International Journal of
Technology and Emerging Research
ISSN: 3068-1995 (Print) | 3068-109X (Online) | https://www.ioro.org/ijter

A Unified QA Test-Matrix Dashboard solution for Test Managers: Integrating Jira, qTest, and Grafana for Data-Driven Quality Management

Urvish Gajjar

Sr Test Manager, Health care service corporation, Dallas, TX, USA

Abstract

Quality assurance (QA) organizations increasingly rely on multiple disconnected tools — test management systems, issue trackers, and continuous integration (CI) pipelines — to plan, execute, and report on testing activity. This fragmentation makes it difficult for QA managers to obtain a timely, unified view of test coverage, defect leakage, and automation maturity. This paper presents a solution architecture that integrates qTest (test case management and execution), Jira (sprint and defect tracking), and Grafana (visualization and alerting) into a single QA test-matrix dashboard. We define a metrics framework covering seven manager-facing indicators — test coverage, missing test coverage, production bugs, in-sprint bugs, in-sprint automation, regression automation, and sprint predictability — and describe the data pipeline, panel design, and alerting rules required to compute them. We further present an illustrative deployment based on a synthetic eight-sprint dataset to demonstrate how the dashboard surfaces coverage gaps and regression debt earlier than manual reporting. The proposed approach reduces manual reporting effort, improves defect-leakage visibility, and gives QA managers an evidence-based basis for test-strategy decisions. We discuss implementation considerations, limitations, and directions for extending the framework with predictive analytics.

Keywords: *software quality assurance · test coverage · Grafana · Jira · qTest · DevOps metrics · regression testing · dashboard · test management*

1. Introduction

Modern software delivery pipelines produce testing artifacts across several disconnected systems: test case repositories such as qTest, issue trackers such as Jira, CI logs, and production incident systems. Each system captures a partial view of quality: qTest records what was planned and executed, Jira records what was found and fixed, and CI logs record what ran automatically. QA managers are commonly required to manually reconcile these sources into spreadsheets or slide decks before every sprint review or release-readiness meeting, a process that is time-consuming, error-prone, and typically several days out of date by the time it reaches stakeholders [2,9].

This reporting gap has practical consequences. Studies of DevOps and continuous-delivery practices consistently identify measurement and visibility as recurring adoption barriers: teams that lack automated, real-time quality metrics struggle to detect regressions early, misjudge release readiness, and are slower to correlate defect trends with process changes [1,3,10]. At the same time, research on regression testing and test prioritization shows that the effectiveness of automated test suites depends heavily on how coverage

International Journal of Technology and Emerging Research (IJTER)
Volume 1 | Issue 1 | 2025 | Article ID: 4141-6890-9110 | https://doi.org/10.64823/ijter.2501016
IORO Publications · Texas, USA · www.ioro.org

and execution data are tracked and acted upon over time [11,12,13] — information that is only useful to a QA manager if it is visible, current, and contextualized against sprint and release boundaries.

Grafana has emerged as a widely adopted, open-source visualization layer for operational and engineering metrics, offering pluggable data sources, alerting, and dashboard templating that make it well suited to consolidating heterogeneous QA data [6]. However, most published guidance on Grafana-based quality dashboards addresses infrastructure or performance-test monitoring rather than the specific reporting needs of QA managers — namely test coverage against requirements, defect leakage from sprint to production, automation maturity of the regression suite, and the predictability of sprint testing commitments.

This paper addresses that gap. We propose a practical, reproducible architecture for integrating qTest, Jira, and Grafana into a single QA test-matrix dashboard, and we define a compact set of seven metrics that we argue represent the minimum viable reporting surface for a QA manager. Our contribution is threefold: (1) a reference architecture for the integration and metric-computation pipeline; (2) a formalized KPI framework with explicit formulas and data sources; and (3) an illustrative dashboard design, validated on a synthetic multi-sprint dataset, that demonstrates how the metrics behave in practice and how they can surface coverage gaps earlier than manual reporting cycles.

