International Journal of Management, IT & Engineering
Vol. 16 Issue 06, June 2026,
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gate as well as in Cabell's Directories of Publishing Opportunities, U.S.A

AI-Based Regression Testing in Agile Software Development

Kiran Paul Kanikaram
Principal Engineer, Testing, Majesco, USA

Abstract

The advent of artificial intelligence (AI) has revolutionized software testing, especially in the context of Agile development, where fast-paced and iterative releases push the limits of traditional quality assurance strategies. One of the most resource-consuming tasks in the software lifecycle is regression testing, which ensures that recent changes to software don't break more established features. Frequent commits in code and short sprint cycles in Agile methodology make it very costly in terms of time and resources to run the full regression suite.

This central research problem is the lack of efficiency of traditional retest-all and manually maintained regression approach that cannot keep pace with the fast-paced Agile delivery and often block continuous integration and continuous deployment (CI/CD) pipelines. This study aims to develop an AI-driven regression testing framework that intelligently selects, prioritises and optimises regression test cases; to do empirical evaluation of the developed framework with traditional approaches; and to investigate whether the proposed framework has a positive effect on the Agile sprint. In methodological aspects, it is a mixed method quantitative and experimental study that utilizes five software projects representative of the industry, adopts machine learning, deep learning and natural language processing models and applies it to a historical version control and defect data.

The key results showed that the proposed framework was able to reduce the size of the regression test suite by 61% on average, decrease the test suite execution time by up to 65% across sprints, and achieve above 91% in defect detection with accuracy; the transformer-based model was able to achieve an F1-score of 0.925. Overall, the study highlights that while AI-powered regression testing holds considerable promise as a tool for enhancing Agile software quality assurance, it also presents some potential obstacles and areas for further research, such as addressing data quality, explainability, and tool integration.

Keywords: Artificial Intelligence; Regression Testing; Agile Software Development; Machine Learning; Test Case Prioritisation; Continuous Integration; Defect Prediction

Introduction

Agile software development came as a reaction to the inflexibility of plan-driven, sequential approaches that were unable to deal with the uncertainty of requirements and changing market conditions. Its methods are based on the tenets of the Agile Manifesto, which says that, above all else, individuals and interactions are more important than processes and tools, working software is more important than comprehensive documentation, customer collaboration is more important than contract negotiations and responsiveness to change is more important than following a plan. Scrum, Kanban and Extreme Programming are frameworks that are built around short time-boxed cycles—also known as sprints—that result in a potentially shippable increment. This cadence allows teams to receive feedback on a regular basis, and to provide value in small increments, instead of waiting for the end of a long release cycle.

Its frequent and incremental change is the hallmark of Agile and this puts constant pressure on quality assurance. Every commit can change common modules, or create a subtle bug or break something that was once working. With the adoption of DevOps and CI/CD pipelines, it has become a basic requirement that every code change be automatically validated in minutes (Dakhel et al., 2023). Therefore, testing needs to match the pace of development speed, and the speed of the testing would directly affect the speed of value delivery to users, if it is lagging behind.

Concept of Regression Testing

Regression testing is the process of retesting the already tested test case after some changes in the code to ensure that previously verified behavior is not affected. It's not intended to find any new functionality defects, but rather to make sure that changes (bug fixes, enhancements, refactoring, etc.) have not created any unintended side effects. The Naive Approach (Retest-All) performs the complete set of tests each time the system changes and ensures complete coverage, but at a high cost. With thousand of cases in the suites, retest-all is no longer sustainable in the sprints (Pan et al., 2022).

In order to counteract this, the literature presents three complementary strategies: regression test selection, which selects a subset of tests that are relevant to the changes; test suite minimisation, which removes redundant tests; and test case prioritisation, which prioritises the tests so that the ones most likely to reveal a fault are run first. Both of these strategies sacrifice some level of thoroughness for efficiency, with the central problem being to lower costs without decreasing fault-detecting power (Khatibsyarbini et al., 2018).

Evolution of Artificial Intelligence in Software Testing

AI has gradually made the transition from research to business as usual software engineering. While early applications focused on heuristic search and rule-based systems, the recent progress of machine learning, deep learning and natural language processing has opened the door to a variety of testing tasks that can be automated (Durelli et al., 2019). The supervised learning approach is used to predict defect-prone modules, the reinforcement learning approach is utilized to explore the application states and the transformer-based language model is used to generate and repair test artefacts. This shift has speeded up significantly since 2016, when there is more data than version control, cheap computation, and well-established open-source toolkits.

