

A Nightly Feedback-Driven Strategy for Quality Control in Safety-Critical Software Development

Simin Sun ✉ 

Chalmers University of Technology and University of Gothenburg, Sweden

Peter Nemeth ✉

Zenseact, Gothenburg, Sweden

Staffan Johansson ✉

Zenseact, Gothenburg, Sweden

Theo Wiik ✉ 

Zenseact, Gothenburg, Sweden

David Friberg ✉ 

Zenseact, Gothenburg, Sweden

Farnaz Fotrousi ✉ 

Chalmers University of Technology and University of Gothenburg, Sweden

Mirosław Staron ✉ 

Chalmers University of Technology and University of Gothenburg, Sweden

Abstract

Software quality control in industry often relies on expensive analyses that run continuously in the Continuous Integration (CI) pipeline. Those analyses, such as static analysis for rule-based compliance, can provide important quality signals and often serve as strict gating points in the CI pipeline to ensure strict quality standards. However, strict continuous gating can slow the pipeline especially in the shared development environment. This study investigates the adoption of different software quality control strategies in safety-critical domain. By changing when the analyses take place and how the information can be delivered, we aim to identify a strategy that improves CI throughput while maintaining high standards of software quality. We conducted action research at a large safety-critical software organization for autonomous driving. Through this process, we investigated existing quality strategies and designed a new strategy that combine nightly analysis with feedback mechanism. We evaluate this strategy using both longitudinal data on static analysis issues and development activities, together with interview data with practitioners. The evaluation shows that nightly CI runs together with feedback mechanism helped keep selected quality issues under control during daily development and reduced reliance on clean-up in later stages. It can also improve the teams' follow-up responsibility and development efficiency. This study contributes an empirically grounded quality control strategy that mitigates the tradeoff between strict quality gate and CI pipeline throughput in safety-critical software development. It also identifies the organizational preconditions, implementation instructions and possible social effects for applying similar mechanisms in other environments that rely on expensive quality analyses.

2012 ACM Subject Classification Software and its engineering → Software development methods

Keywords and phrases Safety-critical Systems, Software Quality Control, Continuous Integration/-Continuous Deployment (CI/CD)

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.81

Category Software Engineering in Practice Track Paper

Funding Software Center has also partially funded this work, a collaboration between the University of Gothenburg, Chalmers, and 18 universities and companies – <https://www.software-center.se>.

Acknowledgements We would like to thank Zenseact for the opportunity to conduct this project with them and providing us with all the required resources.



© Simin Sun, Peter Nemeth, Staffan Johansson, Theo Wiik, David Friberg, Farnaz Fotrousi, and Mirosław Staron;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 81; pp. 81:1–81:20



Leibniz International Proceedings in Informatics
LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Software quality control in industrial practice is assured through a combination of development activities, such as testing, code review, and automated analyses. These mechanisms help organizations detect defects, enforce standard compliance, and monitor the health of the codebase throughout development. Among them, static analysis is a common approach for product quality control [31] since it can automatically and repeatedly examine large codebases for coding-rule violations, bug-prone constructs, and other quality-related issues without executing the software. As a result, static analysis provides a mechanism for enforcing development discipline and applying quality control at scale.

In automotive software development, however, applying these controls efficiently is challenging. Modern vehicles have become more software-intensive and safety-critical, and development environments often include resource-intensive and time-consuming analyses such as compliance checks against MISRA¹ and AUTOSAR [2] rules, as well as memory-related bugs captured by tools such as Valgrind or AddressSanitizer. Although these analyses provide valuable quality signals, many of them are too costly to execute continuously on the Continuous Integration/Continuous Deployment (CI/CD) pipeline. This creates a recurring industrial trade-off: strict enforcement keeps the codebase close to release-ready, but slows development feedback cycles; deferring checks improves short-term throughput, but allows quality debt to accumulate and shifts stabilization effort to later phases.

This trade-off is especially pronounced in monorepo-based development environments, where delays in shared pipelines affect many teams and post-merge quality regressions propagate broadly. In practice, organizations often rely on one of three strategies: continuously gated CI/CD, deferred clean-up during hardening, or nightly execution outside the main integration path. Nightly execution can reduce pressure on the main CI/CD pipeline by moving expensive analyses outside the critical development flow. However, it alone does not solve the problem of follow-up. When quality results are produced outside the normal integration path, they may become less visible, less attributable, and less actionable for the teams responsible for resolving them.

Recent works [1, 6, 29] have suggested that improving CI/CD efficiency is not only a matter of relocating analyses earlier in the pipeline, but also of improving what and how information flows back to developers. Sun et al. [26] argue that traditional shift-left strategies are limited when pre-merge stages are already heavily loaded. They also propose the complementary feedback-left to improve the observability in CI/CD pipeline, arguing that increasing the amount of diagnostic information passed from the later stages of the CI/CD back to developers can improve observability. This perspective suggests that pipeline effectiveness depends not only on where checks are placed, but also on whether their results are clearly analyzed and presented for the developers who need to respond. Building on this view, the central question of this study is whether expensive quality analysis can be relocated from gating CI/CD pipeline to non-gating nightly, supported by a feedback-left mechanism, in order to improve the pipeline throughput while preserving developers' awareness and maintaining software quality.

To answer the question, we conducted an action research study at a large company developing advanced driver assistance and autonomous driving systems. This study investigates how feedback-left mechanisms can support the relocation of expensive static analysis from gated CI/CD to nightly non-gating execution in a safety-critical software organization.

¹ <https://misra.org.uk/>

In collaboration with practitioners, we designed and evaluated a strategy that combines nightly QAC analysis, dashboard-based visibility, Slack-integrated notifications, and code ownership-based responsibility mapping. The study is guided by the following research questions (RQs):

RQ1: How can computationally expensive static analysis be reorganized from gated CI/CD to nightly non-gating execution in a safety-critical software organization?

RQ2: What benefits do developers perceive from adopting a nightly feedback strategy for software quality control?

RQ3: What limitations and risks do developers perceive from adopting a nightly feedback strategy for software quality control?