The remainder of the paper is organized as follows. Section 2 reviews related work on DevOps metrics, test coverage, and regression testing. Section 3 motivates the problem. Section 4 presents the proposed architecture. Section 5 defines the metrics framework. Section 6 describes the implementation. Section 7 presents an illustrative evaluation. Section 8 discusses limitations, and Section 9 concludes the paper.

2. Related Work

Research on DevOps metrics has grown substantially since 2019, driven largely by the popularization of the DORA four key metrics: deployment frequency, lead time for changes, change failure rate, and time to restore service. Sallin et al. operationalized these metrics in an industrial case study and highlighted the difficulty of collecting them consistently across heterogeneous toolchains, a challenge that mirrors the integration problem addressed in this paper [6]. Amaro, Pereira, and da Silva's multivocal literature review of DevOps metrics and KPIs synthesizes both academic and grey literature and finds that quality-related indicators — defect density, test coverage, and escaped-defect rate — remain underrepresented in tool-vendor dashboards compared to delivery-speed metrics [2]. This imbalance is consistent with our observation that QA-specific reporting is often appended to generic engineering dashboards rather than designed around the QA manager's decision needs.

Several systematic reviews have examined DevOps adoption more broadly. Badshah et al. identified measurement and monitoring practices as a recurring process-improvement theme but noted that many organizations lack a unified metrics layer spanning planning, testing, and operations tools [3]. Alnafessah et al.'s survey of quality-aware DevOps research similarly found that most academic work addresses performance and reliability monitoring, while test-management-specific integration — linking planned test coverage to executed and escaped defects — receives comparatively little attention [1]. Mishra and Otaiwi's systematic mapping of DevOps and software quality reaches a related conclusion, observing that quality practices are frequently treated as a downstream consequence of delivery pipelines rather than as a first-class, independently measured concern [10].

Work specifically on continuous reliability and testing pipelines is directly relevant to our architecture. Bertolino et al. proposed DevOpRET, a framework for continuous reliability testing embedded in DevOps

pipelines, and argued that reliability and defect data must be fed back into planning in near real time to be actionable [4]. Rafi et al.'s DevOps readiness model includes measurement and monitoring capability as one of several organizational readiness dimensions, reinforcing that dashboarding maturity is not purely a tooling problem but also a process one [5]. Gomes et al. proposed a set of KPIs specifically for evaluating DevOps team performance, several of which overlap with the QA-facing metrics used in this paper, including defect escape rate and test automation coverage [9]. dos Santos Júnior et al. developed a diagnostic instrument for continuous software engineering practice adoption that includes testing and monitoring dimensions, providing a complementary organizational-assessment perspective to our tool-centric approach [8]. Santos, Alencar da Costa, and Kulesza empirically studied the effect of continuous integration practices on productivity and defect rates in open-source projects, finding that projects with more mature CI and monitoring practices exhibited measurably lower defect densities — supporting the premise that visibility into testing activity is causally linked to quality outcomes, not merely correlated with it [7].

A separate body of work addresses the test-coverage and regression-testing metrics that feed our dashboard. Pradhan et al. and Huang et al. both proposed test-case prioritization techniques driven by code-coverage and historical execution data, underscoring that coverage metrics are most valuable when tracked longitudinally rather than as single-point snapshots [11,12]. Magalhães et al. presented a hybrid regression test selection and prioritization technique validated on an industrial codebase, again relying on continuously updated coverage data [13]. Sharif, Marijan, and Liaaen applied deep learning to test-case prioritization in continuous integration, illustrating that automation-maturity metrics increasingly feed predictive, not just descriptive, tooling [14]. Chen et al. revisited the relationship between coverage adequacy criteria and fault detection, cautioning that raw coverage percentages can be a misleading proxy for test effectiveness if not interpreted alongside defect-leakage data [15] — a finding that directly motivates our decision to pair a coverage metric with production-bug and in-sprint-bug metrics rather than reporting coverage in isolation. Traini's ethnographic study of performance-assurance practices in agile teams, and Edison, Wang, and Conboy's systematic review of large-scale agile methods, both highlight that even where quality data exists, it is frequently underused because it is not presented in a form aligned with the sprint cadence that QA managers and Scrum teams actually operate in [16,17]. Prates et al. extended metrics thinking into the DevSecOps space, showing that the same aggregation-and-visualization pattern used for delivery and quality metrics generalizes to security metrics, supporting the extensibility of the architecture proposed here [18].