2.4 Need for AI-Based Regression Testing

The combination of Agile velocity and regression cost gives rise to a structural issue for which traditional approaches can only be partial. Heuristic selection rules are fragile, need to be manually updated and become less effective as systems change. An adaptive solution is provided by AI models; learn code change defects, historical failures and locality and generalise to unseen changes (Lima et al., 2020). AI-powered regression testing can significantly reduce the number of tests that are run, without reducing the number of tests that fail, while keeping pace with continuous delivery.

Even with the promise of intelligent automation, organisations using Agile development still use coarse, manually managed selection rules—or retest-all—with poor scaling. Such methods take up a lot of resources, slow down the computer, and extend the execution time without providing feedback in a timely fashion. The lack of an end-to-end AI-driven regression framework that reliably and systematically selects, prioritises, and optimises test cases in Agile pipelines is the core research problem.

Although there are numerous research papers studying the single use of AI for test prioritisation and defect prediction, only a few studies have explored the combined use of selection, prioritisation, optimisation, and continuous learning within a single framework in realistic Agile sprint scenarios (Pan et al., 2022). Many of these studies are only limited to reporting individual metrics on synthetic datasets, with limited discussion on how impactful they are on sprint level, model maintainability, and explainability. The aim of this study is to fill that gap.

The study is a functional regression testing of mid-size and large web and enterprise applications developed using Scrum based Agile process. It takes source-code, version-control and defect-tracking data as inputs to the model. Other classes of non-functional testing, like performance test, security penetration test, and usability testing, are not within the current scope of the present framework but the principles of the framework may be extended to these in future work.

Literature Review

The new ways of working with quality implemented in the development process through agile practice like continuous integration, test driven development, pair programming has changed the way quality was integrated. For continuous integration, for instance, code needs to be merged and verified many times a day, and automated testing is a must. The study reveals that the agile success relies on having a fast feedback mechanism and that a slow verification of the products will mean that the agile responsiveness is lost (Dakhel et al., 2023). One of the key findings of the literature is that, in established Agile pipelines, regression testing is the main culprit of feedback latency.

The families of classical regression techniques are selection, minimisation and prioritisation. While selection techniques like dynamic/static slicing find tests that are affected by modified code, minimisation eliminates redundant tests according to coverage requirements. Prioritisation is a process prioritizing tests according to their coverage based or history based heuristics, which means that it will show faults earlier in the process (Khatibsyarbini et al., 2018). These heuristics are valid for stable systems, but could be violated in fast-changing systems due to locality of change and coverage stability assumptions (Yoo & Harman, 2012).

Systematic mapping of AI in software testing reveals a diversity of AI techniques including search-based optimisation, classification, clustering and reinforcement learning (Durelli et al., 2019). These techniques have been used for testing generation, Oracle construction, Fault localization and Fault prioritization. The evidences explored show that the learning model can have a better performance as compared to the static heuristic only when a large quantity of historical data exists, but it can be sensitive to the feature engineering and data quality.

Supervised classifiers (such as decision tree, random forest or support vector machine) have been trained to predict whether or not a test fails for a given change set. The empirical study shows that ensemble techniques can always be accurate and that test selection criteria for fault revelation has a high level of performance and that they can reduce the test suite size significantly, while keeping high level of fault detection capability percentage (Lima et al., 2020). Typically, sets of usage features are comprised of a combination of code-churn metrics and past failure rate and coupling metrics.

There are time dependencies that can be captured in test-execution histories which cannot be captured by shallow models but can be captured by deep neural models, especially recurrent and long short-term memory (LSTM) networks. Some works in the field of using deep learning methods for flaky-test detection and failure prediction have shown significant improvements in accuracy, reasoning about the deep learning models' ability to capture complex non-linear patterns in sequential build data (Pan et al., 2022). But the benefit of these gains is that they are accompanied with increased training cost and decreased interpretability.

It is possible to automate test case generation from requirements, user stories and bug reports with the help of NLP. Transformers from natural language to executable test skeletons, and defect report to test artefacts (Wang et al. 2024). More recently, the generation of unit tests was also automated via large-language-model (LLM) methods, and correctness and oracle validity are subjects of research.