The paper makes two main contributions. First, it provides an industrial action research investigation on adopting different software quality control strategies in a safety-critical software organization, including the perceived benefits and limitations of moving expensive analysis from gated CI/CD execution to nightly non-gating execution. Second, it presents and evaluates the complementary feedback mechanism that transforms nightly static-analysis results into visible, attributable, and actionable feedback through dashboards, internal communication channels, and code guardianship-based responsibility mapping.

2 Background and Related Work

2.1 Software Quality Control Techniques

In industrial software development, software quality management depends on a set of coordinated activities, including quality planning, quality assurance, quality control, and process improvement [31].

In real projects, software quality characteristics can be examined using techniques such as testing, inspections, defect tracking, root cause analysis, and static analysis [31]. Among these techniques, static analysis is particularly popular and useful because it examines source code or other software artifacts without executing the program. This makes static analysis suitable for early, repeated and automated use during development, since it does not require the real execution.

Compared to manual inspection, static analysis tools can automate code inspections and help to find bugs, or defects, faster and cheaper, but their usages and adoptions are limited by issues such as result understandability and integration of the development workflow [13]. There are many popular commercial and open source tools for this purpose, including Cppcheck, FindBugs, SonarQube, PMD [25]. Lenarduzzi et al. [15] compared six static analysis tools in detection, agreement and precision. In industrial settings, tool choice may also be influenced by factors, such as safety standards, compliance requirements, and support for large-scale development workflows. For example, Emanuelsson and Nilsson [7] compared PolySpace Verifier, Coverity Prevent, and Klocwork K7, while QAC has frequently been discussed in relation to MISRA compliance for C and C++ systems [4, 12].

In modern development environments, static analysis is often integrated into Continuous Integration and Continuous Deployment (CI/CD) pipelines. This integration allows quality checks to be triggered automatically when code changes are submitted. However, not all analyses are equally suitable for continuous execution in the critical integration path. Some checks are computationally expensive, require scarce resources such as licenses, or produce feedback too slowly to be practical as always-on gates. This creates a placement problem: organizations must decide whether expensive analyses should block integration, run after integration, or be deferred to scheduled jobs such as nightly pipelines.

2.2 Quality Control Strategies in CI/CD

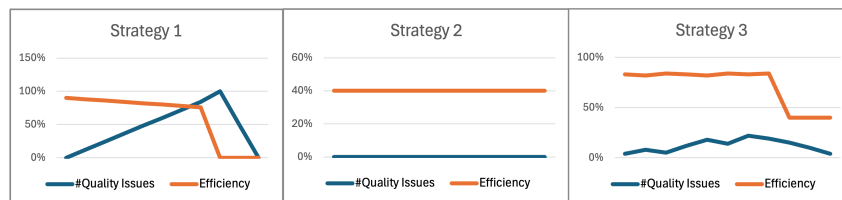
In this study, we distinguish three quality control strategies that reflect different ways of managing the trade-off between strict quality control and development throughput. These strategies differ in when quality checks are executed, whether they block development, and how feedback reaches developers.

The first is *Build-up & Clean-up (Strategy 1)*. This was historically more common in traditional automotive development, before the era of contemporary DevOps-driven practices. It emphasizes long development cycles in which feature growth may proceed quickly for months, followed by a lengthy hardening phase needed to remove accumulated violations. This method decouples fixes from the time when the original code was designed, which risks diluting the intent of the guidelines being enforced.

The second is *Always Gated CI/CD (Strategy 2)*, where quality checks are enforced continuously, often before changes are merged. This can keep the codebase close to a release-ready state at all time, but it may also reduce developer efficiency when checks are expensive, unstable, or too frequent. Strategy 2 can be optimized to some extent, for example by dividing the analyses and placing some of them in pre-merge and deferring the rest parts to post-merge, but the overall throughput remains low. Prior work shows that CI jobs and failures create productivity costs, such as long build durations and frequent breakages, hindering developer work [11, 10]. Similarly, pre-merge gating focuses on early failure preventions but can introduce risks such as longer build time and bottlenecks in the development pipeline [26, 21]. Together, these studies highlight the need to balance quality assurance with development efficiency in CI/CD environments and motivate the third strategy.

In *Nightly Non-gating Analysis (Strategy 3)*, instead of running all expensive checks in the critical integration path, selected analyses are scheduled during off-peak periods when CI/CD demand is lower. This can reduce the immediate burden on developers and improve the throughput of the main CI/CD pipeline [26, 5]. More generally, monitoring CI practices can support software quality and project communication [18]. However, nightly jobs also introduce a feedback challenge. Because results arrive outside the normal integration flow, they may be less visible and less urgent. In addition, nightly jobs often analyze the accumulated changes of many developers rather than a single commit, which can make accountability more difficult. As a result, issues detected by nightly jobs may remain unresolved and contribute to later quality debt.

Figure 1 illustrates the structural dilemma among the three strategies.



■ **Figure 1** Structural dilemma between quality-related issues and efficiency under three strategies. In Strategy 1, developer efficiency starts high but gradually decreases as quality-related issues accumulate. In Strategy 2, developer efficiency remains low, while quality-related issues stay at zero throughout the process. In Strategy 3, developer efficiency initially remains high while quality-related issues slowly accumulate, but both decline upon entering the hardening phase.

2.3 Feedback and Accountability

When quality control analyses are moved from gated CI/CD execution to nightly non-gating jobs, the main challenge shifts from execution cost to follow-up. In gated CI/CD, feedback is naturally visible because developers receive it as part of the integration process, and unresolved issues directly block progress. In nightly non-gating analysis, however, results are produced outside the normal development flow. As a result, developers may not notice the findings, may not know whether the findings concern their code, or may not treat them as urgent. This makes feedback mechanisms and accountability practices central to the effectiveness of non-gating quality control.