Taken together, this literature establishes that (a) measurement is a recognized bottleneck in DevOps and QA maturity, (b) coverage and regression metrics are most useful when computed continuously and contextualized against sprint boundaries, and (c) no widely documented reference architecture specifically integrates qTest, Jira, and Grafana for QA-manager-facing reporting — the gap this paper addresses.

2.1. Motivation and Problem Statement

In practice, QA managers at mid-to-large organizations using Jira for sprint management and qTest for test case management face three recurring problems. First, coverage blindness: because test cases in qTest are only loosely linked to Jira requirements and stories, it is difficult to know, at any given time, which backlog items lack adequate test coverage. Second, delayed defect-leakage visibility: production bugs are logged in a separate Jira project (or a dedicated incident system) from in-sprint bugs, so the relationship between what escaped versus what was caught pre-release is usually computed manually, often only at release retrospectives. Third, automation opacity: regression automation rates are commonly tracked per project in CI tool dashboards that are not accessible to, or easily interpretable by, QA managers without engineering support.

These problems compound as organizations scale. As the number of Jira projects, qTest projects, and CI pipelines grows, manual reconciliation becomes untenable, and the reporting cadence — typically weekly or per sprint — lags behind the pace at which regressions and coverage gaps actually emerge. The result is that quality decisions, such as whether a release is ready, which modules need additional regression investment, or whether a team is under-resourced for its testing scope, are frequently made on stale or incomplete data. The solution proposed in this paper directly targets these three problems by defining a single integrated data pipeline and metric set that updates automatically and is purpose-built for QA management decision-making.

2.2. Proposed Architecture

We propose a four-layer architecture (Fig. 1): a source-systems layer, an integration and aggregation layer, a metric storage layer, and a visualization layer.

The source-systems layer comprises qTest (test case definitions, execution results, and requirement traceability links), Jira (sprint metadata, in-sprint defect issues, and story/epic hierarchy), a production incident system (which may be a dedicated Jira project or an external ITSM tool), and the CI/CD pipeline or version-control system (build results and automated test execution logs).

The integration and aggregation layer consists of scheduled connectors that call the REST APIs exposed by qTest and Jira, normalize heterogeneous field names — for example, differing custom-field configurations across Jira projects — into a common metric schema, and compute derived values such as coverage ratios and automation percentages. This layer is intentionally decoupled from the visualization layer so that metric logic can be unit-tested and versioned independently of dashboard definitions, a separation of concerns consistent with recommendations from prior DevOps-metrics tooling literature [2,9].

Computed and raw metrics are persisted in a time-series or relational metric store. A time-series database such as InfluxDB is favored for trend-oriented panels, since Grafana's native time-series query model performs efficiently against this class of store, while a relational store such as PostgreSQL is more convenient for point-in-time, join-heavy queries such as identifying stories without linked test cases.

The visualization layer is Grafana itself, configured with data-source plugins pointed at the metric store, a library of reusable panel templates (single-stat, time series, bar, heatmap, and table panels), and alert rules that fire when a metric crosses a manager-defined threshold — for example, when missing coverage on a release-candidate branch exceeds 20%, or when the in-sprint-to-production bug ratio inverts. Grafana's templating variables (e.g., $team, $release, $sprint) allow a single dashboard definition to serve multiple teams without duplication, which is important for organizations running many concurrent Jira and qTest projects.

