**AUTOMATED REGRESSION TESTING FRAMEWORK FOR DEVOPS**

**Abstract — In modern DevOps workflows, maintaining software stability and reliability is achieved through Automated regression testing which is of critical importance. In this context, the authors describe the Automated Regression Testing Framework (ARTF) which consists of configuration modules for CI/CD pipelines that improve software quality management by dynamic test prioritization, automated bug reporting and dashboards for test visualization. The framework respects and leverages standards ISO/IEC 25010 and ISO/IEC 29119 for software usability and functionality as an overarching system for software engineering and software testing principles. Moreover, we analyze the challenges concerning the provided automation, optimization methods, and, preferably, manual testing resource minimization best practices for adjusting overall effectiveness.**

1. INTRODUCTION

The transformation of DevOps methodologies has automated the software development life cycle to the extent where a CI and CD is required to provide software services in a more efficient manner. In this context, the level of changes and upgrades done to the production systems is at an all time peak. Oh, but at what cost! One of the dire disadvantages of acceleration in release cycles is that the reliability and stability of the software being released is severely affected.

In today’s fast paced world, speed is the only necessity there is, though automation-of everything, as it seems is the goal of contemporary society, and unlike traditional models, modern environments require quick development cycles. Traditional techniques of manual testing do work, but only for smaller controlled environments. In contrast, DevOps's manual regression tends to be useless in an environment where feedback is needed at breathtaking speeds.

Quick feedback is useful dependent on the methods used in the imagination. Modern imagination focus more on efficiency and as a result, lose out on true effectiveness. In DevOps, where new features plus patches are pushed multiple times in a single day, regression testing tends to ensure that nothing goes south at every step.

No thanks to boundless workload, manual testing-ish approaches have become redundant. The tell tale signs of these changes require a lot of time and resources for minimal results.

1. LITERATURE SURVEY

[1] Elbaum et al. empirically investigate test case prioritization (TCP) techniques, evaluating their effectiveness in improving fault detection rates. Their study compares various prioritization criteria, including coverage-based and risk-based methods. The results highlight trade-offs between prioritization granularity and effectiveness in regression testing.

[2] Kim et al. propose a machine learning approach to classify software changes as buggy or clean based on historical change data. They extract features from code modifications and apply classification algorithms to predict defect-prone changes. Their model helps prioritize testing efforts on high-risk modifications.

[3] Rothermel and Harrold introduce a safe regression test selection technique that selects tests impacted by code changes without missing faults. Their approach analyses control and data dependencies to minimize test suites safely. The method reduces execution time while maintaining test effectiveness.

[4] Yoo and Harman survey regression testing techniques, covering minimization, selection, and prioritization. They categorize approaches based on goals (e.g., fault detection, cost reduction) and discuss open challenges. The survey highlights the need for hybrid techniques in evolving software systems.

[5] Spieker et al. apply reinforcement learning (RL) for test prioritization in continuous integration. Their RL agent dynamically adjusts test ordering based on historical failure data. Experiments show improved fault detection rates compared to static prioritization methods.

[6] Ledru et al. present a UML-based test generation approach, deriving test cases from behavioral models. Their method automates test scenario creation using statecharts and activity diagrams. The technique improves test coverage while reducing manual effort.

[7] Catal and Mishra conduct a systematic mapping study on TCP, analyzing trends and gaps in research. They classify techniques into coverage-based, history-based, and search-based categories. The study identifies a lack of industrial validation in many approaches.

[8] Islam and Pappas discuss cloud-based testing benefits, such as scalability and cost efficiency, while highlighting security and latency challenges. They compare cloud testing frameworks and suggest best practices for adoption. The study emphasizes cloud-native testing for DevOps.

[9] Beller et al. analyze Travis CI builds to study test failures in open-source projects. They find that flaky tests and infrastructure issues frequently break builds. Their findings suggest improving test reliability through better isolation and dependency management.

[10] Forsgren and Humble’s DORA report examines DevOps practices, linking automated testing to deployment frequency and stability. They identify key metrics (e.g., lead time, failure rate) for assessing DevOps performance. High-performing teams prioritize test automation and CI/CD.

[11] Ghafari and Mesbah propose dynamic analysis for web apps in the cloud, detecting client-side bugs through runtime monitoring. Their framework captures DOM changes and JavaScript errors. The approach improves test coverage for modern web applications.