Prior studies show that developers value CI monitoring for build failures, code quality, and related metrics, but that native CI services often lack sufficiently comprehensive monitoring support [30, 19]. To address this limitation, dashboards that continuously monitor and visualize few indicators during software development can be a solution [23]. Dashboards can support awareness by making project status, quality indicators, and bottlenecks continuously available to development teams [28, 14]. Staron et al. [24] provide a dashboard selection model for choosing the right visualization and maximizing the impacts. However, dashboards are often passive mechanisms because developers must remember to visit them, interpret the information, and decide whether action is needed.

The other choice is notification bots, which provide a more active form of feedback by pushing selected information into developers' existing communication channels. Code review bots and CI bots can automate tasks such as reporting CI failures, checking code quality, and tracking code coverage [32]. Such bots can reduce manual effort and provide timely feedback. At the same time, prior work shows that developers may resist bots when they generate excessive notifications or create communication noise [16]. Therefore, effective bot-based feedback must balance visibility with relevance.

Making feedback visible is not sufficient if responsibility for follow-up remains unclear. Prior work on code ownership has shown that modules with clear ownership tend to have fewer defects than modules with shared or weak ownership [9, 3, 8]. In settings with weak or collective ownership, multiple developers contribute to the same components, making accountability less clear and increasing the risk of technical debt [20]. At the same time, strong ownership is not a universal solution, especially at the beginning of the adoption. Sundelin et al. [27] argue that compared with strict ownership models, self-selected ownership can actually contribute to fewer code clones.

Taken together, those studies suggest that nightly non-gating analysis requires more than scheduled execution. To support follow-up, results must be made visible through appropriate feedback channels, attributable through ownership information.

3 Research Methodology

We employed action research because the goal of the study was not only to observe existing quality-control practices, but also to intervene in an industrial setting and learn from the effects of that intervention. Action research is suitable for our study because it has been developed to combine learning, researching and making actions [22]. It is especially useful in industry-academia collaboration where the researchers and practitioners working together for a shared goal [17]. An action research cycle normally includes five steps, starting with diagnosing, action planning, action taking, evaluating and end with learning [17, 22].

For this study, we collaborated with a team from Zenseact for one year. The timeline for this project is listed as Table 1.

■ **Table 1** Overview of the action research timeline.

Cycle	Diagnosis	Action planning	Action taking	Evaluation	Learning
1	The existing strategy used quality-related signals as gating mechanisms in the CI pipeline, which reduced pipeline throughput.	Adopt Strategy 3 in practice.	Move quality-related from gated CI/CD execution to non-gating nightly execution.	The throughput rate of CI increases as well as the number of issues during the development.	The nightly job lacks output illustration and makes it less obvious for developers to follow up.
2	Moving quality-related signals from gated CI pipeline to non-gating nightly execution improved throughput, but decrease the visibility of results.	Introduce a dashboard to present the nightly analysis results.	Develop a dashboard for viewing the results of nightly analysis.	The dashboard provides access but the usage is low among developers.	The dashboard lacks proactive features to engage developers.
3	The dashboard provides access to analysis results, but its passive nature did not prompt immediate developer follow-up.	Introduce active notifications to increase developer awareness.	Introduce Slack-integrated notification bot based on code guardianship.	The bot leads to increased visit traffic with the dashboard and quick responses.	The current approach improves the CI throughput while preserving developers' quality awareness.

3.1 Diagnosis

The collaborating company, Zenseact, is a Swedish company develops advanced driver assistance and autonomous driving software in a safety-critical setting. The main programming language for product development is C++, and the organization uses a monorepo-based development environment with shared CI pipelines and nightly quality-analysis jobs. Static analysis is performed using Perforce Helix QAC (QAC), to check compliance with internal coding rules and industry standards (e.g., ISO 26262).

The company manages the code responsibility through code guardianship instead of the strict code ownership to create a collaborative rather than hierarchical culture [27]. Zenseact utilizes the concept of code guardianship to create a shared responsibility across components, this is a more relaxed form of code ownership with a coarser granularity and self-selected responsibility. In this setting, a team instead of a single developer is self-reported to be the main owner of a part of the code.

During the diagnosis phase, practitioners described a recurring tension between strict quality control and development throughput. To maintain high software quality in a safety-critical setting, the company used several automated checks in the integration pipeline, including compliance and code-quality checks. However, some of these checks were expensive to run and could reduce the throughput of the shared CI pipeline, especially in the monorepo setting where delays affect many teams. The central problem identified in the diagnosis was therefore how to maintain strict quality standards while reducing the cost of expensive analyses in the critical CI/CD path.

The action research team in this study consisted of three researchers and five practitioners. The practitioners contributed domain knowledge, access to development practices, and feedback on the intervention, while the researchers supported problem formulation, data collection, analysis, and reflection.

3.2 Action Planning

After identifying the problem during the diagnosis phase, we plan actions to address it. The planning was guided by three requirements. First, the non-negotiable requirement from the company is the “zero-defeat” threshold, meaning that selected quality issues could not remain unresolved before release. Second, the organization wanted to reduce the accumulation of quality issues during daily development, rather than relying mainly on late-stage clean-up. Third, the intervention should reduce the burden of expensive analyses on the critical CI/CD path and thereby support higher development throughput.

To meet those requirements, different software quality control strategies (as mentioned in Section 2) have been studied. Strategy 1, build-up and clean-up, was considered unsuitable because it conflicts with the company’s need for strict high quality threshold and continuous awareness of safety-related quality issues. Strategy 2, always gated CI/CD, provided strict enforcement but imposed high costs on the shared CI/CD pipeline. Strategy 3, nightly non-gating analysis, could reduce the burden on the critical CI/CD path, but risked making results less visible and weakening follow-up.

Based on this analysis, the team planned a fourth strategy: *Nightly Analysis with Feedback-left*. The strategy combines nightly non-gating execution of expensive analyses with feedback mechanisms that make results visible, attributable, and actionable. In contrast to passive nightly reporting, the planned intervention would connect QAC findings to dashboard views, team communication channels, and code guardianship mappings. The goal was to preserve the development-efficiency benefits of non-gating analysis while supporting timely follow-up by responsible teams.

3.3 Action Taking

The action-taking phase implemented the planned Strategy 4. The intervention consisted of three main components: moving QAC to nightly non-gating execution, providing dashboard-based visibility, and introducing Slack-integrated notifications based on code guardianship.