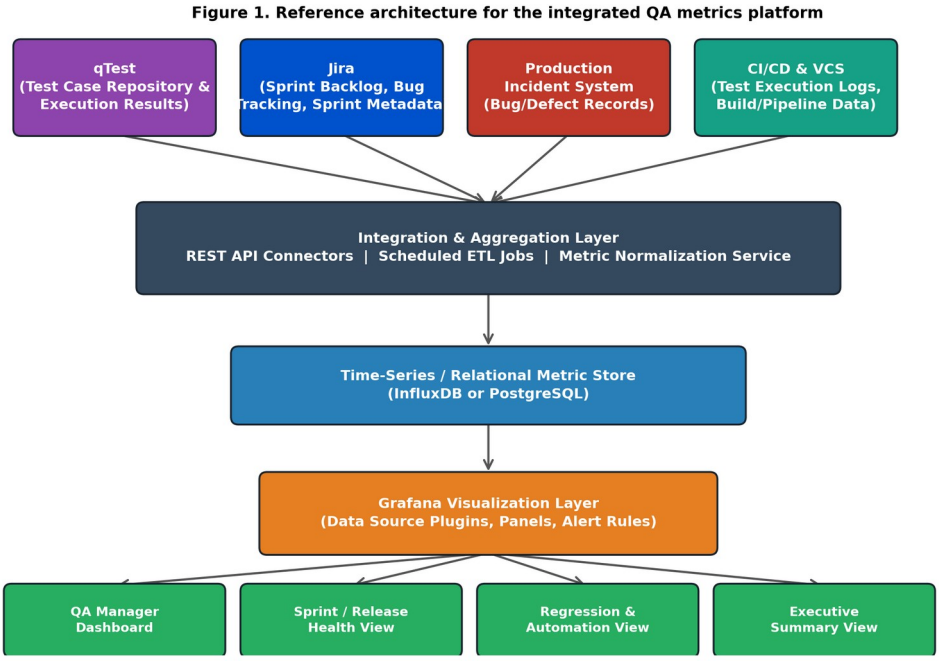

Fig. 1. Reference architecture for the integrated QA metrics platform, from source systems through the Grafana visualization layer.

2.3. QA Metrics Framework

We define seven core metrics (Table 1), each with an explicit formula and primary data source, chosen to jointly answer the three questions a QA manager repeatedly needs to answer: what is not yet tested, what escaped to production, and how mature is the automation suite.

Test coverage is computed as the percentage of requirements (Jira stories and epics) with at least one linked, executed qTest test case, rather than as code-line coverage, since the target audience is QA managers reasoning about functional and requirement coverage rather than developers reasoning about unit-test coverage. Missing test coverage is the complementary metric, reported per module or component to prioritize remediation.

Production bugs and in-sprint bugs are both counted from Jira issue queries filtered by issue type and originating environment; the ratio between them approximates a defect-leakage rate, consistent with the escaped-defect indicators identified by Gomes et al. and Amaro et al. as underrepresented in generic engineering dashboards [9,2]. In-sprint automation and regression automation are distinguished because they answer different questions: in-sprint automation measures whether new functionality is being automated as it is built (a leading indicator), while regression automation measures the cumulative automated share of the existing suite (a lagging, suite-health indicator) — a distinction supported by prioritization research showing these two populations of tests behave differently over time [11,13].

Predictability is computed as the ratio of test cases delivered or executed to test cases planned within a sprint, adapted from delivery-predictability concepts in DORA-style DevOps metrics [6] but applied specifically to testing throughput rather than to deployment cadence.

Metric	Definition / Formula	Primary Data Source	Cadence
Test Coverage	(Requirements with ≥1 linked, executed test case ÷ Total requirements) × 100	qTest ↔ Jira link, Jira	Per sprint

Metric	Definition / Formula	Primary Data Source	Cadence
Missing Test Coverage	100% − Test Coverage, broken down by module/component	qTest ↔ Jira link	Per sprint
Production Bugs	Count of defects logged against a production/released environment in a period	Jira (incident project)	Rolling 7/30 day
In-Sprint Bugs	Count of defects raised and resolved within the same active sprint	Jira	Per sprint
In-Sprint Automation	(New test cases automated in current sprint ÷ New test cases created in sprint) × 100	qTest, Jira	Per sprint
Regression Automation	(Automated regression test cases ÷ Total regression suite size) × 100, by component	qTest, CI/CD logs	Weekly
Predictability Index	Test cases delivered/executed ÷ Test cases planned at sprint start	qTest, Jira sprint report	Per sprint

Table 1. QA test-matrix metrics framework: definitions, formulas, primary data sources, and reporting cadence.