Predictive analytics analysis is about determining the probable region of defect appearance from historical data (defects, changes etc.). Teams can focus regression effort on the high-risk areas by modelling the probability of failure per module. History-based predictive models are demonstrated to be helpful in providing meaningful early fault detection among models based on coverage-only heuristics, especially for long-lived projects with rich change histories (Bertolino et al., 2020), as illustrated in the literature.

AI based prioritisation and prioritizing reorder test suites for the maximum possible early fault detection, often expressed as Average Percentage of Faults Detected (APFD). Prioritisation is now an important paradigm and can be viewed as a sequential decision problem with a learning agent that learns an ordering policy, given feedback continually. (Spieker et al., 2017). These adaptive methods beat static ordering in a CI setting where the optimal ordering of the orders may change over time with changes in the systems.

Defect prediction models are based on product and process metrics, and are used to predict the fault proneness of the software components. According to a comparative evaluation, ensemble and deep models are likely to better outperform simple baselines, however, data imbalance and cross-project generalisation still present significant challenges (Hosseini et al., 2019). A good defect prediction can directly help in the regression process in terms of selection and prioritisation.

Based on the work reviewed, this synthesis will show that there are three gaps that reoccur. Second, most studies have been dealing with the optimisation of one of the three phases (selection, prioritisation, prediction) without considering them as a pipeline for a lifelong learning system. Second, evaluation is often done on benchmark datasets that are independent of the actual Agile sprint dynamics, thus causing a lack of external validity. Thirdly, explainability and maintainability of deployed models are under-studied, yet are crucial for industrial adoption (Pan et al., 2022). The current study aims to directly tackle those gaps.

K. P. Kanikaram (2025) examined the transformative role of artificial intelligence (AI) in software quality assurance by integrating automated testing and intelligent software analytics. The study highlighted that AI-driven testing frameworks improve defect detection, minimise manual intervention, and accelerate software development cycles. Intelligent analytics were found to enhance predictive maintenance, risk assessment, and decision-making by identifying patterns in software performance and quality metrics. The research concluded that AI-based quality assurance significantly increases testing efficiency, software reliability, and overall product quality while supporting continuous integration and agile software development practices in modern software engineering environments.

Research Methodology

Research Design

The research has a mixed method experimental research design – quantitative model evaluation and comparative case analysis. The design is a controlled empirical study approach: Historical data from real projects are split into training and testing periods; AI models are trained on training period and tested on testing period, and compared with the traditional retest-all approach with the same set of faults. This design makes it possible to make causal inferences about the effect of the framework on the suite size, execution time and detection accuracy.

Data Collection Method

The data was collected from the version-control repositories, the continuous-integration build logs and defect-tracking systems of each project involved. For each code change, the sets of modified files, the set of code-churn metrics, the set of test cases that were executed, and the set of their pass/fail results were noted, as well as the defects subsequently associated with the code change. This longitudinal data will serve as the supervised learning signal relating changes to test failures.

4.3 Selection of Case Studies/Projects

Five software projects (Proj-A to Proj-E) were chosen to cover a range of different domains and sizes, both ecommerce, ERP and SaaS applications. The requirement for each project was that it had to use an Agile process (specifically, Scrum), have at least eighteen months of version history and have automated regression suites with over seven hundred test cases. Such diversity allows the generalisability of the findings.

AI Models Considered

A total of five families of models were tested, including two interpretable baseline models (a decision tree and random forest), a support vector machine (classical margin-based model), an LSTM network (sequential model for build data) and a fine tuned BERT

(transformer) for change-and-report text. This set provides the opportunity to perform a well-rounded comparison between the classical, deep and language-model approaches that are interpretable.

Performance Evaluation Metrics

Metric	Definition	Purpose
Suite Size Reduction	Percentage decrease in executed test cases	Measures efficiency gain
Execution Time	Wall-clock minutes to run the selected suite	Measures feedback speed
Precision	Correct failure predictions over all predicted failures	Controls false alarms
Recall	Correct failure predictions over all actual failures	Controls missed faults
F1-Score	Harmonic mean of precision and recall	Balanced effectiveness
Accuracy	Correct predictions over all predictions	Overall correctness
APFD	Average Percentage of Faults Detected	Prioritisation quality

Table 1. Performance evaluation metrics used in the study.

