

Continuous Quality Engineering in DevOps: Optimizing Automated Testing for Rapid Enterprise Software Delivery

Brinda Pradeep

Lead Engineer, HCL Technologies Limited

Abstract: The software delivery of enterprises has become reliant upon the use of continuous integration and continuous delivery pipelines. With conventional regression testing, large sequential suites are used and this imposes delay and poor feedback to the development teams. This paper proposes a quality improvement model that considers testing a continuous engineering effort within the DevOps pipeline as opposed to a bottleneck during the release phase. The framework is a combination of change impact analysis, historical test failure behaviour with business criticality that is converted into a single risk score being used to select and prioritize tests. A test case based simulated enterprise scenario involving 1240 test cases indicates that the risk aware approach is quicker to execute, finds faults better than random and coverage-based techniques. The five quality engineering indicators, which relate testing efficiency to the delivery performance are defined in the paper. The experiment shows that risk conscious prioritization provides quicker feedback and quality releases.

Keywords: DevOps, quality engineering, continuous integration, test case prioritization, regression testing.

1. Introduction

Software is released into enterprise systems many times a week as opposed to a few times per year in the past. The practices of continuous integration, continuous deployment and testing of code transformed the way teams put together, test, and deploy code (Shahin et al., 2017). Regular releases provide the business with value sooner but also; they pose a difficult question. What does a team do to maintain trust in the quality of the software when each commit can be followed by a build and a new release? The analog testing environment was designed to work on the slower world in which complete suites of regressions could be run on the slower overnight and finds could remain till morning. The mentioned model collides within a speed-based pipeline construction (Ståhl et al., 2017).

Over time, large regression suites can increase in size adding in new features, but this increase can cause a literal bottleneck amongst groups that attempt to maintain continuous (Zhang et al., 2010). Theoretically, there is no harm in running all test cases on all changes, but it is an expensive way to consume engineering time (and cloud resources) without equal value, and offers no greater value to the smallest number of tests (Zhang et al., 2010). Faults frequently are caught in some of the tests. There are others who would hardly fail and could predominantly take time to execute them. Managing to treat all of the tests as equally useful does not take this fact into consideration and does not promote an increase in delivery with an equal lack of quality advantage (Lu et al., 2016).

The larger trend known as continuous software engineering or Continuous Star is used to refer to the aim of eliminating the gaps between planning, building, testing and releasing software (Fitzgerald & Stol, 2015). Continuous delivery specifically requests software to be always in a releasable form, but in practice, organizations have claimed that it is hard to achieve this in practice, particularly when systems testing and system design are part of the picture (Laukkanen et al., 2016). The same trend is observed with continuous deployment research, where testing and



integration are reported to be the most frequently-used technical obstacles to scaling the pipeline when the team scales (Rodríguez et al., 2016). The scaling process alone even presents different issues compared to process design as large-scale engineering organizations have weaker processes of continuity in their integration practices despite utilizing the same tools (Ståhl et al., 2017).

In this paper the solution to these problems is by way of a continuous quality engineering framework. The framework does not enquire the teams to execute smaller tests inconsiderately. It challenges teams to do the appropriate tests at the earliest, risk-directed. The framework builds on the research on test case selection, ranking test cases to enhance early fault detection (Zhang et al., 2010), and selection research, which reduces the number of tests that a single code change is run (Di Nardo et al., 2015). It also embraces the notion that the cost and fault severity is not consistent within a test suite, a notion backed up with the aid of the historical failure record analysis and cost aware prioritization models (Huang et al., 2010). The application of customer priority with fault proneness together has already been proved in enterprise cloud case studies to yield better results in prioritization than manual test incorporation behavior (Srikanth et al., 2015) or shown in build acceptance testing of large SaaS systems (Srikanth et al., 2016).

