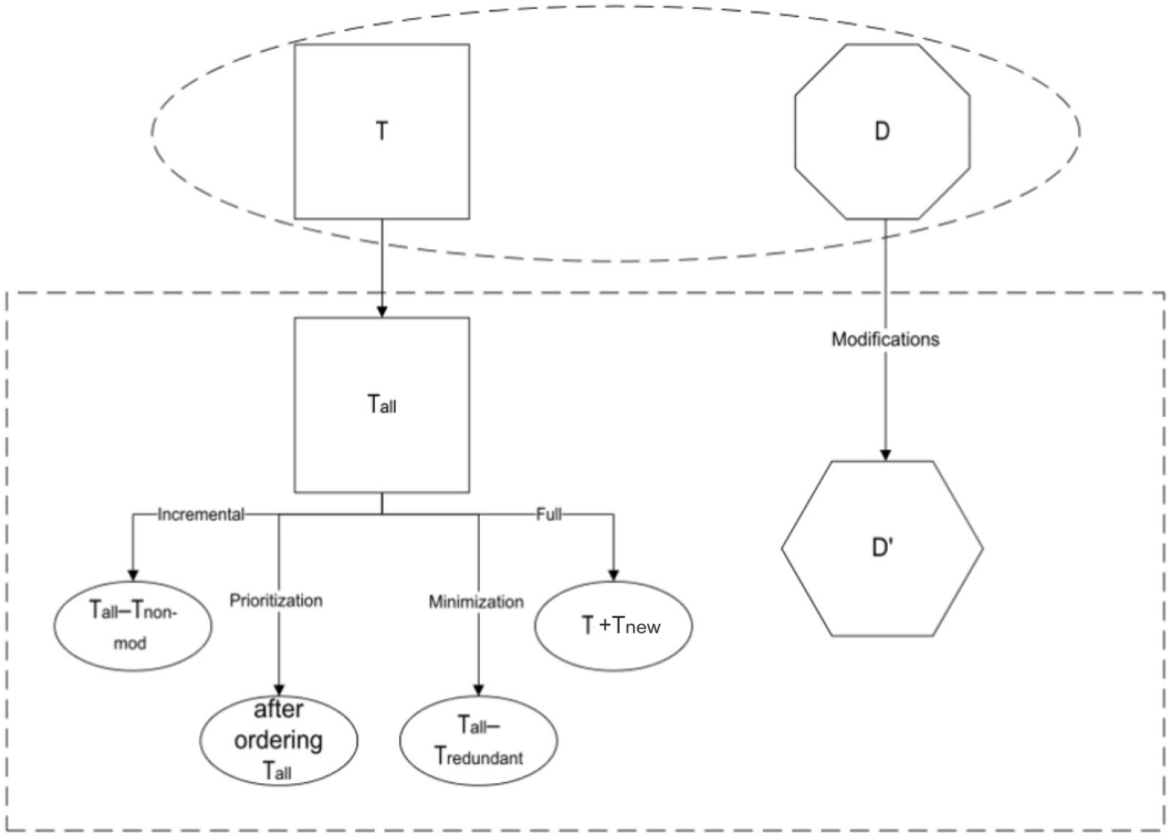
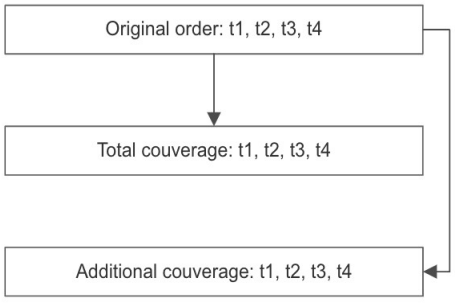


Figure 2: Regression testing techniques

1. **Full regression testing** involves running the entire set of tests every time changes are made to the codebase. This approach allows for the verification of all system functionalities, providing maximum confidence in the absence of regression errors. In the context of continuous testing and DevOps, full regression testing ensures that each new version of the software undergoes a complete cycle of checks, which is especially important for critical systems. For example, there are four test scenarios t1, t2, t3, and t4 (Table 1). In the case of full regression testing, all tests are executed without changing the order.