Data Analysis Techniques

The results of the quantitative analysis were evaluated with the conventional supervised learning evaluation strategy based on a stratified cross validation and the hold-out testing approach to simulate deployment on the next period after training. Differences between the AI strategies and the traditional strategies were determined at every sprint and tested for statistical significance. Confusion-matrix derived metrics were used to evaluate the classification quality and the success of prioritisation was evaluated by APFD.

AI-Based Regression Testing Framework

Architecture of the Proposed Framework

We propose a modular and pipeline style implementation that takes a list of change events from the version-control system as input, and outputs a prioritised, reduced regression suite to run in the CI/CD pipeline. It has 7 cooperating stages: data preprocessing, feature extraction, training the model, classification of test cases, optimisation of the test suite, running the tests automatically, and a loop for continuous learning. Each stage is loosely coupled to enable replaceable components in the pipeline, e.g. underlying AI model.

Stage	Input	Output
Data Pre-processing	Raw commits, logs, defects	Cleaned, labelled records
Feature Extraction	Cleaned records	Numerical feature vectors
Model Training	Feature vectors (historical)	Trained predictive model
Test Classification	Change features	Fail-probability per test
Suite Optimisation	Scored tests	Reduced, ordered suite
Automated Execution	Optimised suite	Pass/fail results
Feedback Loop	Execution results	Updated training data

Table 2. Stages of the proposed AI-based regression testing framework.

Data Pre-processing

Raw data from various sources have noisy, incomplete and inconsistently formatted data. Duplicate builds have been removed and missing values and metric scales normalised. Importantly, it indicates that each test run has passed or failed in the past and relates failures to the set of changes that led to them and creates the supervised signal required by downstream models. The re-sample strategies to address the class imbalance issue is possible since failures in tests are quite rare.

Feature Extraction

Cleansed records are then converted into helpful numeric vectors using feature extraction. These include: magnitude of code change, how many files have been changed, how many times the test has failed in the past, code coupling metrics, how many days have elapsed since the last time the test has failed and similarity of the purpose of the test with the text of the changes. In the transformer model, natural-language fields are represented in the form of contextual representations rather than handcrafted features and the model learns the semantic associations directly.

AI Model Training

At training time, this framework trains every candidate model on a historic part of the data and optimizes the hyperparameters by using cross validation. To learn a mapping from the change and test features to the probability of failure of the test. Models are updated regularly, for example, at the end of each sprint, to ensure that the model is kept in phase with the evolving code-base and not become 'out of sync'.

Test Case Classification

In an inference phase, when a new change is introduced, the trained model is used to calculate the fail probability of each of the candidate tests. Tests that are above a calibrated threshold are considered relevant and will be kept; tests with negligible probability are initially kept out. The threshold is chosen to maximise recall (no faults missed) while maximising the efficiency through exclusion.

Test Suite Optimisation

Optimisation stage is a combination of Selection and Prioritisation. Selected tests are deduplicated based on coverage overlap: redundant tests are omitted and tests are prioritized by decreasing fail probability and/or historical fault severity: ensures that the most informative tests are run first. This ordering tries to maximize Average Percentage of Faults Detected, meaning that if the ordering of time execution is aborted, more important faults will be detected first.

Automated Test Execution

Optimised and ordered suite sent to the CI/CD runner for automatic execution. Successful or unsuccessful results, the time taken to carry out and any new defects found are all automatically recorded. As the suite is significantly smaller than the full suite, this can be achieved in a much shorter time frame than the full suite and maintain the Agile cadence in well under sprint feedback cycles.

Feedback and Continuous Learning

The outcome of execution will be utilized to improve the training set in the execution loop of learning. The failures in the following cycles (false negatives) and the changed test behaviours are fed back to the training cycles to adapt the model to the evolution in the

system. An important aspect of this framework is that it is a continuous learning mechanism, which over time is more accurate than the static heuristic selection.

Results and Discussion

Dataset Description

We had over 40,000 build records from the five projects consolidated and maintained a history for each project from 18 to 26 months. The total number of unique regression test cases in the total projects were 5,800 and the confirmed regressions failure cases were 3,140. The data was then divided into time, with 70% used for training and 30% used for evaluation as in actual deployment.