2.4. Implementation

We describe a reference implementation of the architecture in Section 4, intended as a reproducible blueprint rather than as a report of a specific production deployment. Extraction from Jira uses JQL (Jira Query Language) filters scoped by project, issue type, and sprint, retrieved through the Jira REST API on a scheduled interval (e.g., every 15 minutes for in-sprint views, hourly for release-level views). Extraction from qTest uses the qTest Manager API to retrieve test cases, test runs, execution logs, and requirement-to-test-case traceability links; because qTest's traceability model varies across editions, the integration layer includes an adapter pattern so that the metric-computation logic remains stable even if the underlying linkage mechanism changes.

A lightweight scheduler (e.g., a cron-triggered job or an orchestration tool such as Apache Airflow) executes the extraction and transformation jobs, writes normalized records to the metric store, and computes the seven KPIs from Table 1 using parameterized SQL or Flux queries. Field-mapping configuration — which Jira custom field represents 'environment,' which qTest field represents 'automation status' — is externalized to a configuration file so that the pipeline can be adapted to a new Jira or qTest project without code changes, addressing a portability concern noted in prior multi-project DevOps tooling studies [3].

Grafana dashboards are defined as provisioned JSON models under version control, which allows dashboard changes to go through the same code-review process as application code and avoids configuration drift between environments. Panels are grouped into rows corresponding to a QA manager's typical review flow: headline single-stat KPIs at the top, trend charts for coverage and automation in the middle, and drill-down heatmap and table panels for module-level detail at the bottom (Fig. 2). Alert rules are configured using Grafana's unified alerting engine and routed to the team's existing notification channel (e.g., Slack or Microsoft Teams), so that threshold breaches — such as a sudden drop in regression automation coverage after a large refactor — are surfaced proactively rather than discovered at the next scheduled review.

Fig. 2. Illustrative mock-up of the QA manager Grafana dashboard layout, showing headline KPI panels, trend charts, a module coverage-gap heatmap, and a regression automation breakdown table. Values shown are synthetic and included for layout demonstration only.

2.5. Illustrative Evaluation

To demonstrate how the proposed metrics behave over time, we constructed a synthetic eight-to-ten-sprint dataset modeled on typical mid-size team velocities, rather than reporting figures from a specific organization's confidential data. The purpose of this illustrative evaluation is to validate the descriptive and diagnostic value of the metric set and dashboard layout, not to make a generalizable empirical claim about expected coverage or defect rates in any particular organization; a controlled field evaluation with real organizational data is identified as future work in Section 8.

In the synthetic scenario, test coverage increases from 61% in sprint 1 to 82.4% by sprint 8, while regression automation rises from 40% to 68% over the same period (Fig. 2). Over the same window, production bugs decline from 11 to 6 per sprint window while in-sprint bugs decline more gradually from 22 to 14, consistent with the expected pattern in which automation investment first reduces escaped defects before overall defect discovery volume falls. The module-level coverage heatmap (Fig. 2) surfaces two components — Refunds (35% missing coverage) and Ledger (41% missing coverage) — as clear remediation priorities that would not be readily visible from an aggregate coverage percentage alone, illustrating the diagnostic value of reporting coverage at component granularity rather than only at the release level, consistent with the caution raised by Chen et al. against relying on a single aggregate coverage figure [15].

Figure 3 presents the predictability index derived from planned-versus-delivered test execution across ten synthetic sprints. The index fluctuates between 0.82 and 0.98, with two visible dips corresponding to sprints where planned scope was not fully executed. In an operational deployment, a sustained predictability index below a manager-defined threshold (for example, 0.80 over three consecutive sprints) would be configured as a Grafana alert condition, prompting a capacity or scope conversation before it affects release timelines rather than after.