Table 1

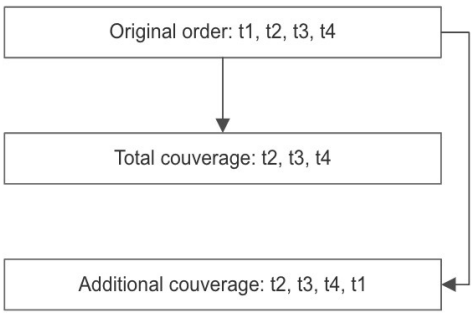
Blocks	t_1	t_2	t_3	t_4
1	x			
2		x		
3			x	
4	x	x		
5				x
6		x		x
7			x	



2. **Incremental regression testing** focuses on verifying only those parts of the code that have been modified or affected by changes. This approach significantly reduces testing time while maintaining the ability to detect errors in the modified parts of the system [15]. For example, there are four test scenarios t1, t2, t3, and t4. Changes were made in blocks 2, 5, 6, and 7. Using this method of regression testing, the priority is given to testing t2, t3, and t4 (Table 2). The test scenario t1 will either be tested last or may be skipped
altogether.

Table 2

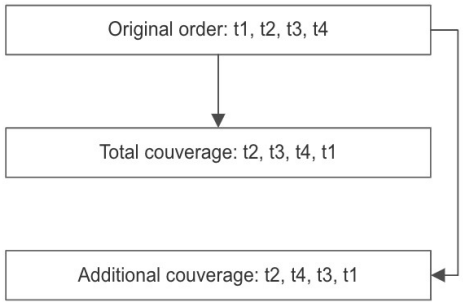
Blocks	t_1	t_2	t_3	t_4
1	x			
2			x	
3				x
4	x	x		
5		x		
6		x		
7				x
8	x		x	



3. **Test prioritization** involves ordering tests in such a way that the most critical or highrisk tests are executed first. This ensures the rapid detection of critical issues, which is particularly important in continuous testing environments [16]. For example, there are four test scenarios t1, t2, t3, and t4 (Table 3). Using test prioritization, t2 is tested first, followed by t3, t4, and t1. However, since t3 covers the same blocks as t2, its execution can be deferred to the end.

Table3

Blocks	t_1	t_2	t_3	t_4
1		x	x	
2				x
3		x		
4		x	x	x
5		x	x	
6				x
7		x	x	
8	x			

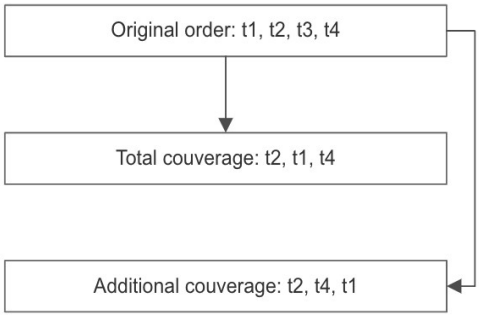


4. **Test minimization** aims to reduce the number of tests to the minimal necessary set that ensures effective defect detection. This approach significantly reduces the time and resources spent on testing, which is crucial in environments with frequent releases [17]. For example, there are four test scenarios t1, t2, t3, and t4 Table 4). In this case, scenario t2 covers a significant

portion of the system, t1 and t4 cover the areas missed by t2, while t3 repeats blocks already covered, so it is excluded from the testing process.

Table 4

Blocks	t_1	t_2	t_3	t_4
1	x			
2		x		
3		x	x	
4	x	x		
5				x
6				x
7		x	x	
8	x			



Having outlined the above-mentioned methods, it is appropriate to analyze the challenges of regression testing in a Continuous Integration and Delivery (CI/CD) environment [18], speciVically:

1. **Testing Duration**: Full regression testing can take a significant amount of time, especially in large projects with a vast number of test scenarios. This can slow down the deployment process and delay the release of updates. Even with parallel testing, the duration of tests can remain considerable, requiring additional resources. However, there are several strategies to mitigate these effects:
    - 1.1. **Selective Testing** [19]: Use change-based testing to run only those tests that are affected by code changes.
    - 1.2. **Parallel Testing** [20]: Execute tests in parallel on multiple machines or in containers to reduce the overall testing time.
    - 1.3. **Test Optimization:** Review and optimize existing tests by removing duplicates or combining tests that check similar functionalities.