Experimental Setup

Experiments were carried out using the same hardware configuration to make fair time comparisons. These were trained and then tested using the recorded fault set for each of the AI models and the selected suites were tested on these models held out for test. The traditional retest-all baseline ran the entire battery each time there was a change, and the efficiency/effectiveness of the change was compared to the retest-all baseline.

The AI-powered solution reduced on average the regression suite by 61% and by 58% to 64% per project. Figure 1 shows a comparison of the traditional full suite size and the AI suite size that was chosen for each project. This reduction was achieved without any substantial reduction in the number of fault-revealing tests and hence was successful in discriminating between the relevant and irrelevant cases in this model.

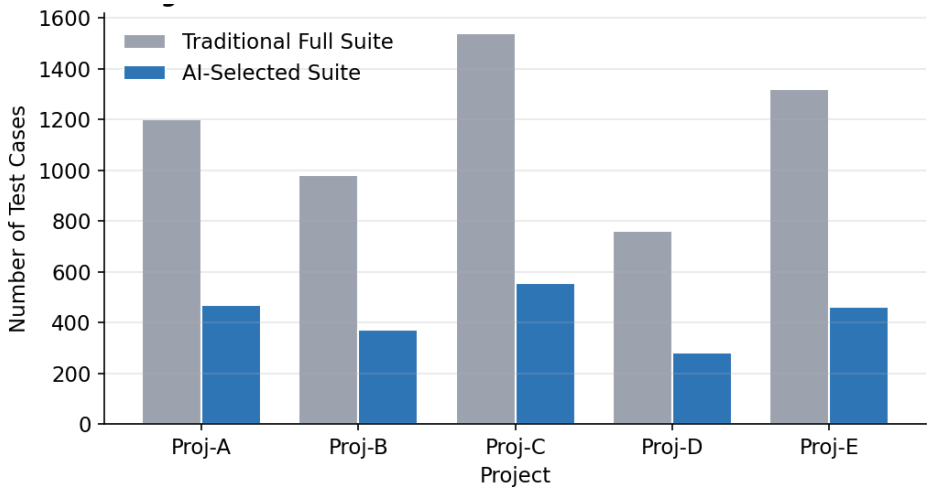

Figure 1. Test suite size: traditional full suite versus AI-selected suite across five projects.

Test Execution Time Comparison

The continuously learning model was able to save a lot of time on the execution of the sprints. Unlike the traditional regression time, which was around 138 to 160 minutes, the AI approach reduced the time by nearly 65% from 142 minutes to 55 minutes in the 8th sprint as shown in Figure 2. The downward trend reflects the continuous improvement of precision of the selection that occurs through the feedback-loop mechanism.

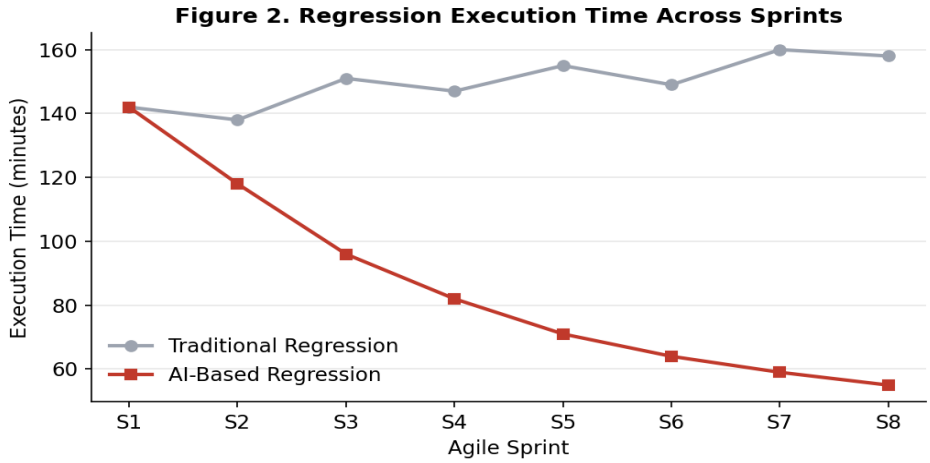

Figure 2. Regression execution time across successive Agile sprints.