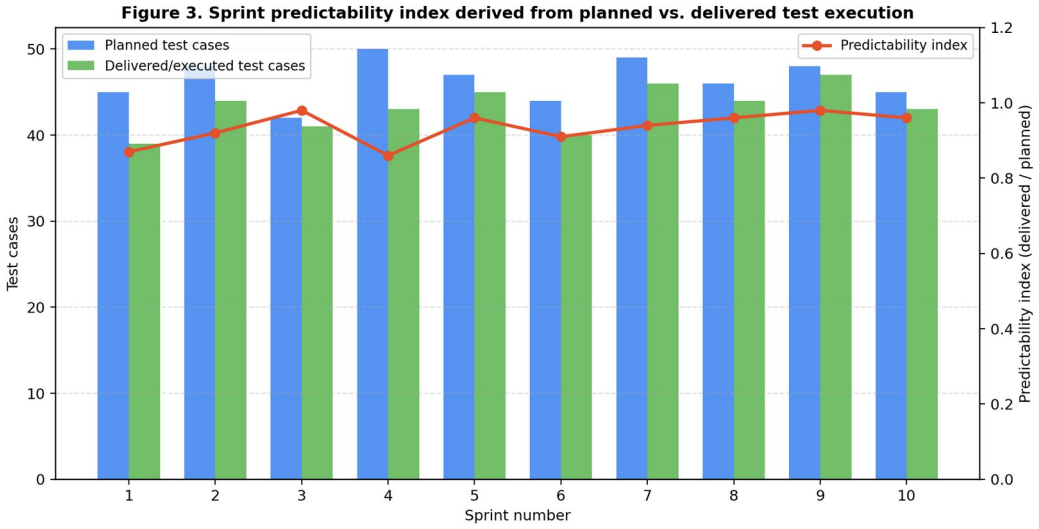

Fig. 3. Illustrative predictability index (delivered ÷ planned test execution) computed across ten synthetic sprints, plotted against planned and delivered test case volumes.

Collectively, the illustrative dataset shows that the seven-metric framework surfaces three distinct classes of insight from a single dashboard view: trend-level progress (coverage and automation growth), leakage-level risk (the in-sprint-to-production bug relationship), and commitment-level risk (predictability), which together map closely onto the three problems identified in Section 3.

2.6. Discussion and Threats to Validity

Several limitations should be considered when applying this architecture. First, the evaluation in Section 7 uses a synthetic dataset for illustrative purposes; a field study measuring actual QA manager decision-making before and after adoption, across multiple organizations, would be needed to substantiate causal claims about reporting-cycle time reduction or defect-leakage improvement, and is left as future work. Second, requirement-level test coverage is only as reliable as the discipline with which teams maintain qTest-to-Jira traceability links; organizations with inconsistent linking practices will see coverage metrics that understate actual test effort, a limitation also noted in prior coverage-metric research [15]. Third, the architecture depends on the continued availability and stability of the Jira and qTest REST APIs; vendor API changes require the adapter layer described in Section 6 to be maintained over time. Fourth, while we scope the framework to seven metrics to avoid dashboard overload, some organizations may require additional metrics (e.g., mean time to detect, flaky-test rate) not covered here; the architecture is designed to be extensible to such additions without structural change, following the same aggregation pattern demonstrated for security metrics in DevSecOps contexts [18]. Finally, alerting thresholds proposed here are illustrative starting points; miscalibrated thresholds risk alert fatigue, a known challenge in monitoring system design that should be tuned iteratively per team.

3. Conclusion

This paper presented a reference architecture and metrics framework for integrating qTest, Jira, and Grafana into a single QA test-matrix dashboard for QA managers. By defining seven complementary metrics — test coverage, missing test coverage, production bugs, in-sprint bugs, in-sprint automation, regression automation, and predictability — and describing a reproducible integration, storage, and visualization pipeline, the proposed approach addresses coverage blindness, delayed defect-leakage visibility, and automation opacity identified as recurring problems in QA reporting. An illustrative evaluation on a synthetic

multi-sprint dataset demonstrates the diagnostic value of component-level, sprint-aligned reporting over static, manually assembled reports. Future work includes a controlled field evaluation across multiple organizations, extension of the metric set to include flaky-test and mean-time-to-detect indicators, and integration of predictive models — such as those explored in test-case prioritization research [14] — to forecast coverage gaps and defect-leakage risk ahead of release rather than only reporting them descriptively.