2. **Test Updating**: Tests must be continuously updated to reflect changes in the codebase, which can be a labor-intensive process requiring significant effort from the testing team. Incorrect or unstable tests can lead to false positives or negatives, reducing the effectiveness of regression testing and requiring additional effort for analysis. There are several trade-offs to consider:
    - 2.1. **Automation of Test Updates**: Use tools to automatically update test scripts according to changes in the code. This may include using machine learning models to generate or update tests.
    - 2.2. **Monitoring and Analysis of Test Results**: Use tools to automatically analyze test results to quickly identify false positives or negatives and reduce the cost of manual verification.

3. **Resource Utilization**: Running a large number of tests requires significant computational resources, which can be problematic under a limited budget or infrastructure. Managing a

large volume of tests and results can be a challenging task, especially when using different tools and technologies. Common methods for balancing the load include:

3.1. **Using Cloud Services** [21]: Cloud platforms allow for scaling computational resources according to testing needs, reducing the load on local infrastructure.

3.2. **Containerization** [22]: Using containers (e.g., Docker) to isolate testing environments enables optimal resource utilization and ensures environment stability.

4. **Complexity of Test Selection:** Determining which tests to run after each code change can be a challenging task. An insufficiently thorough approach may lead to important defects being overlooked. Proper test prioritization can be complex and may require additional analytical data and tools. There are several methods for optimization:

4.1. **Risk-Based Testing** [23]: Prioritize tests based on the criticality of the functionality and the likelihood of defects occurring. This allows you to focus on the most important aspects of the system.

4.2. **Test Coverage Analysis**: Use code coverage analysis tools to identify which areas of the code are not being tested, and adjust the test suite accordingly.

5. **Automation:** Despite automation, not all aspects of regression testing can be easily automated, particularly when it comes to complex integration tests or user interface tests. A partial solution to this is:

5.1. **Improvement of Automation Tools:** Implementing more advanced automation tools that support complex testing scenarios, such as user interface testing or integration tests. This may include using specialized frameworks or AI-assisted tools to enhance automation.

6. **Instability of the Testing Environment:** Changes in the testing environment or system configuration can lead to unstable test results, making it difficult to identify real issues. By using the following tools, the risks of failure can be minimized:

6.1. **Prometheus** can automatically collect data on system status, resources, memory usage, CPU load, and other parameters. This helps quickly detect anomalies and instability in the environment.

6.2. **Using Grafana,** you can visually monitor the testing environment in real time, simplifying the detection and analysis of issues. Additionally, setting up alerts in Grafana allows for automatic notifications about critical changes or problems in the environment.

7. **Psychological Impact on the Team:** In CI/CD environments, where development and testing occur continuously, team members may experience significant stress due to the demands for rapid product releases and frequent updates. This can lead to burnout or decreased motivation if the team lacks support or if processes are poorly organized. Key issues include stress from the fast pace, false positive or negative results, feedback, and recognition. To improve this aspect of testing, it is necessary to:

7.1. Optimize Testing Processes: Parallel Test Execution: Running tests in parallel speeds up the CI/CD process and reduces delays, helping to avoid situations where the team waits for test results.

7.2. Reduce the Number of False Results: Analysis and Improvement of Test Quality: Regular audits of tests can help identify the causes of false positive or negative results. Some tests may need updating or removal if they yield unpredictable results. Also use of Stable Testing Environments: Ensuring a stable testing environment to avoid conflicts with external factors that may cause false results. This will make the tests more accurate.

7.3. Recognition and Incentives: Regular Recognition of Achievements: Evaluating and recognizing team accomplishments, even minor successes, contributes to motivation and morale. This can be through verbal acknowledgment in meetings or small awards or bonuses. Also recommended for use progressive Reward System: Implementing a reward system for achieving goals or completing tasks on time can enhance motivation. It is important that the system is fair and transparent.

These issues can significantly impact the quality and speed of software development and integration processes, making it essential to consider them when planning and organizing regression testing in CI/CD.

For a comprehensive evaluation of the effectiveness of regression testing methods, the Overall Efficiency Index was used. The metrics required for its calculation include:

1. **Coverage** Metric – This metric represents the proportion of code blocks tested by the test suite relative to the total number of blocks in the system:

$$\text{Block Coverage} = \frac{\text{Number of Covered Blocks}}{\text{Total Number of Blocks}}$$

The maximum value of the coverage metric is achieved with full regression testing, which equals 1 (100%).