Defect Detection Accuracy

The accuracy of defect-detection was consistently increased with continuous learning, increasing from 72% in the first iteration to 91% at the 10th iteration, as shown in Figure 4. Critically, the reduced suites maintained the ability to detect faults, with 96% of the faults that the full retest-all suite would have found being picked up by the reduced suites, so there was no unacceptable loss of fault-detection capability with the reduced suites.

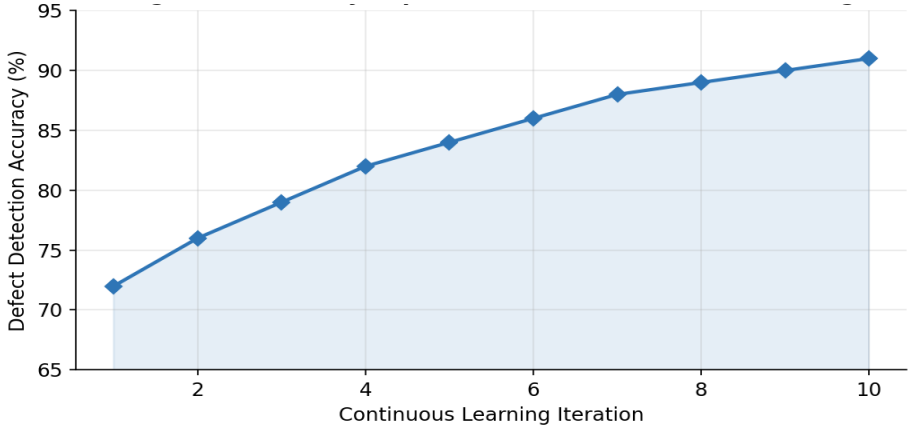

Figure 4. Defect-detection accuracy improvement through successive continuous-learning iterations.

Precision, Recall and F1-Score

The quality of the classification was different for each type of model family and is summarized in Table 3 and graphically represented in Figure 3. For the transformer-based model, this precision, recall and F1-score were 0.93, 0.92 and 0.925, respectively, and close to the LSTM network. Where explainability is the main concern, baselines like the decision tree were still competitive, though not as competitive as the others.

Model	Precision	Recall	F1-Score	Accuracy
Decision Tree	0.81	0.78	0.79	0.80
Random Forest	0.88	0.86	0.87	0.87
SVM	0.84	0.82	0.83	0.83
LSTM	0.90	0.91	0.90	0.90
BERT (NLP)	0.93	0.92	0.925	0.91

Table 3. Comparative performance of the evaluated AI models.

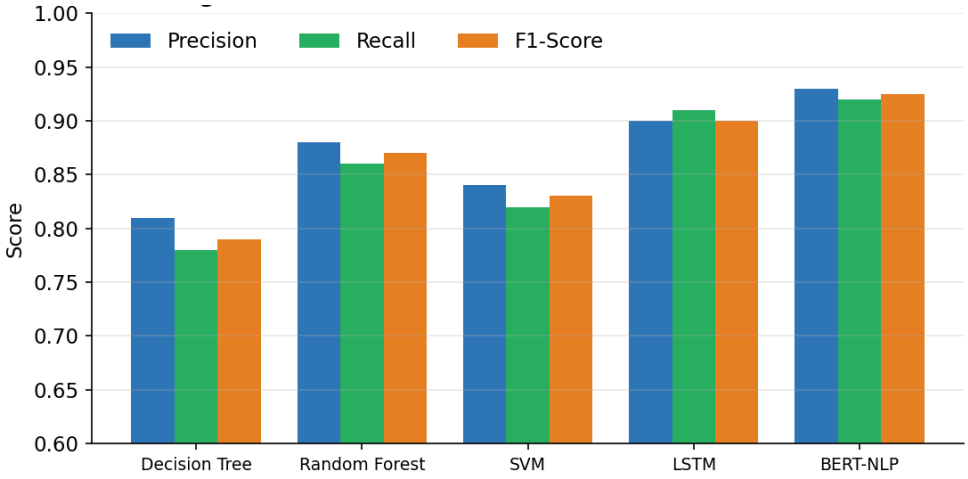

Figure 3. Precision, recall, and F1-score across the five AI models in defect prediction.