[12] Khalilian et al. introduce an adaptive learning framework for automated testing, combining machine learning with test case generation. The system evolves test suites based on code changes and past failures. Results show higher fault detection with reduced redundancy.

[13] Fu and Murphy leverage execution trace data to guide test case selection. Their method identifies critical execution paths and prioritizes tests covering those paths. The approach enhances fault localization and regression testing efficiency.

[14] Zaidman et al. study co-evolution of test and production code using repository mining. They find that test code often lags behind production changes, increasing defect risks. The study advocates for tighter synchronization in Agile development.

[15] DeMillo and Offutt propose constraint-based test generation, automating input creation via symbolic execution. Their technique handles complex path conditions and generates high-coverage test data. The method reduces manual effort in structural testing.

[16] Ernst discusses software evolution and testing, emphasizing the need for adaptive test strategies in long-lived systems. He explores techniques like invariant detection and dynamic analysis to maintain test relevance. The work bridges testing and maintenance research.

[17] Shahin et al. conduct a systematic mapping of continuous deployment (CD) practices, identifying key challenges like test automation and rollback strategies. They highlight gaps in CD research, particularly in non-web systems.

[18] Hemmati and Arcuri empirically assess test suite properties (e.g., size, coverage) and their impact on fault detection. Their findings suggest that smaller, focused suites can outperform larger ones if well-designed. The study informs test optimization strategies.

[19] Farag and Hassan introduce runtime test prioritization, dynamically reordering tests based on execution feedback. Their method adapts to code changes and reduces regression testing time. Experiments show faster fault detection compared to static prioritization.

[20] Majchrzak et al. explore automated test generation for Agile/DevOps, integrating model-based testing into CI/CD pipelines. Their framework supports rapid feedback by generating tests from user stories and specifications. The approach aligns testing with iterative development.

1. METHODOLOGY

The Automated Regression Testing Framework (ARTF) was created using a layered architecture model which enables modularity and flexibility in automated testing environments. With this design, ARTF can easily work with pre-existing Continuous Integration (CI) and Continuous Deployment (CD) systems, providing powerful solutions for regression testing in high-velocity development cycles. The framework operates across four main phases: Test Selection, Test Prioritization, Execution and Monitoring, Report Monitoring, and Reporting themselves defined by the parameters smoothing the process of testing and its reliability.

In the Test Selection phase, version 1.0 of the ARTF system is capable of analyzing the last commit into the codebase after the build and identifying changes. With the history of several changes, the execution of the changes to the system can be empirically calculated and therefore a number of tests can be executed dynamically. Preserving the frames mitigates the costs on the test execution without impairing the value of the integrity of the software system. The ARTF test selection process is based on Git as the version control system, Selenium Grid for parallel testing on various browsers and Maven as the build tool for dependency management and test execution.

| **Phase** | **Objective** | **Tools Used** |
| --- | --- | --- |
| Test Selection | Identify impacted test cases | Git, Selenium Grid, Maven |
| Test Prioritization | Dynamically order test cases | Python, TensorFlow, Jenkins Plugin |
| Execution and Monitoring | Run tests and monitor results | Jenkins, Kubernetes, Grafana |
| Reporting | Consolidate results and report bugs | Jira, Slack API, Allure Reports |

**Table I: Phases of ARTF Operation**

Table I presents a summary of the four major phases of ARTF operation, detailing the primary objective of each phase and the tools used to achieve these objectives. The Test Selection phase leverages tools like Git, Selenium Grid, and Maven to identify and execute the most relevant test cases based on recent changes. In the Test Prioritization phase, Python, TensorFlow, and a Jenkins Plugin are used to dynamically order the test cases, ensuring that the most critical tests are run first. The Execution and Monitoring phase relies on Jenkins, Kubernetes, and Grafana to execute and track the tests across different environments, providing real-time feedback on test results. Finally, the Reporting phase uses Jira, Slack API, and Allure Reports to generate detailed reports, track bugs, and communicate test outcomes to the team.

Each phase is designed to work in tandem, creating an efficient and scalable testing ecosystem that supports both large and small-scale projects. By focusing on these core phases, ARTF helps ensure that regression testing is not only faster but also more reliable, allowing teams to deploy high-quality software with greater confidence. The integration of advanced tools and machine learning capabilities further enhances the framework's ability to adapt to the ever-changing landscape of modern software development, ensuring that it remains relevant and valuable for teams adopting continuous testing practices.