2. **Defect Detection** Metric – This metric evaluates the effectiveness of an approach in identifying defects in the code and measures the percentage of detected defects relative to the total number of defects:

$$\text{Percentage of Detected Defects} = \frac{\text{Number of Detected Defects}}{\text{Total Number of Defects}}$$

3. **Execution Time** Metric – This metric represents the total time taken to execute the tests relative to their number:

$$\text{Average Execution Time} = \frac{\text{Total Execution Time}}{\text{Number of Tests}}$$

4. **Resource Utilization** Metric – This metric assesses the amount of computational resources, time, and effort required for the preparation and execution of tests, assuming that full regression testing consumes all resources and has a value of 1:

$$Recources\ Used = \frac{1}{Total\ Expenditure}$$

Once the values of each metric are obtained, the **Overall Efficiency Index** can be calculated as follows:

$$Overall\ Efficiency\ Index = \sum_{i=1}^{n} \left(\omega_i \times Metric \right)$$

Where ω_i – represents the weighting factor for each metric.

3. Results

To calculate the effectiveness of the testing methods using the Overall Efficiency Index, we need to determine the metric values for each method and establish the weighting coefficients for each metric.

Steps for Calculation

1. Selection of Metrics and Weighting Coefficients:
    - Block Coverage: $\omega_{\$}$=0.3
    - Percentage of Detected Defects: $\omega_{\%}$=0.4
    - Average Execution Time: $\omega_{\&}$=0.2
    - Resources Used: $\omega_{\cdot}$=0.1

Metric Values for Each Method (Table 5).

Table 5

Metric	Full Regression Testing	Incremental Testing	Test Minimization	Test Prioritization
Coverage Metric	1.0	0.75	0.6	0.85
Defect Detection Metric	0.95	0.85	0.7	0.9
Resource Utilization	1000	400	200	300
Average Execution Time	1.0	0.6	0.4	0.7

2. Next, it is necessary to normalize the values for accurate efficiency evaluation (Table 6), as the normalized execution time is inversely proportional to the total execution time. This means that shorter test execution times contribute more positively to the overall efficiency index.

$$\text{Normalized Execution Time} = \frac{1}{\text{Execution Time}}$$

The same principle applies to resource usage—normalized resource utilization is also inversely proportional to the amount of resources used. Lower resource usage increases the overall efficiency index.

$$\text{Normalized Resource Utilization} = \frac{1}{\text{Resources Used}}$$

Table 6

Metric	Full Regression Testing	Incremental Testing	Test Minimization	Test Prioritization
Coverage Metric	1.0	0.75	0.6	0.85
Defect Detection Metric	0.95	0.85	0.7	0.9
Resource Utilization	0.001	0.0025	0.005	0.00333
Average Execution Time	1.0	1.666	2.5	1.42857

3. Calculation of the Overall Efficiency Index for Each Method.

3.1. Full Regression Testing:

$0.3\times1.0+0.4\times0.95+0.2\times0.001+0.1\times1.0=0.3+0.38+0.0002+0.1=0.7802$ 3.2. Incremental Testing:

$0.3\times0.75+0.4\times0.85+0.2\times0.0025+0.1\times1.666=0.225+0.34+0.0005+0.1666=0.7321$ 3.3. Test Minimization:

$0.3\times0.6+0.4\times0.7+0.2\times0.005+0.1\times2.5=0.18+0.28+0.001+0.25=0.711$ 3.4. Test Prioritization:

$0.3\times0.85+0.4\times0.9+0.2\times0.00333+0.1\times1.42857=0.255+0.36+0.000666+0.142857=0.758523$

Results:

1. Full Regression Testing: Overall Efficiency Index = 0.7802
2. Incremental Testing: Overall Efficiency Index = 0.7321
3. Test Minimization: Overall Efficiency Index = 0.711
4. Test Prioritization: Overall Efficiency Index = 0.7585

Full regression testing remains the most effective method with an index of 0.7802. However, test prioritization has a very close index of 0.7585, making it an effective compromise between testing quality and resource expenditure. Incremental testing and test minimization have lower indices but can also be effective depending on the specific needs of the project, particularly for reducing time and resource usage.