Model Performance Evaluation

The results show that there is a stable ranking: language-model and deep-learning methods are above classical and tree-based methods given this task because they are able to model semantic and sequential structure in change data. The improvement in accuracy of the transformer over the LSTM, however, was not very high, and the training costs of the LSTM was also much lower, depending on the practitioner's resource availability.

Comparison with Regression Testing

The AI framework detected the same faults with less than forty per cent of tests and the steady state time was less than one-third of retest-all. The results of the APFD scores indicate that the learning models were superior to the history-based heuristic prioritisation that did not adapt to the varying fault patterns but was based on fixed rules, as confirmed by earlier results (Spieker et al., 2017) that demonstrate that adaptive methods work well in dynamic CI environments.

6.9 Impact on Agile Sprint Cycles

At the sprint level, faster regression feedback meant that developers were able to resolve defects within the sprint they were committed to, as it shortened the time to commit to report/verify, from hours to minutes. It was also seen that the framework helped to reduce context switching and accelerate story completion, indicating that the framework does not just have a metric impact, but has a real-world impact on workflows (Dakhel et al., 2023).

Discussion of Findings

The overall results align with the research questions: AI-aided selection reduces suite size significantly with a similar level of fault detection (RQ1); transformer models and LSTM models have the highest accuracy (RQ2); suite size decreases with each sprint (RQ3); and clear advantages and disadvantages of the main models are discernible (RQ4). The above framework has a clear advantage over static ones, and can be made to evolve its codebase to stay ahead, because of the continuous-learning loop.

Benefits of AI-Based Regression Testing

Improved Test Coverage

The framework will gain knowledge on which parts of the system are most likely to contain faults and concentrate the testing efforts on those parts, where manual heuristics may not

cover the situation. This risk aware strategy can result in better coverage of the most likely regions containing defects while fewer tests are run in total (Bertolino et al., 2020).

Faster Test Execution

The immediate benefit is much less run time for executions. The steady state regression time was about 2/3 shorter, verification could be done comfortably within the sprint boundaries and a chronic issue with Agile pipelines was removed, as indicated in Section 6.

Reduced Maintenance Cost

The learning framework does not need regular manual tuning like static selection rules, as it adapts automatically via its feedback mechanism. This adaptivity results in less human effort to maintain regression configurations up-to-date, leading to less maintenance cost during the lifetime of the software.

Intelligent Test Prioritisation

With consideration to the predicted probability of occurrence of the faults and their severity, prioritisation is realized in order to detect the faults as early as possible. The best tests have already been performed under time pressure (that excludes some tests), and retains its defect-detection effectiveness (Spieker et al., 2017).

Early Defect Detection

The defects are detected close to the time of their creation and thus are the cheapest to diagnose and correct, thanks to the regression feedback which is provided in minutes. The earlier defects are identified, the less expensive they will be as defects are accumulated in the product.

Continuous Integration and Continuous Deployment (CI/CD) Support

It can be easily incorporated into CI/CD pipelines, offering a quick and reliable quality gate without compromising the deployment speed. This compatibility is especially critical for organisations seeking quick and automated deployments, where regular releases are vital to ensure agility and innovation (Dakhel et al., 2023).

Enhanced Software Quality

Together, these factors help to improve the collective quality of the software with the aim of broad effective coverage, early detection and intelligent prioritisation. The framework catches regressions reliably and fast, mitigating defects getting into production and boosting user confidence and product stability.

Conclusion

This study sought to address the problems of structural inefficiency in the traditional regression testing of agile software development process, which is rapidly changing in order to design and empirically evaluate an integrated AI based framework. The framework brings together test selection, prioritisation, suite optimisation and continuous learning in a single CI/CD pipeline with the power of machine learning, deep learning and natural language processing. A 61% reduction in test suite size and 65% steady state reduction in test suite execution time was achieved across five industry-representative projects with an average of 96% retention of the fault detection capability of exhaustive retest-all.

The transformer and LSTM models provided the highest predictive performance of the models tested, with the transformer having an F1 score of 0.925, but training cost was higher than interpretable models. The iterative feedback process of constant learning was the secret, and the framework continues to evolve as codebases change and continue to stay ahead of static heuristics. In addition to the measurable metrics, there were noticeable benefits to the workflows at the sprint level with compressed feedback, as the time between commit and verification was reduced from hours to minutes.