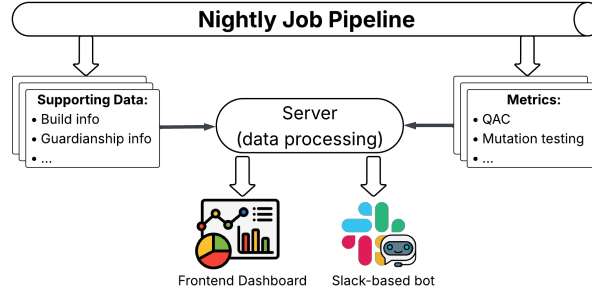
First, QAC was moved from the normal gated CI/CD pipeline to a nightly non-gating pipeline. This change removed the expensive analysis from the critical integration path while preserving regular execution of the check. The nightly job produced analysis results that could be used to monitor coding-rule compliance and other selected quality issues.

Second, a central code-health dashboard was developed to make the nightly results accessible. The dashboard transformed raw QAC output into a navigable representation of the codebase. Developers could inspect quality findings at both team level and file level, trace findings to affected files, and identify the guardian teams associated with those files. The dashboard therefore aimed to provide an information source comparable to the feedback previously available through gated CI/CD results, but without blocking the main pipeline.

Third, a Slack-integrated notification bot was introduced to make the feedback more active. While the dashboard made information available, it still required developers to visit it manually. The bot addressed this limitation by sending targeted notifications to internal team channels after nightly analysis results had been processed. These notifications focused on newly introduced QAC issues and linked them to the relevant guardian teams. The purpose was to increase awareness of non-gating findings and support timely follow-up without reintroducing expensive checks as blocking gates.

Figure 2 illustrates the implementation of Strategy 4. In this strategy, the feedback mechanism collects nightly analysis results together with runtime and metrics-related data from the nightly pipeline, processes the data on a central server, and maps the findings

to affected files and guardian teams. The processed results are then presented through a dashboard and a notification bot. In this way, the intervention transforms raw analysis outputs into summarized feedback that developers can act upon.



■ **Figure 2** Implementation of Strategy 4: Nightly Analysis with Feedback-Left. Runtime and metrics-related data are extracted from the pipeline, processed and analyzed on a central server. The resulting feedback is then published through the dashboard and internal communication channels.

3.4 Evaluation & Specifying Learning

To evaluate the impact of the action taken, we collected both quantitative and qualitative data to study how the nightly feedback mechanism functioned in practice and how it was perceived by developers.

The quantitative data were gathered from the software delivery environment over a longitudinal observation period of ten months (2025/06-2026/03). These data included repository activity, guardianship information, QAC analysis results, dashboard usage, and notification-related data. We used the number of successful merges to non-master branches as a proxy indicator of integration activity. We used the number of unresolved QAC issues to examine how selected quality findings evolved over time. The absolute number of merged commits and QAC issues will be normalized using z-scores (Equation 1) to preserve confidentiality.

$$z_i = \frac{x_i - \mu}{\sigma} \quad (1)$$

where: (x_i) is the original value, (μ) is the mean of all values, (σ) is the standard deviation of all values, (z_i) is the normalized (z-score) value.

Dashboard traffic was used to assess whether developers actively accessed the passive feedback mechanism, and notification-response data were used to understand follow-up behavior after active feedback was introduced. Because the study was conducted in a real industrial environment, these quantitative measures were interpreted as indicators rather than controlled causal measurements.

To complement the quantitative data with qualitative data, we conducted semi-structured interviews with practitioners from teams affected by the nightly feedback process. Based on their team roles and their years of software development experience, ten participants were selected to represent a wide range of practitioner perspectives. The interviews focused on how participants experienced the new strategy in daily work, what benefits they perceived, what limitations or risks they experienced, and what kinds of quality-related signals they considered suitable for similar feedback mechanisms. The interview data were analyzed using thematic narrative analysis to capture participants' experiences, perceptions, and interpretations of the process.

■ **Table 2** Demographic information about participators.

ID	Role	Years of experience
P1	Software Developer	28
P2	Software Developer	14
P3	Software Developer	19
P4	Software Developer	10
P5	Technical Expert	12
P6	Software Developer	3
P7	Product Owner	20
P8	Operational Flow Lead	7
P9	Software Developer	30
P10	Software Developer	14

In this study, learning was specified through joint interpretation of quantitative trends, practitioner feedback, and interview results.

The evaluation results and lessons learned are presented in Section 4 as empirical findings and further discussed in Section 6.

4 Results

4.1 RQ1

RQ1 investigates how expensive static analysis can be reorganized from gated CI/CD to nightly non-gating execution in a safety-critical software organization. The action research process showed that nightly execution had to be combined with feedback mechanisms that preserved visibility, attribution, and follow-up. The resulting strategy consisted of three connected components: 1) moving from pre-merge gating to nightly non-gating, 2) providing dashboard-based visibility, and 3) introducing active notifications.

1) Moving from pre-merge gating to nightly non-gating. In December, QAC was moved from pre-merge gating to nightly, non-gating execution. The purpose of this change was to reduce the cost of expensive static analysis in the critical integration path while preserving regular analysis of the codebase.

To examine the effects of this change, we first analyzed the number of successful merges to non-master branches. As shown in Figure 3, the number of commits merged to non-master branches increased after QAC was moved to nightly execution. The observed increase suggests an improvement in CI/CD pipeline throughput. However, this improvement cannot be attributed solely to the removal of QAC from pre-merge gating, as participants noted that other concurrent factors may also have contributed. Nevertheless, many participants regarded the change as an important contributor to the improved throughput.