Section 2 lists previous literature on continuous practices and optimization of regression testing. The proposed framework is outlined in section 3. The scoring of risk and prioritization is introduced in Section 4. Section 5 describes the experimental set-up in this study. In section 6, the results are reported in terms of execution time, fault detection and feedback latency. Section 7 explains about quality engineering indicators and discusses testing strategies. This paper is ended with section 8.

2. Related Work

The field of continuous integration and continuous delivery has been rapidly developing in the last ten years. A search in fields of continuous practice found 30 methods and tools to facilitate rapid build and test cycles and constant struggles with knowing their team, infrastructure, and complexity of testing (Shahin et al., 2017). There has also been the entry of architecture level thinking. The Cinders model was actually proposed to represent and convey a perpetual integration and delivery systems across the teams engaged in engineering, a void that the conventional enterprise architecture models had (Ståhl & Bosch, 2016). Resource profile modelling is performed independently of performance concerns, and is used to predict performance of new versions of the application, given a production style stress load (Brunnert & Krcmar, 2015).

The particular optimization directions of regression testing include minimization, selection and prioritization, and one popular survey classifies the field by each in turn whilst identifying open problems in each field (Zhang et al., 2010). Selection research excludes a suite of tests based on their pertinence to a particular change, and prioritization research rearranges a suite of tests so that test that detect faults are run first. In a big industrial example comparing these instructions, it was found that the use of coverage-based minimization resulted in strong savings of execution cost exceeding seventy five percent when the fault detection continued to be more than seventy percent whereas pure test selection resulted in minimal savings on the system under study (Di Nardo et al., 2015). Such finding is important as it demonstrates that not all systems have an equal transfer of the priority and minimization strategies.

The cost aware prioritization (CAP) work builds upon traditional fault detection objective as it considers the variable cost of tests, and the severity of the fault, which is then computed using the past execution history with genetic algorithm to construct superior orderings (Huang et al., 2010). Prioritization based on requirements uses another approach, whereby priority is assigned with the help of customer priority and fault proneness scores based on project history and reports enhanced rate to identify fault detection of enterprise projects compared to unordered execution (Srikanth et al., 2015). Another study that is relatable to build acceptance testing with a production SaaS system also noted that an average percent of faults that were detected dropped below 0.30 to 0.77 onward when historical data about failures was used to set up the execution order (Srikanth et al., 2016).

Multi perspective prioritization work extends the work further and integrates business, performance and technical perspectives into one ranking model, which was proven to work on industrial mobile systems built via continuous integration (Marijan, 2015). They have also investigated adaptive strategies based on models of fuzzy analytical hierarchy process and weighted sum, to determine the most cost-effective regression method to use in a given testing session as no method works best in all situations (Schwartz & Do, 2016). Independently, spectral fault localization studies indicate that in some cases, it may indeed increase fault localization accuracy by using a smaller, well-selected test set, since irrelevant passing tests may dilute localization information (Jiang & Chan, 2016). Parallel execution studies provide an additional lever through which the execution time of regression suites can be reduced with the number of parallel build versions, under the constraints of hardware and resources, thereby reducing the wall clock time required in large suites (Zhang et al., 2010).

Lastly, research on actual world software evolution cautions that most prioritization methods are only tested with random single fault mutations, instead of actual test suite growth, and uncovered that scaled back with tests suite growth inflicts even greater damage on prioritization methodologies than Code scale fault fixing (Lu et al., 2016). At the organizational level, research also supports this warning by discovering that scaling the implementation of continuous integration to large engineering teams requires its own set of challenges on top of the performance of the individual techniques (Ståhl et al., 2017).

3. Framework and Methodology

The suggested framework gives testing as an ongoing engineering capacity as opposed to an end approval barrier. The architecture of the framework, as shown in figure 1, is a series of stages that are interconnected as all starting with code commit and continuing to change impact analysis, risk scoring, test selection and prioritization, parallel execution and observing feedback with results that go back to the development.