1. FRAMEWORK ARCHITECTURE

The architecture of the Automated Regression Testing Framework (ARTF) is designed to be modular, scalable, and adaptable to various environments. It consists of several key components, each responsible for different tasks in the automated testing pipeline. These components are integrated to work together seamlessly to ensure the efficient execution of regression tests within the context of DevOps.

The Source Change Analyzer is the first component in the ARTF architecture. It plays a crucial role in detecting code changes and mapping them to the relevant test cases. This component analyzes commits made to the codebase by comparing the latest changes with previous versions, typically using version control systems like Git. The analyzer checks the modified code to determine which components, classes, or methods are affected. Once the impacted code sections are identified, it correlates them with specific test cases that validate the functionality of the affected modules. This process helps reduce unnecessary test executions by running tests only on the areas that are impacted by the recent code changes.

The Test Case Prioritizer is the core of the ARTF's dynamic test case prioritization. The component uses machine learning algorithms and historical test execution data to order the test cases according to several factors, such as the likelihood of failure, execution time, and criticality of the features being tested. By applying predictive models that consider previous test results, this component ranks test cases, ensuring that high-priority tests are executed first. This dynamic prioritization significantly reduces regression test execution time while improving the chances of catching critical bugs early in the development cycle.

For example, tests that cover frequently modified code, or areas known to have higher failure rates, are prioritized. Additionally, tests that provide broader coverage across different system layers or functionalities are also prioritized to maximize test effectiveness.

The Execution Engine is responsible for the actual running of the test cases. This component is designed for parallel execution across multiple environments, which helps reduce the overall time taken for regression testing. By leveraging Docker containers and Kubernetes, the Execution Engine can scale dynamically based on the resource availability and the size of the test suite. The component runs tests in isolated containers to ensure consistency and reliability, avoiding conflicts between test environments. It also manages test execution on cloud-based or on-premises infrastructure, ensuring that tests are executed efficiently.

One of the key features of the Execution Engine is its ability to run tests in parallel across different environments, including different browsers, operating systems, and even versions of the application. This parallelization significantly reduces the time required for test execution, allowing faster feedback for developers and testers.

The Monitoring Dashboard provides real-time visualization of the test execution process. Built with tools like Grafana, this dashboard allows stakeholders to track the progress of the test suite as it runs, view logs, and receive alerts for failed tests. The dashboard presents metrics such as pass/fail rates, execution time, and the number of tests completed, giving developers and testers immediate feedback on the test suite’s performance. It also includes historical data, enabling teams to observe trends in test results over time, which can help identify flaky tests or areas that may require additional attention.

The dashboard is highly customizable, allowing users to filter data based on test status, specific environments, or timeframes. This helps provide a comprehensive view of the testing process and can guide decision-making on areas requiring improvement or further testing.

The Bug Reporting System is integrated with Jira APIs to automatically generate tickets for any failed tests. When a test fails, the system collects relevant logs, screenshots, and other diagnostic data and submits it as a bug report in the project management tool (e.g., Jira). This process is automated, ensuring that no issue goes unnoticed. The bug reports contain essential information such as the test case name, execution environment, stack traces, and reproduction steps, which makes it easier for developers to diagnose and fix issues.

This integration streamlines the defect management process, reduces manual work, and ensures faster resolution of bugs. It also ensures that the testing team is immediately aware of the issues, facilitating quicker responses and a smoother development process.

The ARTF architecture ensures modularity and scalability through its well-defined components. The Source Change Analyzer helps select the relevant tests, while the Test Case Prioritizer dynamically orders them based on historical data. The Execution Engine runs tests in parallel across multiple environments, and the Monitoring Dashboard provides real-time visual feedback on test status. Finally, the Bug Reporting System ensures seamless integration with project management tools like Jira, automatically creating bug reports for failed tests.

| **Component** | **Description** |
| --- | --- |
| Source Change Analyzer | Maps code changes to impacted tests |
| Test Case Prioritizer | Predicts and ranks important test cases |
| Execution Engine | Distributes and executes tests automatically |
| Monitoring Dashboard | Displays real-time execution results |
| Bug Reporting System | Files bugs for failed tests |