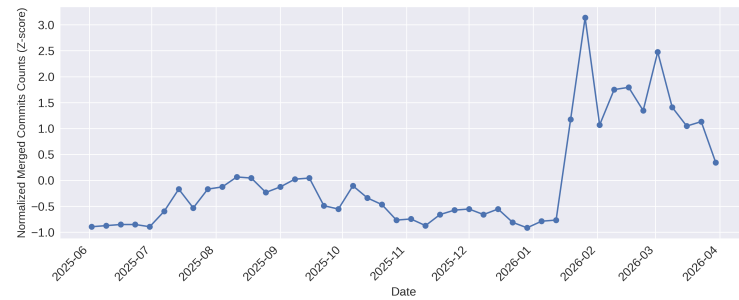
(P3)...type of commits made to the release branches. Some arbitrary category ideas:

1. New feature development 2. QAC fixes. 3. Test coverage fixes. 4. Bug fixes. 5. Documentation fixes.

(P4) Removing QAC from premerge is definitely a contender/contributor here.

These results suggest that moving expensive static analysis out of pre-merge gating can reduce pressure on shared CI/CD pipelines and support higher integration throughput. However, when we examined the number of unresolved QAC issues during the same period (the exact number of unresolved issues was normalized using z-scores), the data show that this change also introduced a follow-up challenge. As shown in Figure 4, unresolved QAC issues

81:10 A Nightly Feedback-Driven Strategy for Quality Control



■ **Figure 3** Normalized number of commits that were merged to non-master branches.

increased after the move to nightly execution. This indicates a trade-off: moving QAC out of gating pre-merge was associated with higher integration throughput, but also with greater accumulation of unresolved quality issues. Interview data supported this interpretation:



■ **Figure 4** Normalized number of QAC issues.

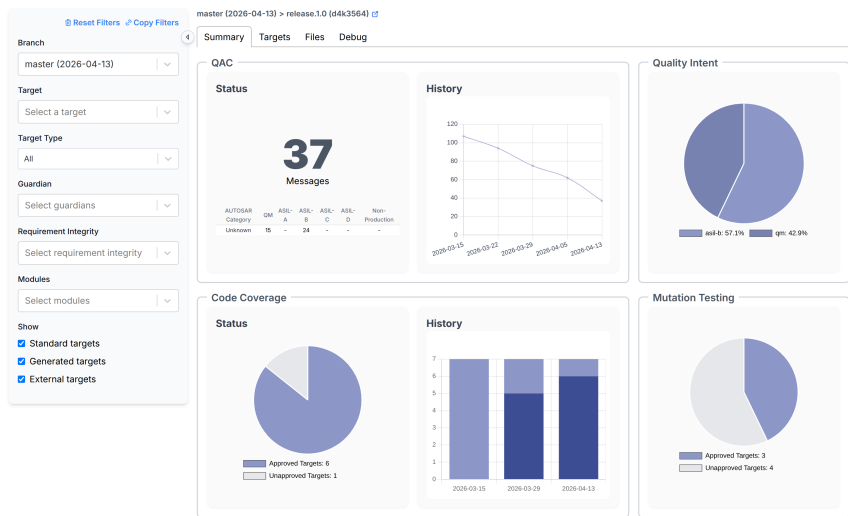
(P8) My anecdotal evidence is that we have had a lack of follow up on QAC and Code Coverage which has led to many patches having to be brought towards the release branches.

This finding shows that nightly non-gating execution alone was insufficient in the studied safety-critical setting. Once QAC no longer blocked integration, additional mechanisms were needed to ensure that findings remained visible and were followed up by the responsible teams.

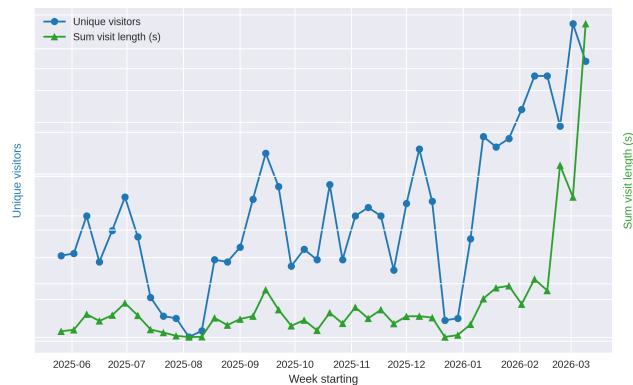
2) Providing dashboard-based visibility. To address the loss of immediate CI/CD feedback, the action research team developed a central code-health dashboard. The dashboard transformed nightly QAC results into a navigable view of the codebase. Developers could inspect findings at team and file level, trace issues to affected files, and identify the guardian teams associated with those files. An overview of the dashboard is shown in Figure 5.

The dashboard provided an equivalent information source outside the gated CI/CD path. In this sense, it restored access to information that had previously been exposed through pre-merge failures. However, dashboard usage data (Figure 6) showed that passive availability was not enough as developers did not frequently visit the dashboard after launching (2025/10–2025/12). It suggests that a dashboard can support transparency, but it cannot by itself guarantee timely follow-up.

Developers also expressed the need for more direct feedback through existing communication channels:



■ **Figure 5** Overview of the dashboard using mock data to preserve confidentiality. The homepage provides an overview of software metrics and analysis results, along with filtering options for branches, targets, guardians, and related attributes. The dashboard also offers multiple views for exploring affected targets and files, as well as supporting code inspection and debugging activities.



■ **Figure 6** Weekly traffic: unique visitors and total visit length.

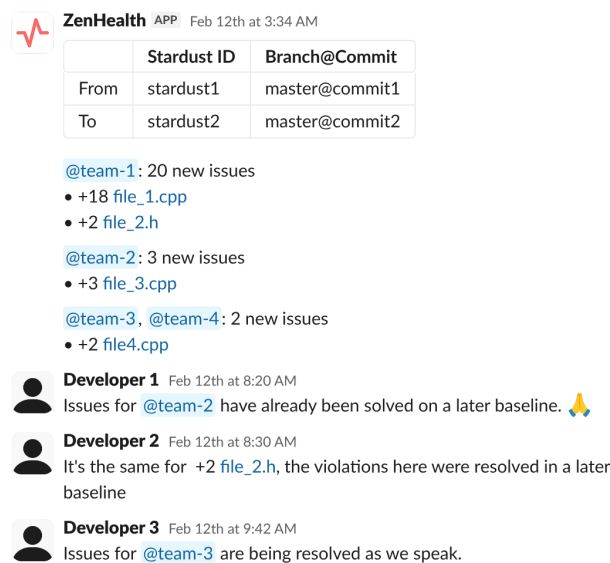
(P1) Can the system send slack notifications to a team’s channel (if configured) for post-merge errors?