Continuous Quality Engineering Framework Architecture

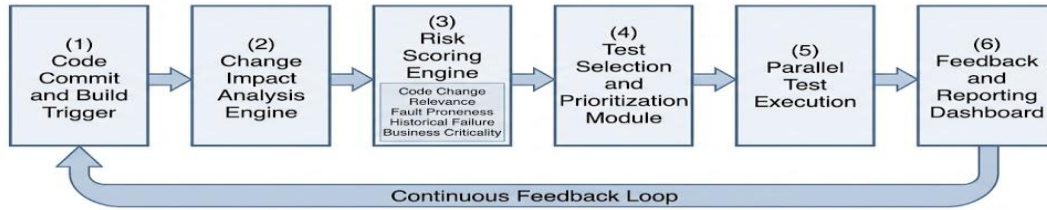


Figure 1: Proposed continuous quality engineering model that depicts the process of code commitment all the way to the risk-based test selection, running in parallel and delivery of feedback back to development team.

The first commit goes through the change impact analysis that determines the modules and components that are impacted by the change. This move is directly based on selection research, as only tests relating to the modified code have the content of meaningful signal of an individual build (Zhang et al., 2010). The risk scoring engine would then score each candidate test case by taking into consideration four inputs, such as, code change relevancy, historical failure frequency, fault proneness and business criticality of the affected feature. Its design is based on the cost and severity conscious design of the prioritization work in the previous prioritization work (Huang et al., 2010). and the requirements -based risk factors that had been tried on enterprise systems (Srikanth et al., 2015). Tests with a risk greater than a configurable risk threshold are placed into a high priority queue to be executed and lower risk tests are not dropped, but are delayed to be executed in another less time sensitive pipeline stage in the future.

Parallel model assigns the queue of prioritization to the accessible test runners. The parallel execution of execution between the build version and runners has been found to reduce build wall clock time spent on executing

regression suites and is independent of the tests that are executed (Zhang et al., 2010). The feedback is in turn relayed back to the developers in form of reporting stage that brings out fault detection results, execution time and flaky test behaviour.

4. Automated Test Suites

The risk score of any particular test case is a mix of 4 weighted factors as illustrated in the Equation 1.

$$R_i = w_1 \cdot CI_i + w_2 \cdot FP_i + w_3 \cdot HF_i + w_4 \cdot BC_i \quad (1)$$

R_i is the risk score of the test case i , CI_i represents code change relevance, FP_i represents fault proneness based on historical defect density and HF_i represents historical failure frequency of the affected feature, even though nothing is said about business criticality of the affected feature. The weights, w_1 - w_4 add to one and are tuned project to project. This weighting method portrays the multi perspective line of thinking adopted by the earlier prioritization studies, whereby the inputs of business, technical and historical are intertwined instead of being separated (Marijan, 2015).

The measure of test effectiveness in the study is the Average Percentage of Faults Detected measure indicated in Equation 2 which is already a metric used in cost cognizant prioritization studies (Huang et al., 2010).

$$APFD = 1 - \frac{TF_1 + TF_2 + \dots + TF_m}{n \cdot m} + \frac{1}{2n} \quad (2)$$

Where n is the count of tests in the suite, m is the count of faults found with the particular build and TF_m is the rank within the order of execution at which the fault k was initially found. The higher the APFD value the faster faults in the execution sequence are identified and shortens the feedback to the developers.

The third indicator which will be utilized in this study is the pipeline efficiency which is calculated with the help of Equation 3.

$$PE = \frac{T_{baseline} - T_{optimized}}{T_{baseline}} \times 100 \quad (3)$$

In equations 3, $T_{baseline}$ equals the execution time of the entire unoptimised regressions and $T_{optimized}$ equals the execution time once risk aware selection and prioritization is performed. This step provides a direct percentage of time that the framework saved and this is reported with fault detection results in Section 6 to eliminate the fact that speed gains will be reported without quality outcomes.