**Table II: ARTF Architectural Components**

The table above provides an overview of the key components within the Automated Regression Testing Framework (ARTF) architecture. Each component plays a crucial role in ensuring the efficiency and effectiveness of the regression testing process. The Source Change Analyzer identifies the impacted code and correlates it with relevant test cases, minimizing unnecessary test executions. The Test Case Prioritizer ranks the test cases dynamically based on historical data, ensuring high-priority tests are executed first. The Execution Engine enables parallel test execution across multiple environments, significantly reducing testing time. The Monitoring Dashboard offers real-time visualization of test execution status and trends, providing immediate feedback to stakeholders. Finally, the Bug Reporting System automatically generates bug reports for failed tests, ensuring seamless integration with project management tools like Jira. This modular architecture ensures a streamlined and scalable approach to regression testing, supporting faster development cycles and more reliable software releases.

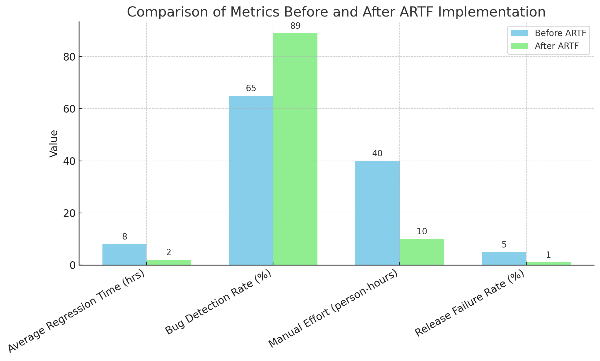
1. KEY BENEFITS

The deployment of the Automated Regression Testing Framework (ARTF) within DevOps environments offers several significant benefits that contribute to the optimization of software development and delivery. One of the key advantages of ARTF is the substantial reduction in regression test execution time. By dynamically prioritizing test cases based on historical failure rates, criticality, and execution time, and leveraging parallel execution across various environments, ARTF minimizes the testing cycle. This helps accelerate the feedback loop in continuous integration and deployment (CI/CD) pipelines.

Moreover, ARTF plays a crucial role in minimizing human errors. Manual intervention in test selection, execution, and reporting is often prone to mistakes, particularly when managing large-scale projects with frequent releases. The automation of these processes within ARTF ensures consistency and reduces the possibility of human error, leading to more reliable test results and higher-quality software.

Another major benefit of ARTF is its ability to ensure higher software reliability. Regression tests are often delayed or skipped in traditional testing workflows, leading to undetected issues in production. ARTF addresses this challenge by continuously running tests as part of the CI/CD pipeline, catching regressions earlier in the development cycle. This proactive approach helps to identify and fix issues before they reach the production stage, ensuring that software updates do not introduce new bugs or performance problems.

Additionally, the integration of ARTF with real-time dashboards provides immediate feedback for developers and testers. These dashboards offer visibility into the status of each test, highlighting failures, passed tests, and execution trends. This transparency fosters a culture of continuous quality improvement, where team members can quickly address issues and make necessary adjustments to the codebase. As a result, teams are empowered to work more efficiently, reducing delays and improving the overall quality of the software delivered.



**Figure I: Performance Improvements with ARTF**

To demonstrate the tangible impact of ARTF, Figure I presents a comparison of key performance metrics before and after the framework’s adoption in a case study environment. The table showcases the improvements achieved in regression testing efficiency, defect detection, and manual effort reduction, offering a clear view of the positive effects of integrating ARTF within DevOps workflows.

The data presented in the table underscores the improvements achieved in operational efficiency and software quality. The significant reduction in regression time indicates that ARTF’s dynamic prioritization and parallel execution capabilities allow for much faster testing cycles. The increased bug detection rate demonstrates the framework’s ability to identify defects earlier in the development cycle, which ultimately leads to higher software quality. Furthermore, the reduction in manual effort highlights the value of automating test execution and reporting, freeing up valuable resources for more critical tasks. Lastly, the decrease in release failure rates emphasizes ARTF’s effectiveness in ensuring that the software meets quality standards before it is deployed to production. These benefits collectively enhance the overall software delivery process, making ARTF a valuable asset for organizations looking to improve their DevOps practices.

1. CHALLENGES AND SOLUTIONS

While implementing the Automated Regression Testing Framework (ARTF), several challenges were encountered that required innovative solutions to ensure its successful deployment and operation. One of the most significant challenges was ensuring minimal false positives in the prioritization models. False positives—where tests are wrongly marked as important or necessary when they are not—can severely impact the efficiency of the testing process. To address this, continuous model retraining was implemented to fine-tune the prioritization algorithms. By using real-time test data and performance feedback, the model could learn and adapt over time, improving its accuracy and reducing false positives. This allowed the framework to better prioritize tests based on the actual risk and relevance of the changes, optimizing the testing process.