This motivated the intervention in the next cycle: active notifications.

3) Introducing active notifications. On January 23, 2026, the organization introduced a Slack-integrated notification bot. The bot changed the feedback mechanism from passive access to active delivery. Instead of requiring developers to manually visit the dashboard, the bot sent targeted notifications to internal team channels after nightly analysis results had been processed. An example notification is shown in Figure 7.

The notifications focused on newly introduced QAC issues and used code guardianship information to route findings to the relevant teams. This made the nightly feedback more attributable because teams could see which findings were connected to the parts of the codebase they guarded. It also made the feedback more actionable because notifications linked developers to the relevant issue information.

81:12 A Nightly Feedback-Driven Strategy for Quality Control



■ **Figure 7** A screenshot of the Slack notification.

After the bot was introduced, unresolved QAC issues fluctuated in the short term but then decreased and remained more stable in a saw-tooth pattern, as shown after January in Figure 4. This pattern is consistent with more regular follow-up on newly introduced issues rather than continued accumulation. Developers also described the bot as focused and accurate:

(P1) I think the bot does exactly what I envisioned, using the guardian system to get focused pings on new violations.

In summary, the answer to RQ1 is that expensive static analysis can be reorganized from gated CI/CD to nightly non-gating execution, but complementary feedback mechanisms are required to preserve developer awareness and responsibility. In our study, the nightly results were made visible through a dashboard, actively delivered through Slack notifications, and attributed to responsible teams through code guardianship. They all have their own disadvantages, but the combination of all three enhances the immediate follow-up.

4.2 RQ2

RQ2 investigates what benefits developers perceived from adopting the nightly feedback strategy for software quality control. The interviews showed that developers perceived the strategy as beneficial mainly because it made non-gating QAC findings easier to notice, easier to assign to responsible teams, and easier to follow up without reintroducing expensive checks into the critical CI/CD path.

Less manual checking. A recurring benefit was that the strategy reduced the need for developers to manually inspect quality control results. Before the Slack-integrated feedback was introduced, developers had to actively visit the dashboard or rely on manual reminders to identify new QAC issues. Participants described this as inefficient because quality findings could remain unnoticed when they were only available through passive channels.

After the bot was introduced, developers received targeted messages in their existing team communication channels. This made follow-up more immediate and less dependent on developers remembering to check the dashboard:

(P5) Instead of relying on the somewhat inefficient manual process of “please check the dashboard daily for new debt”, we now see teams engaging in fixing newly discovered debt ASAP, prompted by the bot’s guardianship-directed messages.

This suggests that developers perceived the active feedback mechanism as improving awareness of newly introduced quality issues. The benefit was not only that the information was available, but that it reached developers in a place where they already coordinated daily work.

Focused and relevant feedback. Developers also perceived the feedback as useful because it was focused on relevant issues rather than broad status reporting. The bot notified teams about newly introduced QAC violations and used guardianship information to direct these notifications to the teams associated with the affected files. This reduced the risk that developers would receive unrelated quality information.

Participants described the notifications as focused and aligned with the intended use of the guardianship system and the accuracy of the notifications:

(P1) I think the bot does exactly what I envisioned, using the guardian system to get focused pings on new violations.

(P4) As for my experiences with the QAC bot in particular, it’s very good. It’s accurate, and when my team gets notified, it’s always been a true positive.

These perceptions indicate that developers valued the selectiveness of the feedback. Rather than treating nightly analysis as a general report that all developers had to inspect, the new strategy filtered and routed information so that notifications were more likely to be relevant to the receiving team.

Quality awareness. Finally, developers perceived the strategy as useful because it supported quality awareness while allowing expensive analysis to remain outside the critical CI/CD path. In the studied safety-critical setting, selected quality issues could not be ignored before release. At the same time, running expensive analyses as pre-merge gates created costs for development throughput. The nightly feedback strategy was perceived as a compromise between these concerns.

One participant described the importance of maintaining a complete view of non-negotiable quality metrics:

(P5) In safety-critical contexts, some software quality metrics are non-negotiable. We cannot ship to production with technical debt in these areas... the new strategy gives us a complete view of our delivery status against these metrics, providing a single source of truth.

The answer to RQ2 is that developers perceived this strategy as useful not simply because QAC was no longer a pre-merge gate, but also the value in separating the execution of expensive analysis from the communication and follow-up of its results. This strategy allowed the organization to keep expensive checks outside the critical integration path while still maintaining issues’ visibility and developers’ awareness.

4.3 RQ3

RQ3 investigates what limitations developers perceived from adopting the nightly feedback strategy for software quality control. Although participants generally perceived the strategy as useful, they also identified limitations related to message scope, notification noise, and feedback latency.

Message scope trade-off. One limitation concerned the scope of information included in Slack notifications. The implemented bot focused on newly introduced QAC issues. This design was intended to make notifications concise and action-oriented by highlighting regressions that required follow-up. However, some developers wanted notifications to include the current overall state in addition to newly introduced issues:

(P7) The only specific feedback has been to have also the current state in the messages not only regressions.

(P10) Would it be possible to have the existing issues be posted as well?

These requests indicate that developers sometimes need more contextual information than regression-only notifications provide. Knowing only that new issues have appeared may not be sufficient for understanding the broader quality situation of a component or team. At the same time, including the full current state in every notification could make messages longer, more repetitive, and less focused. This created a design trade-off. Regression-focused notifications supported concise and actionable feedback, while full-state information supported broader situational awareness. In the studied strategy, this trade-off was handled by using Slack notifications for newly introduced issues and the dashboard for complete status information. This separation suggests that push-based channels are better suited for concise alerts, while dashboards are better suited for detailed exploration.