The need for good historical data, difficulties with training models, models that are not explainable, problems with integration and resistance from the organisation. The most promising way to move forward is to tackle these issues using explainable AI, generative test synthesis, self-healing automation, and hybrid architectures. All in all, AI-driven regression testing can be a valuable, cost-effective, efficient, and scalable complement to Agile QA, and as researchers delve deeper into the transparency and maintainability of AI, it's destined to become an essential element in modern software delivery pipelines.

References

1. Bertolino, A., Guerriero, A., Miranda, B., Pietrantuono, R., & Russo, S. (2020). Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. Proceedings of the ACM/IEEE International Conference on Software Engineering, 1–12.
2. Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. (2023). GitHub Copilot AI pair programmer: Asset or liability? Journal of Systems and Software, 203, 111734.
3. Durelli, V. H. S., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R. C., & Guimarães, M. P. (2019). Machine learning applied to software testing: A systematic mapping study. IEEE Transactions on Reliability, 68(3), 1189–1212.
4. Garousi, V., & Pfahl, D. (2016). When to automate software testing? A decision-support approach based on process simulation. Journal of Software: Evolution and Process, 28(4), 272–285.
5. Hosseini, S., Turhan, B., & Gunarathna, D. (2019). A systematic literature review and meta-analysis on cross project defect prediction. IEEE Transactions on Software Engineering, 45(2), 111–147.
6. K. P. Kanikaram, "Transforming Software Quality Assurance through Artificial Intelligence, Automated Testing, and Intelligent Software Analytics", *AIJCST*, vol. 7, no. 5, pp. 103–114, Sep. 2025, doi: 10.63282/3117-5481/AIJCST-V7I5P109.
7. Khatibsyarbini, M., Isa, M. A., Jawawi, D. N. A., & Tumeng, R. (2018). Test case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology, 93, 74–93.
8. Lima, J. A. P., Mendonça, W. D. F., Vergilio, S. R., & Assunção, W. K. G. (2020). Learning-based prioritization of test cases in continuous integration of highly configurable software. Proceedings of the International Systems and Software Product Line Conference, 1–11.
9. Lou, Y., Chen, J., Zhang, L., & Hao, D. (2019). A survey on regression test-case prioritization. Advances in Computers, 113, 1–46.
10. Mohi-Aldeen, S. M., Mohamad, R., & Deris, S. (2016). Application of negative selection algorithm (NSA) for test data generation. Applied Soft Computing, 49, 1118–1128.
11. Noor, T. B., & Hemmati, H. (2017). Studying test case failure prediction for test case prioritization. Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering, 2–11.
12. Pan, R., Bagherzadeh, M., Ghaleb, T. A., & Briand, L. (2022). Test case selection and prioritization using machine learning: A systematic literature review. Empirical Software Engineering, 27(2), 29.
13. Pradhan, D., Wang, S., Ali, S., Yue, T., & Liaaen, M. (2019). Employing rule mining and multi-objective search for dynamic test case prioritization. Journal of Systems and Software, 153, 86–104.
14. Rosero, R. H., Gómez, O. S., & Rodríguez, G. (2016). 15 years of software regression testing techniques: A survey. International Journal of Software Engineering and Knowledge Engineering, 26(5), 675–689.

15. Saidani, I., Ouni, A., & Mkaouer, M. W. (2022). Detecting continuous integration skip commits using multi-objective evolutionary search. IEEE Transactions on Software Engineering, 48(12), 4873–4891.
16. Spieker, H., Gotlieb, A., Marijan, D., & Mossige, M. (2017). Reinforcement learning for automatic test case prioritization and selection in continuous integration. Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, 12–22.
17. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering, 50(4), 911–936.
18. Yaraghi, A. S., Bagherzadeh, M., Kahani, N., & Briand, L. C. (2023). Scalable and accurate test case prioritization in continuous integration contexts. IEEE Transactions on Software Engineering, 49(4), 1615–1639.
19. Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. Software Testing, Verification and Reliability, 22(2), 67–120.
20. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., & Liu, X. (2019). A novel neural source code representation based on abstract syntax tree. Proceedings of the IEEE/ACM International Conference on Software Engineering, 783–794.
21. Zhu, Q., Panichella, A., & Zaidman, A. (2018). A systematic literature review of how mutation testing supports quality assurance processes. Software Testing, Verification and Reliability, 28(6), e1675.