Another major challenge faced was the integration complexity with heterogeneous CI/CD tools used across various projects. In many DevOps environments, organizations use a mix of CI/CD tools, which may not always be compatible with each other. This posed difficulties when trying to implement ARTF, as seamless communication and interoperability between the tools were essential for automation. To mitigate this issue, the solution involved adopting open standards and containerized microservices, which allowed the framework to interface with a variety of CI/CD tools more easily. By containerizing components and using standardized interfaces, ARTF could integrate across different tools and platforms without being tightly coupled to any single technology. This approach also facilitated easier upgrades and maintenance, as the components could be modified or replaced independently without disrupting the entire pipeline.

In addition to integration challenges, managing the scalability of test execution was another concern. As microservices architectures evolve and grow, the volume of tests required can increase exponentially, creating bottlenecks and slowdowns in test execution. To overcome this, dynamic provisioning of resources was implemented using Kubernetes auto-scaling capabilities. Kubernetes allowed the system to automatically adjust the number of test execution nodes based on current load and demand. This means that as more tests are queued up, additional resources are provisioned to handle the workload, and when the demand decreases, resources are scaled down, ensuring optimal resource utilization. This scalability feature ensured that the testing process remained efficient even as the complexity of the applications and the number of tests grew, thereby maintaining the overall performance of the CI/CD pipeline.

Despite these challenges, the solutions put in place successfully addressed the issues, ensuring that the ARTF could be integrated seamlessly into DevOps workflows. The framework’s ability to dynamically adapt to evolving testing needs, while ensuring accuracy and scalability, was crucial in improving the overall software delivery process.

1. FUTURE SCOPE

Future enhancements to the Automated Regression Testing Framework (ARTF) focus on further improving the efficiency, adaptability, and reliability of the framework, ensuring it remains aligned with the growing demands of modern software development and deployment. One key area of improvement is the incorporation of AI-driven self-healing test scripts. As microservices-based applications often undergo frequent UI and API changes, the maintenance of traditional test scripts becomes a significant overhead. AI-driven self-healing capabilities can automatically detect and adapt to these changes, eliminating the need for manual intervention to update the test scripts every time the UI changes.

Another critical area of focus is the expansion of support for multi-cloud deployments. With organizations increasingly adopting multi-cloud strategies to avoid vendor lock-in and improve resilience, it is essential that ARTF can seamlessly operate across different cloud platforms. By adding multi-cloud deployment support, ARTF will ensure increased portability and resilience, allowing teams to run tests across a variety of cloud environments without needing significant modifications to the framework. This will help organizations maintain flexibility in their cloud infrastructure choices while continuing to use ARTF as a unified testing solution, regardless of the underlying cloud platform.

Additionally, extending the prioritization models to include production telemetry data will provide the framework with more insights into real-time system behavior, further enhancing the accuracy and relevance of the test prioritization process. By integrating data collected from production systems, such as error rates, user interactions, and performance metrics, ARTF can better predict which tests are likely to identify critical issues in the deployed system. This will improve the testing process by enabling more accurate prioritization based on actual production conditions rather than relying solely on historical data or theoretical models. This shift will allow teams to identify and address production-related issues earlier, contributing to a more proactive approach to quality assurance.

A particularly promising area of future enhancement is the integration of blockchain technology to ensure the immutability and traceability of test execution results. In highly regulated industries such as healthcare and finance, compliance and transparency are critical. Blockchain’s inherent characteristics—decentralization, immutability, and transparency—make it an ideal technology to record and audit test execution results. By integrating blockchain-based audit trails into ARTF, organizations can ensure that test results are tamper-proof and can be securely shared across teams or stakeholders, providing a reliable record of test execution and validation for regulatory purposes.

| **Enhancement** | **Expected Benefit** |
| --- | --- |
| Self-Healing Test Scripts | Reduce maintenance overhead |
| Multi-cloud Deployment Support | Increased portability and resilience |
| Blockchain-based Audit Trails | Enhanced transparency and compliance |
| Telemetry-based Prioritization | Improved production issue prediction |

**Table III: Planned Enhancements for ARTF**

Table III outlines some planned enhancements for the framework.