Notification fatigue. A second perceived risk was notification fatigue. Although active notifications improved visibility, participants were concerned that sending too much information could make the feedback channel noisy. This was especially relevant for repeated notifications about unchanged issues:

(P6) I wonder if it would become too noisy? Like the same 50 messages would be posted the last three analyses. Our idea was that this would notify when it has gotten worse, and you can open the UI to see all.

This concern shows that increasing visibility does not automatically improve follow-up. If notifications are too frequent, too broad, or too repetitive, developers may start ignoring them. In that case, the mechanism intended to improve awareness could instead reduce attention to quality feedback.

The perceived risk of notification fatigue supports the design choice to notify mainly about regressions rather than all existing issues. It also suggests that similar feedback mechanisms should provide configurable notification policies, such as allowing teams to choose notification frequency, thresholds, channels, or types of quality signals. Such configuration could help preserve relevance while reducing unnecessary interruption.

Feedback latency and stale notifications. A third limitation concerns the delay between code changes and feedback. Because QAC execution is computationally expensive and depends on available resources, codebase size, and analysis scope, nightly analysis results may only become available several hours after the relevant code changes have been introduced.

(P9) The running time of QAC can in the ballpark of 12 hrs.

In some cases, this meant that developers received notifications about issues that had already been fixed. This delay can weaken the immediacy of feedback and may make it more difficult for developers to connect reported issues to specific changes.

(P4) The only downside I can think of is the delay, which sometimes causes it to flag violations that have already been resolved.

This latency reduces the immediacy of the feedback and can make notifications appear stale. The delay may come not only from analysis runtime, but also from practical constraints such as licensing issues, queueing, and the time needed to collect and process results before notifications are sent. As a result, the feedback may not always reflect the latest repository state.

This limitation is inherent to some extent in moving expensive analysis outside the critical CI/CD path. Nightly non-gating execution improves development throughput by avoiding immediate blocking, but it also weakens the temporal connection between a code change and the feedback produced by static analysis. Therefore, nightly feedback should be understood as a complement to faster checks rather than a complete replacement for all pre-merge quality control.

Overall, the answer to RQ3 is that developers perceived the nightly feedback strategy as useful but sensitive to feedback design. The main risks were that notifications could become either too narrow to provide sufficient context or too broad to remain useful, that repeated alerts could create notification fatigue, and that long analysis times could make feedback stale. These limitations indicate that nightly feedback mechanisms must balance visibility with relevance and timeliness.

5 Threats to Validity

A threat to **conclusion validity** arises from the extended study period of one year. During this time, multiple events occurred, including product releases, organizational restructuring, and holiday periods, all of which may have influenced the observed results. To mitigate this, we analyzed outliers together with team members and identified alternative factors that could explain variations beyond the studied variables. Nevertheless, it remains difficult to fully isolate the effects of the studied factors from these external influences.

The primary threat to **external validity** stems from the several preconditions of the proposed framework on several preconditions in the studied environment: (1) clear code ownership, (2) a mono-repository setup, (3) a mature CI/CD and nightly pipeline, and (4) established internal communication tools. These conditions tightly couple the framework to the specific industrial context, thereby limiting its generalizability to other settings. In addition, this is a single-company case study, which further constrains the applicability of the findings across organizations and domains. Due to confidentiality constraints, we are unable to replicate the exact setup in other environments within this study. As future work, we plan to conduct similar studies in additional companies to further validate the approach.

One threat to **construct validity** is the relatively short time span between moving QAC from the CI/CD pipeline and releasing the bot. Consequently, the observation period for assessing dashboard-related effects is limited, and delayed responses may not be fully captured. We mitigate this by focusing on trends rather than absolute values. Another threat concerns the measurement of perceived benefits through interviews with employees. The selected participants were primarily developers most affected by the strict threshold, which

may introduce bias in their responses. As a result, the measured perceived benefits may not fully represent the perspectives of other stakeholders. In addition, the responses primarily focused on bot notifications rather than the dashboard, making it difficult to distinguish the individual effects of each component. To address this, we evaluate the combined effect of both tools as a single unit when answering RQ3.

A key threat to **internal validity** is the potential influence of organizational and pipeline maturation over time, independent of the studied factors. To reduce this risk, we restricted the study period to twelve months, aiming to limit the impact of long-term evolutionary changes.

6 Discussion

6.1 Visible, Attributable, Actionable Feedback

The findings of this study suggest that the effectiveness of non-blocking feedback from nightly analyses depends on how those results are presented to developers. In the studied case, three key properties were especially important: visible, attributable, and actionable.

Visible feedback. must reach developers through channels that fit their daily work. Dashboards can provide a comprehensive overview, but they may not be effective when used alone as they require developers to actively check them. Notification bots increase the visibility by pushing relevant updates into existing communication channels, but their usefulness depends on the frequency and content of the notifications. We also hear requests from developers to create a CLI tool to support the visibility, which is also a good option to consider when designing the feedback mechanism. This suggests that effective visibility requires selective presentation of the most relevant information and tools that can cause developers' awareness.

Attributable feedback. make it clear who should be responsible for following up on the identified issues. In practice, this means we need to link the findings to responsible teams or developers. In the studied case, the concept of code guardianship provided a natural way to attribute the results to specific teams, which made the process much more efficient. If such mechanism or information is not available in the organization, this can also be done manually with a dedicated team or person, but it is more effective to automate this process and we recommend to adopt similar code ownership practices.

Actionable feedback. must support sufficient follow-up once the issues are identified and visible to the responsible team or developers. In practice, this means that the issues should be clear presented and their emergence should be easily assessed. What's more, it should also be easy to trace the original files and relevant information to support the diagnose of the issues. Actionability is reduced when findings are hard to interpret, disconnected from the related code, or when the results are aggregated in ways that make it difficult to identify specific problems.

Taken together, feedback-left is not simply about feeding back the results of nightly analyses to the developers faster, but also about what and how to feed the results back to the developers.