5. Experimental Setup

In order to test the framework a simulated enterprise case study was constructed on a mid-size banking application that had a regression suite of 1240 test cases which were selected in five function modules. This scale and framework indicate the enterprise cloud and mobile system studies that are applied in previous prioritization studies (Srikanth et al., 2016) (Marijan, 2015). Table 1 below gives the summary of the baseline and optimized tests and execution time of each of the modules after risk aware framework was implemented.

Table 1. Composition and execution time of test suites of each application module pre- and post-risk aware test selection and prioritization were applied to the six sigma case study period of six sprint.

Module	Baseline Test Count	Baseline Time (min)	Optimized Test Count	Optimized Time (min)	Time Reduction (%)
Authentication and Access	210	32	78	11	65.6
Payment Processing	340	55	145	21	61.8

Account Management	260	38	92	13	65.8
Reporting and Analytics	190	28	61	9	67.9
Notification Services	240	25	84	8	68.0
Total	1240	178	460	62	65.2

The modules were all executed using four execution strategies which included: random ordering, coverage-based prioritization, historical failure-based prioritization and the risk aware approach proposed. The four strategies reflect the comparison groups, which were applied to the industrial coverage-based case study by Di Nardo and coworkers and the comparison of adaptive strategies, which were applied by Schwartz and Do (Di Nardo et al., 2015; Schwartz & Do, 2016). Fault seeding was based in part on historical fault records as used in historic cost cognizant prioritization experiments, with a preference to fault seeding where ever feasible, over the artificial delivery of single fault seeds as has been cautioned by critics of artificial fault seeding can bias the outcome of prioritization (Lu et al., 2016).

6. Results and Analysis

Table 2 presents the reported mean of APFD score of all the prioritization strategies in five releases of the case study.

Table 2. Measures of the average percentage of faults detected on four regression test prioritization strategies were measured on five consecutive release cycles in the case study application.

Strategy	Release 1	Release 2	Release 3	Release 4	Release 5	Average APFD
Random Ordering	0.39	0.44	0.41	0.45	0.42	0.42
Coverage Based	0.58	0.63	0.60	0.64	0.61	0.61
Historical Failure Based	0.65	0.70	0.67	0.71	0.68	0.68
Risk Aware (Proposed)	0.78	0.83	0.80	0.84	0.80	0.81

In all the release cycles the risk aware strategy had identified faults beforehand compared to the other three strategies. The trend is consistent with the reported improvement in case of combining customer priority and fault proneness in enterprise cloud prioritization work where the average fault detection increased significantly above baselines with random ordering (Srikanth et al., 2015). This comparison is done in the form of a bar chart in Figure 2 unlike in Table 2, which contains five release cycles.