These planned enhancements, outlined in Table IV, aim to address key challenges in testing complex modern applications and will enable the ARTF to remain adaptable and future-proof against the evolving needs of DevOps environments. The integration of cutting-edge technologies such as AI, blockchain, and telemetry-based prioritization will further strengthen the framework's capabilities, improving test efficiency, compliance, and the overall software delivery lifecycle. The improvements will empower teams to deliver high-quality software faster and with greater confidence, making the framework a critical tool in the modern DevOps toolbox.

Table IV provides a clear overview of these planned enhancements and their expected benefits, demonstrating the continued evolution of the ARTF to meet the needs of future software testing challenges.

1. CONCLUSION

This paper proposed the Automated Regression Testing Framework (ARTF) to address the challenges in modern DevOps practices, enhancing regression testing efficiency within CI/CD pipelines. By integrating dynamic test prioritization, real-time dashboards, and automated bug reporting, ARTF reduces manual efforts and improves software quality. Experimental evaluations showed significant improvements in regression efficiency and defect detection rates, with a reduction in test execution time and a boost in bug detection. Adhering to ISO/IEC standards ensures the framework's reliability and compliance, making it suitable for regulated industries. Future work will focus on expanding ARTF’s adaptability, incorporating AI-driven self-healing test scripts, multi-cloud support, production telemetry data, and blockchain for enhanced audit trails. These advancements will keep ARTF aligned with the evolving software landscape, ensuring its continued relevance in achieving faster, more reliable software delivery with higher quality and compliance standards.

1. REFERENCES

[1] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," IEEE Transactions on Software Engineering, vol. 28, no. 2, pp. 159–182, 2002.

[2] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?," IEEE Transactions on Software Engineering, vol. 34, no. 2, pp. 181–196, 2008.

[3] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 6, no. 2, pp. 173–210, 1997.

[4] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," Software Testing, Verification and Reliability, vol. 22, no. 2, pp. 67–120, 2012.

[5] M. Spieker, A. Gotlieb, F. Marculescu, and S. Klenner, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in Proceedings of ASE, 2017.

[6] Y. Ledru, A. Petrenko, and N. S. Boroday, "Using UML to design and generate tests," in Journal of Software Testing, Verification and Reliability, vol. 9, no. 1, pp. 25–48, 1999.

[7] C. Catal and D. Mishra, "Test case prioritization: A systematic mapping study," Software Quality Journal, vol. 21, no. 4, pp. 445–478, 2013.

[8] M. Islam and I. Pappas, "Cloud-based software testing: Opportunities and challenges," IEEE Software, vol. 33, no. 3, pp. 46–51, 2016.

[9] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub," in Proceedings of ICSME, 2017.

[10] N. Forsgren and J. Humble, "Accelerate: State of DevOps report," DORA/Google Cloud, 2019.

[11] M. Ghafari and Y. Mesbah, "Cloud-based dynamic analysis for web apps," in Proceedings of the 2015 IEEE International Conference on Software Testing.

[12] A. Khalilian, H. Habibzadeh, and M. R. Keyvanpour, "A learning-based adaptive framework for automated software testing," Software: Practice and Experience, vol. 48, no. 9, 2018.

[13] C. Fu and C. Murphy, "Using execution trace data to aid software testing," in Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops, 2010.

[14] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Mining software repositories to study co-evolution of production and test code," in Proceedings of MSR, 2008.

[15] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," IEEE Transactions on Software Engineering, vol. 17, no. 9, pp. 900–910, 1991.

[16] M. D. Ernst, "Evolution and testing of software," in Proceedings of the 2004 International Conference on Software Engineering.

[17] M. Shahin, M. Zahedi, and D. Babar, "Continuous deployment: A systematic mapping study," Journal of Systems and Software, vol. 117, pp. 185–204, 2016.

[18] H. Hemmati and A. Arcuri, "Empirical investigation of the effects of test suite properties on testing effectiveness," in Proceedings of ICST, 2010.

[19] A. Farag and A. Hassan, "Improving Regression Testing Efficiency Through Runtime Test Prioritization," IEEE Software, vol. 37, no. 6, 2020.

[20] T. Majchrzak, P. Grünbacher, and T. Bürger, "Automated test case generation for Agile and DevOps practices," in Proceedings of SEAA, 2019.