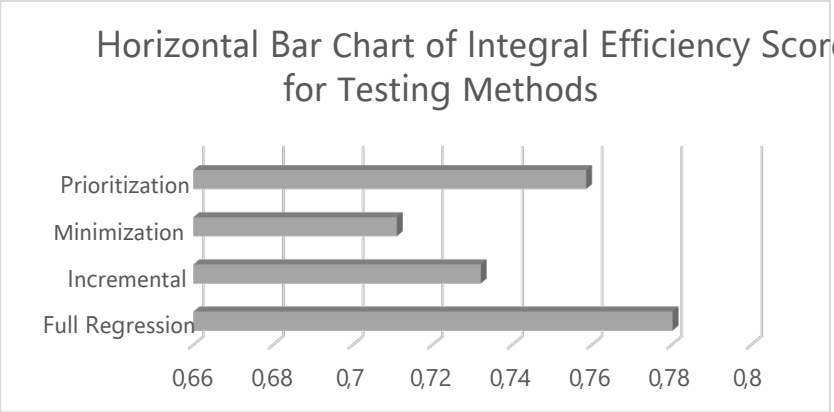


Figure 3: Comparison of research result

Conclusions

Summary of Experimental Results

1. **Full Regression Testing:** Provides the highest defect detection but is impractical for rapid feedback due to significant time consumption. It is most optimally used for:

a **Critical Systems**: Employed when it is crucial to ensure that every part of the system functions correctly, especially for systems with high security or stability requirements.

b **Infrequent Releases**: Used when new versions are released rarely, allowing sufVicient time for thorough testing.

c **After Major Changes**: Applied after signiVicant changes to the codebase to ensure that no functionality has been compromised.

This method also lowers the team's morale, as a significant amount of time is spent on complete verification, increasing the wait for results and feedback, despite a small number of false or negative positive results. Most often used in Web applications and mobile applications. Example: During a major release that includes significant changes to the application's architecture, the team needs to ensure that all features are functioning correctly.

2. **Incremental Regression Testing:** Balances efficiency and defect detection, making it suitable for CI/CD environments. It is advisable to use in the following cases:

    1. **Frequent Small Releases:** Used in CI/CD environments where frequent releases are the norm, and there is a need to quickly verify the most critical parts of the system.
    2. **Stable Code Areas:** When there are parts of the code that rarely change or are well-tested, selective testing allows focusing on the modiVied or new features.

3. **Limited Resources:** When there is a need to reduce testing costs while maintaining an acceptable level of veriVication.

Additionally, this method strikes a balance between the team's morale and the quality of testing, as checks and feedback are received relatively quickly with a moderate number of false or negative positive results. Most often used in Agile software development. Example: After making small code changes, such as updating a single module or adding a new feature, the team performs incremental regression testing to verify that the new changes have not affected the existing functionalities of the application.

3. **Test Prioritization:** Optimizes critical defect detection and efficiency, making it suitable for scenarios where quick feedback on the most important parts of the software is needed. It is effectively used in:

    1. **Critical Releases:** Employed when it is important to quickly obtain feedback on the status of the most crucial parts of the software.
    2. **Limited Time:** Used when testing needs to be conducted quickly, and there is no time to test the entire system.
    3. **Risky Changes:** Applied when changes are made to critical or high-risk parts of the code, and it is necessary to ensure that they do not introduce errors.

Regarding the morale aspect of the team, this testing method is similar to incremental testing. Most often used in systems with high availability requirements (e.g., financial or medical applications). Example: In projects where continuous operation is critically important, the team uses test prioritization to run the most critical tests first. This helps to quickly identify serious issues that could impact users or business processes.

4. **Test Minimization:** Reduces test execution time by limiting the number of tests to the minimum necessary set, while maintaining a sufficient level of defect detection. Despite lower defect detection efficiency, it is beneficial in the following cases:

    1. **Frequent Releases with Limited Time:** Used when regular releases need to be delivered with minimal delays, particularly in CI/CD environments.
    2. **Limited Testing Resources:** Applied when there are constraints on the use of computational or human resources for testing.
    3. **Testing of Stable Systems:** When the system is stable and well-tested, test minimization allows for resource conservation by focusing on key tests.

In terms of morale, the Minimization Testing method is the opposite of Full Regression Testing. Although the process occurs much faster and feedback is very quick, it has a strong negative impact due to the stress from the fast pace and a

higher number of false positive or negative results. Most often used in Systems with large data volumes or microservices architecture. Example: In projects with numerous unit tests, the team may use test minimization to focus on the most important tests related to the modified parts of the code. This allows them to reduce testing time and speed up the CI/CD process while maintaining an acceptable level of quality.