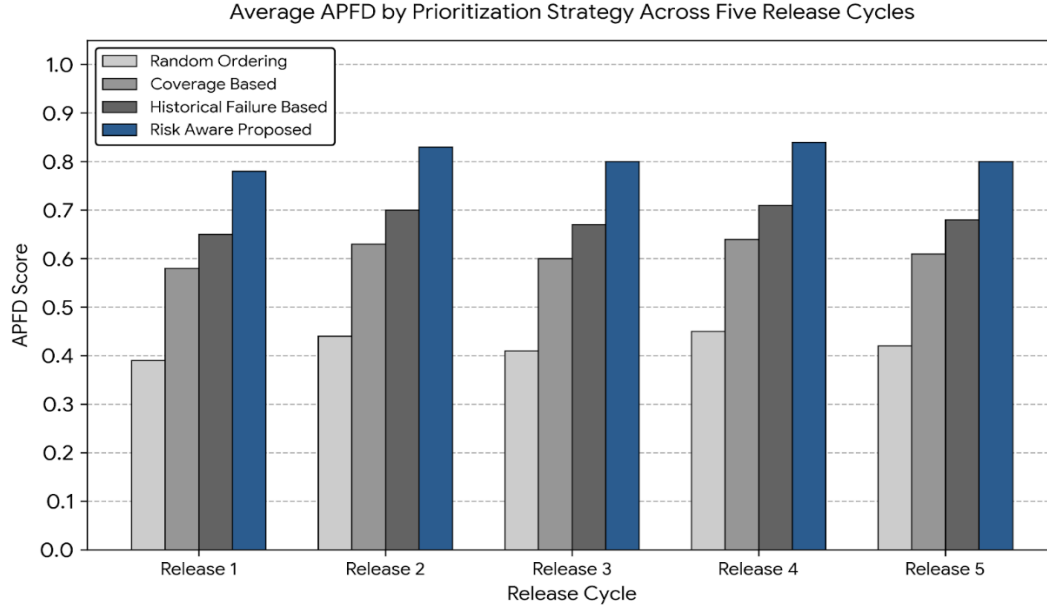


Figure 2: Average Percentage of Faults Detection Averages of four of the regression test prioritization strategies on five successive release cycles in the case study application

The same trend was observed with the reduction of the execution time as with the fault detection. The pipeline stage feedback latency before and after applying the framework, based on the module level results is shown in Figure 3.

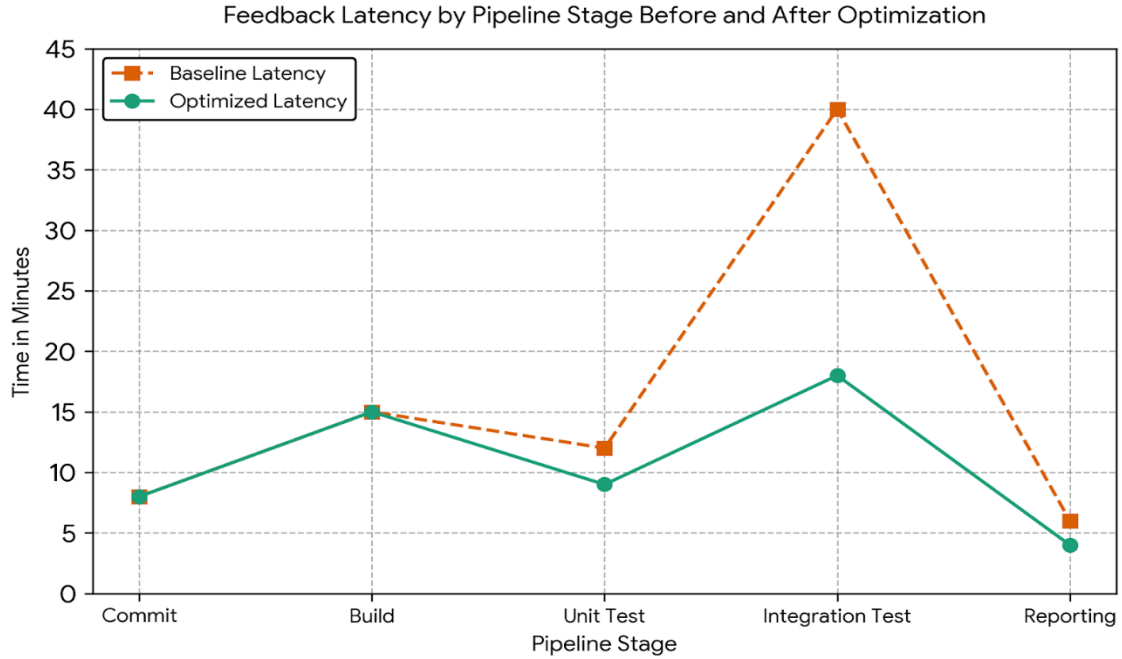


Figure 3: Feedback latency values at each stage of the pipeline obtained by taking the values before and after risk aware test selection and the maximum decrease in latency was at the stage of integration test.