References

[1] Alnafessah, A., Gias, A.U., Wang, R., Zhu, L., Casale, G., Filieri, A.: Quality-aware DevOps research: Where do we stand? IEEE Access 9, 44476–44489 (2021)

[2] Amaro, R., Pereira, R., da Silva, M.M.: DevOps metrics and KPIs: a multivocal literature review. ACM Comput. Surv. 56(9), 1–41 (2024)

[3] Badshah, S., Khan, A.A., Khan, B.: Towards process improvement in DevOps: a systematic literature review. In: Proc. 24th Int. Conf. Evaluation and Assessment in Software Engineering (EASE), pp. 427–433. ACM (2020)

[4] Bertolino, A., De Angelis, G., Guerriero, A., Miranda, B., Pietrantuono, R., Russo, S.: DevOpRET: Continuous reliability testing in DevOps. J. Softw. Evol. Process 35(3), e2298 (2023)

[5] Rafi, S., Yu, W., Akbar, M.A., Mahmood, S., Alsanad, A., Gumaei, A.: Readiness model for DevOps implementation in software organizations. J. Softw. Evol. Process 33(4), e2323 (2021)

[6] Sallin, M., Kropp, M., Anslow, C., Quilty, J.W., Meier, A.: Measuring software delivery performance using the four key metrics of DevOps. In: Int. Conf. Agile Software Development, LNBIP, pp. 103–119. Springer, Cham (2021)

[7] Santos, J., Alencar da Costa, D., Kulesza, U.: Investigating the impact of continuous integration practices on the productivity and quality of open-source projects. In: Proc. 16th ACM/IEEE Int. Symp. Empirical Software Engineering and Measurement (ESEM), pp. 137–147. ACM (2022)

[8] dos Santos Júnior, P.S., Perini Barcellos, M., Borges Ruy, F.: Tell me: am I going to heaven? A diagnosis instrument of continuous software engineering practices adoption. In: Proc. Evaluation and Assessment in Software Engineering (EASE), pp. 30–39. ACM (2021)

[9] Gomes, M., Pereira, R., Silva, M., Braga de Vasconcelos, J., Rocha, Á.: KPIs for evaluation of DevOps teams. In: World Conf. Information Systems and Technologies (WorldCIST), pp. 142–156. Springer, Cham (2022)

[10] Mishra, A., Otaiwi, Z.: DevOps and software quality: A systematic mapping. Comput. Sci. Rev. 38, 100308 (2020)

[11] Huang, R., Zhang, Q., Towey, D., Sun, W., Chen, J.: Regression test case prioritization by code combinations coverage. J. Syst. Softw. 169, 110712 (2020)

[12] Pradhan, D., Wang, S., Ali, S., Yue, T., Liaaen, M.: Employing rule mining and multi-objective search for dynamic test case prioritization. J. Syst. Softw. 153, 86–104 (2019)

[13] Magalhães, C., Andrade, J., Perrusi, L., Mota, A., Barros, F., Maia, E.: HSP: a hybrid selection and prioritisation of regression test cases based on information retrieval and code coverage applied on an industrial case study. J. Syst. Softw. 159, 110430 (2020)

[14] Sharif, A., Marijan, D., Liaaen, M.: DeepOrder: deep learning for test case prioritization in continuous integration testing. arXiv:2110.07443 (2021)

[15] Chen, Y.T., Gopinath, R., Tadakamalla, A., Ernst, M.D., Holmes, R., Fraser, G., Ammann, P., Just, R.: Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In: Proc. 35th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), pp. 237–249. IEEE (2020)

[16] Traini, L.: Exploring performance assurance practices and challenges in agile software development: an ethnographic study. Empir. Softw. Eng. 27, 74 (2022)

[17] Edison, H., Wang, X., Conboy, K.: Comparing methods for large-scale agile software development: a systematic literature review. IEEE Trans. Softw. Eng. 48(8), 2709–2731 (2021)

[18] Prates, L., Faustino, J., Silva, M., Pereira, R.: DevSecOps metrics. In: EuroSymposium on Systems Analysis and Design, pp. 77–90. Springer, Cham (2019)