These strategies demonstrate trade-offs between thoroughness and efficiency. In CI/CD environments, Incremental and Prioritized Regression Testing are generally more suitable, offering a balance between comprehensive testing and the need for quick, continuous feedback.

The choice of testing method depends on the specific project requirements, available resources, risks, and release frequency. Proper use of these methods allows for the optimization of the regression testing process and ensures a balance between development speed and product quality.

Applying these conclusions will enable practitioners to select the most appropriate regression testing strategy based on their specific needs, resources, and the criticality of the software being developed.

References

[1] 2021 Accelerate State of DevOps report addresses burnout, team performance

[2] The GitLab 2022 Global DevSecOps Survey Thriving in an insecure world

[3] R. Kazmi, D. N. A. Jawawi, R. Mohamad, I. Ghani, Effective regression test case selection: A systematic literature review, ACM Comput. Surv. 50 (2) (2017) 29:1–29:32.

[4] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, R. Tumeng, Test case prior- itization approaches in regression testing: A systematic literature review, Information and Software Technology.

[5] S. U. R. Khan, S. P. Lee, N. Javaid, W. Abdul, A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guide- lines, IEEE Access 6 (2018) 11816– 11841.

[6] Vo ¨st S, Wagner S (2016) Trace-based test selection to support continuous integration in the automotive industry. In: Proceedings of the international workshop on continuous software evolution and delivery,CSED, pp 34–40

[7] Basu, Anirban (2015). Software Quality Assurance, Testing and Metrics. PHI Learning.

[8] ADVANCES INCOMPUTERSEdited byATIF M. MEMONCollege Park, MD, United States pp 5670

[9] Lity, S., Morbach, T., Thum, T., and Schaefer, I. (2016). Ap- ¨plying incremental model slicing to product-line regression testing. In International Conference on Software Reuse, pages 3–
19. Springer

[10] Lochau, M., Schaefer, I., Kamischke, J., and Lity, S. (2012). Incremental model-based testing of deltaoriented software product lines. In International Conference on Tests and Proofs, pages 67–82. Springer.

[11] Marijan D, Gotlieb A, Sen S (2013) Test case prioritization for continuous regression testing:

an industrialcase study. In: Proceedings of IEEE international conference on software maintenance, pp 540–543

[12] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). ACM, New York, NY, 975--980.

[13] B. Miranda and A. Bertolino, "Scope-aided test prioritization, selection and minimization for software reuse," J. Syst. Softw., vol. 131, pp. 528–549, 2017

[14] J. W. Lin, R. Jabbarvand, J. Garcia and S. Malek, "Nemo: Multi-criteria test-suite minimization with integer nonlinear programming," Proc.-Int. Conf. Softw. Eng., no. 1, pp. 1039–1049, 2018.

[15] Software Product Line Regression Testing: A Research Roadmap, Willian D. F. Mendonc, Wesley K. G. Assunc and Silvia R. Vergilio

[16] Automated code-based test selection for software product line regression testing: Pilsu Jung, Sungwon Kang, Jihyun Lee

[17] A learning algorithm for optimizing continuous integration development and testing practice: Dusica Marijan, Arnaud Gotlieb, Marius Liaaen

[18] A Survey on Regression Testing Using Nature-Inspired Approaches: Anu Bajaj, Om Prakash Sangwan (2019)

[19] S.A. Barylska, N.V. Zahorodna Ph.D. Assoc. Prof. REDUCTION-BASED METHODS AND METRICS FOR SELECTIVE REGRESSION TESTING

[20] Methods for improving the testing of digital systems (2023): V. Kardashuk, K. Bortnyk, N. Bahniuk

[21] Cloud computing technology: improving small business performance using the Internet (2018) Mohsen Attaran & Jeremy Woods

[22] Emerging Trends, Techniques and Open Issues of Containerization: Junzo Watada, Arunava Roy, Ruturaj Kadikar, Hoang Pham, Bing Xu (2019)

[23] Risk-Based Test Case Prioritization by Correlating System Methods and Their Associated Risk (2019): Hosney Jahan, Ziliang Feng, S. M. Hasan Mahmud

[24] CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study (2021): Fiorella Zampetti,Salvatore Geremia, Gabriele Bavota, Massimiliano Di Penta