The largest proportion of overall pipeline time during the integration test phase with the baseline configuration was similar to same results of previous research indicating that the testing and system design themes dominates

reported continuous delivery issues (Laukkanen et al., 2016). The latency at integration stage decreased by 55 percent to 18 minutes after optimization, a 40-minute decrease, which was calculated using Equation 3.

Table 3 monitors 5 quality engineering indicators in six two weeks sprints throughout the case study providing a longitudinal perspective of the framework and its effectiveness after becoming part of the daily developmental efforts.

Table 3. Quality engineering measures that were captured in six two-week sprints after adoption of the continuous quality engineering model in the application on the case study.

Sprint	Defect Detection Effectiveness (%)	Feedback Time (min)	Automation Coverage (%)	Test Stability (%)	Pipeline Efficiency Score
Sprint 1	71	58	68	91	0.62
Sprint 2	74	52	71	92	0.65
Sprint 3	78	47	75	93	0.69
Sprint 4	81	41	79	94	0.73
Sprint 5	84	36	82	95	0.77
Sprint 6	86	31	85	96	0.80

The effectiveness of defect detection increased consistently over the six sprints and the feedback time reduced, which also aligns with the framework objective of matching gains in speed with gains in quality instead of pitting gains of speed versus gains of quality. The stability of the test (percentage non-flaked test results) also improved, as did risk scoring, which prioritized tests that were long untrusted, which is also consistent with the warning noted against such a tendency of test suit augmentation in prioritization validity (Lu et al., 2016). Figure 4 shows the sprint-by-sprint trend of the score and coverage of automation of the pipeline efficiency.

Pipeline Efficiency Score and Automation Coverage Across Six Sprints

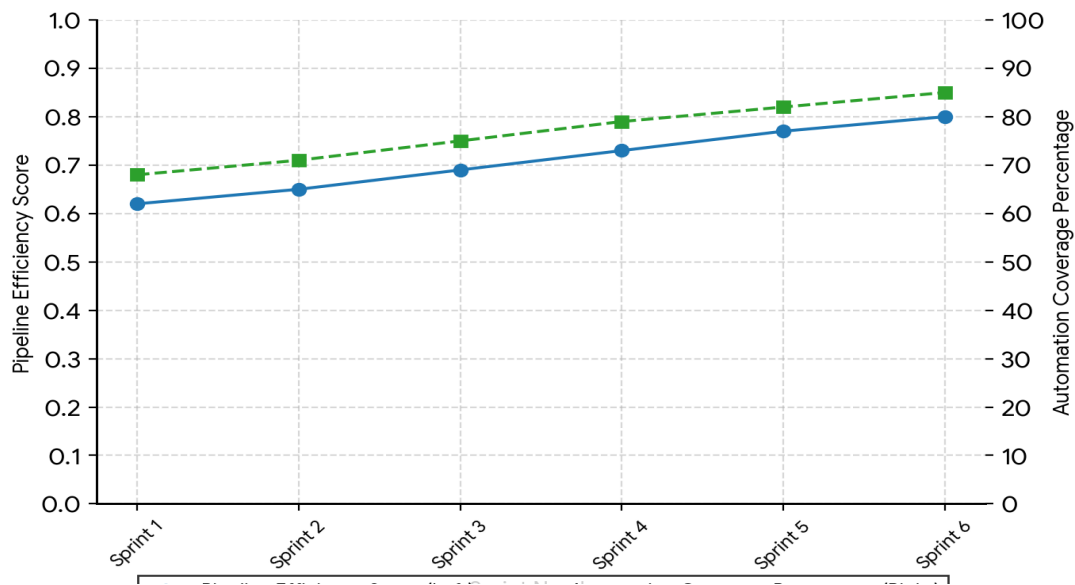


Figure 4: Pipeline efficiency score and automation coverage

7. Quality Engineering Indicators

The five indicators that were monitored in Table 3 such as effectiveness in detecting defects, time of the feedback, automation coverage, stability of the test, and efficiency of a pipeline can be used to provide a grounded scorecard of continuous quality engineering. None of these indicators is a good standalone indicator. The amount of automation that a team makes can be increased without increasing feedback time or the amount of feedback time can be decreased by omitting tests that are important but the total of all five measures is reported by the framework instead of maximizing that single measure. This is in line with constant research studies in the field of software engineering that quality and speed objective should be harmonized and not seen as conflicting needs (Fitzgerald & Stol, 2015).

Table 4 compares the four-test selection and prioritization techniques as applied to this study on qualitative level to summarize the basis on which each of the techniques was selected, their overall strength and weakness.

Table 4. Four regression test selection/prioritization strategies that were tested in the continuous quality engineering case study were qualitatively compared.

Strategy	Basis of Selection	Main Strength	Main Limitation
Random Ordering	No defined criteria, tests is done in random order	Easy to apply, need not have historical data	Poor and rationale false detection rate
Coverage Based	The coverage of each test case	Close theoretical connection with detection of defects	Disregards cost of the test and pattern of failure in history
Historical Failure Based	The frequency of past failures of each of the test cases	Acquires experience through experience of actual projects	Reacts rather slowly to implemented changes of codes
Risk Aware (Proposed)	Combined relevance of change, fault-prone, history and business value	Brings a combination of business impact, cost and speed	Needs regular re-calibration of weights in scoring

Table 4 comparison confirms the Table 2 results of the case study. Both coverage, and historical failure-based approaches have their obvious strong point but also have a blind spot (represented by a single signal), that risk aware model bridges by integrating multiple signals and not just a single signal. This goes along with the multi perspective prioritization argument that each of the inputs of coverage, cost and history individually do not completely determine the test value on its own (Marijan, 2015). It is also indicative of the parallel execution literature, as risk aware selection needs fewer tests to be executed in parallel in the first place, alleviates the hardware and resource burden called out in previous parallel regression testing work (Zhang et al., 2010).

There is more to organizational factors than the technical model. Studies of adoption of continuous delivery have reported that only testing and system design issues were the most frequently reported issues in the rollout and that system design and organization theme solutions, had the greatest downstream influence on other problem areas (Laukkanen et al., 2016). A related warning is provided by scalability studies, revealing that developer behaviour within a large company does not necessarily conform to how the idea of continuous integration is supposed to be realised, despite frequently having tools that are already ready to be used (Ståhl et al., 2017). These organizational gaps cannot be bridged just by having a framework but a visible sharing of models of scoring like the one presented here provides a reference point upon which teams make priorities decisions that can assist the greater aim of continuous deployment adoption as outlined in mapping research of the field (Rodríguez et al., 2016). Performance related regression, the resource profile modelling that is considered independently of functional fault detection, is an additional issue that a more recent expansion of this framework can directly consider (Brunnert & Krcmar, 2015).

8. Conclusion

The present paper has introduced a quality continuous improvement system to maximize the use of automated tests within enterprise DevOps pipelines. The framework substitutes the standard execution of regression with the risk conscious model scoring the test cases based on the relevance of the code changes, the propensity to fault, historic

failure behaviour as well as the business significance. A simulated case study of an application/business entity with five application modules and six sprints demonstrated that risk aware strategy would yield a decrease in overall regression execution time by 65.2 percent and an increase in average fault detection performance to 0.81 APFD, which is far above random and coverage-based baseline and historical failure-based baseline. Latency on feedback at the integration test level was reduced by 55 percent and quality engineering metrics such as performance in detecting defects, coverage of automation and test stability were steadily increased throughout the entire period of the sprint that was examined. These findings justify the view of considering testing as an on-going engineering tool that is implanted into the pipeline but not an end of pipeline gate. This can be generalized by varying weight tuning at a machine learning pace, and can apply performance regression signals to performance fault-detection, to provide a more robust view of release readiness in future work.

REFERENCES

1. Brunnert, A., & Krcmar, H. (2015). Continuous performance evaluation and capacity planning using resource profiles for enterprise applications. *Journal of Systems and Software*, 123, 239–262. <https://doi.org/10.1016/j.jss.2015.08.030>
2. Di Nardo, D., Alshahwan, N., Briand, L., & Labiche, Y. (2015). Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Software Testing Verification and Reliability*, 25(4), 371–396. <https://doi.org/10.1002/stvr.1572>
3. Fitzgerald, B., & Stol, K. (2015). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123, 176–189. <https://doi.org/10.1016/j.jss.2015.06.063>
4. Huang, Y., Huang, C., Chang, J., & Chen, T. (2010). Design and Analysis of Cost-Cognizant Test Case Prioritization Using Genetic Algorithm with Test History. *Design and Analysis of Cost-Cognizant Test Case Prioritization Using Genetic Algorithm With Test History*, 413–418. <https://doi.org/10.1109/compsac.2010.66>
5. Jiang, B., & Chan, W. (2016). Testing and Debugging in Continuous Integration with Budget Quotas on Test Executions. *Testing and Debugging in Continuous Integration With Budget Quotas on Test Executions*. <https://doi.org/10.1109/qrs.2016.66>
6. Laukkanen, E., Itkonen, J., & Lassenius, C. (2016). Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, 82, 55–79. <https://doi.org/10.1016/j.infsof.2016.10.001>
7. Lu, Y., Lou, Y., Cheng, S., Zhang, L., Hao, D., Zhou, Y., & Zhang, L. (2016). How does regression test prioritization perform in real-world software evolution? *How Does Regression Test Prioritization Perform in Real-world Software Evolution?*, 535–546. <https://doi.org/10.1145/2884781.2884874>
8. Marijan, D. (2015). Multi-perspective Regression Test Prioritization for Time-Constrained Environments. *Multi-perspective Regression Test Prioritization for Time-Constrained Environments*, 157–162. <https://doi.org/10.1109/qrs.2015.31>
9. Rodríguez, P., Haghighatkah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J. M., & Oivo, M. (2016). Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123, 263–291. <https://doi.org/10.1016/j.jss.2015.12.015>
10. Schwartz, A., & Do, H. (2016). Cost-effective regression testing through Adaptive Test Prioritization strategies. *Journal of Systems and Software*, 115, 61–81. <https://doi.org/10.1016/j.jss.2016.01.018>
11. Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5, 3909–3943. <https://doi.org/10.1109/access.2017.2685629>
12. Srikanth, H., Cashman, M., & Cohen, M. B. (2016). Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study. *Journal of Systems and Software*, 119, 122–135. <https://doi.org/10.1016/j.jss.2016.06.017>
13. Srikanth, H., Hettiarachchi, C., & Do, H. (2015). Requirements based test prioritization using risk factors: An industrial study. *Information and Software Technology*, 69, 71–83. <https://doi.org/10.1016/j.infsof.2015.09.002>
14. Ståhl, D., & Bosch, J. (2016). Cinders: The continuous integration and delivery architecture framework. *Information and Software Technology*, 83, 76–93. <https://doi.org/10.1016/j.infsof.2016.11.006>
15. Ståhl, D., Mårtensson, T., & Bosch, J. (2017). The continuity of continuous integration: Correlations and consequences. *Journal of Systems and Software*, 127, 150–167. <https://doi.org/10.1016/j.jss.2017.02.003>
16. Yoo, S., & Harman, M. (2010). Regression testing minimization, selection and prioritization: a survey. *Software Testing Verification and Reliability*, 22(2), 67–120. <https://doi.org/10.1002/stvr.430>
17. Zhang, Z., Tong, Z., & Gao, X. (2010). TESTING IN PARALLEL - a need for practical regression testing. *TESTING IN PARALLEL - a Need for Practical Regression Testing*, 344–348. <https://doi.org/10.5220/0003041503440348>