6.2 Other Metrics

There are many quality-related signals existing in CI pipeline and can be moved to nightly CI jobs to enhance the development efficiency. In this study, we focused on QAC as the example metric to study the effect of moving a metric from gating (daily CI) to non-gating (nightly CI), and the effect of the feedback mechanism. However, there are many other metrics that can be moved to nightly jobs, and the feedback mechanism can also be applied to other metrics. Code coverage is one of the most frequently mentioned metrics during the interviews, and there are discussions on whether it should be moved to non-gating as well.

(P2) I think we should move code coverage to non-gating. ... The code coverage files could be retired other than for justifying why we should actually have lower coverage than the limit.

Their main motivation of code coverage should be moved is mentioned as follows, and we believe code coverage is not the only case, in general, metrics have those properties hinder the current pipeline should be considered for moving to non-gating.

(P2) First, it (code coverage) is slow to run. Second, it's prone to A+B errors. Third, it adds another build configuration creating cache churn. Fourth, it's overly sensitive to minor changes.

6.3 Behavioral and Social Effects

Our findings also suggest that the effectiveness of the feedback mechanism is not only a technical issue, but also a behavioral and social one. One important factor is that the developers have clear understanding of organizational expectations of zero issue and the consequences of not meeting them. In the studied case, the concept of “zero defeat” provided a clear standard for code quality, which made it easier for developers to understand the importance of addressing issues promptly. This shared understanding of quality standards appears to have made the feedback easier to accept as part of daily work.

A second factor is the social visibility of the feedback. Because the selected results were communicated through shared internal channels and tagged to responsible teams. This social visibility may have created a “positive shame factor” that motivated teams to address issues more promptly. In our interpretation, this did not function primarily as blame, but rather as a form of social accountability that encouraged teams to maintain their reputation for quality. Therefore, when other organizations consider implementing similar feedback mechanisms, they should prompt the collaborative setting in which follow-up is socially encouraged without becoming individualized blame.

7 Conclusion

In this study, we adopted an action research approach to explore different software quality control strategies in safety-critical settings, along with their benefits and limitations. The main contribution of this paper is a new strategy that combines nightly analysis with a feedback-left approach to balance CI throughput and strict quality standards. By shifting selected quality related signals from gated CI pipelines to non-gating nightly analysis, the strategy improves developer efficiency during the development phase. In combination with targeted feedback notifications and supporting dashboards, the approach also maintains high quality awareness and preserves timely follow-up actions. For future work, we plan to evaluate this strategy using additional software quality metrics and apply it across different organizational contexts.

References

- 1 Sumit Asthana, Hitesh Sajnani, Elena Voyloshnikova, Birendra Acharya, and Kim Herzig. A case study of developer bots: Motivations, perceptions, and challenges. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 1268–1280, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3611643.3616248.
- 2 AUTOSAR. *Guidelines for the use of the C++14 language in critical and safety-related systems*, 2019. AUTOSAR Standard, Document ID 839, available as a PDF. URL: https://www.autosar.org/fileadmin/standards/R19-03/AP/AUTOSAR_RS_CPP14Guidelines.pdf.
- 3 Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don’t touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE ’11, pages 4–14, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2025113.2025119.
- 4 Cathal Boogerd and Leon Moonen. Assessing the value of coding standards: An empirical study. In *2008 IEEE International Conference on Software Maintenance*, pages 277–286, 2008. doi:10.1109/ICSM.2008.4658076.
- 5 Steven L. Cornford, Alex Wheeler, Martin S. Feather, and Jeannette F. Plante. Assurance equations: A cost and criticality model for optimizing quality assurance surveillance. In *2022 IEEE Aerospace Conference (AERO)*, pages 1–13, 2022. doi:10.1109/AERO53065.2022.9843807.
- 6 Isal Dias, Ruwan Wickramarachchi, and Shan Jayasinghe. Key success factors for adoption of ci/cd with agile project management - systematic literature review. In *2025 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, pages 1–7, 2025. doi:10.1109/SCSE65633.2025.11031019.
- 7 Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 2008. Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008). doi:10.1016/j.entcs.2008.06.039.
- 8 Csaba Faragó, Péter Hegedűs, and Rudolf Ferenc. Code ownership: Impact on maintainability. In *Computational Science and Its Applications – ICCSA 2015*, pages 3–19, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-21413-9_1.
- 9 Matthieu Foucault, Cédric Teyton, David Lo, Xavier Blanc, and Jean-Rémy Falleri. On the usefulness of ownership metrics in open-source software projects. *Inf. Softw. Technol.*, 64(C):102–112, August 2015. doi:10.1016/j.infsof.2015.01.013.
- 10 Taher A. Ghaleb, Safwat Hassan, and Ying Zou. Studying the interplay between the durations and breakages of continuous integration builds. *IEEE Transactions on Software Engineering*, 49(4):2476–2497, 2023. doi:10.1109/TSE.2022.3222160.
- 11 Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Softw. Engg.*, 24(4):2102–2139, August 2019. doi:10.1007/s10664-019-09695-9.
- 12 Alfredo Imparato, Raffaele Rodolfo Maietta, Stefano Scala, and Vladimiro Vacca. A comparative study of static analysis tools for autosar automotive software components development. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 65–68, 2017. doi:10.1109/ISSREW.2017.21.
- 13 Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don’t software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. doi:10.1109/ICSE.2013.6606613.
- 14 Artem Kruglov, Giancarlo Succi, and Idel Ishbaev. Metrics representation and dashboards. In *Developing Sustainable and Energy-Efficient Software Systems*, pages 47–59. Springer International Publishing, Cham, 2023. doi:10.1007/978-3-031-11658-2_5.

- 15 Valentina Lenarduzzi, Fabiano Pecorelli, Nytyi Saarimaki, Savanna Lujan, and Fabio Palomba. A critical comparison on six static analysis tools: Detection, agreement, and precision. *J. Syst. Softw.*, 198(C), April 2023. doi:10.1016/j.jss.2022.111575.
- 16 Nivishree Palvannan and Chris Brown. Suggestion bot: Analyzing the impact of automated suggested changes on code reviews. In *2023 IEEE/ACM 5th International Workshop on Bots in Software Engineering (BotSE)*, pages 33–37, 2023. doi:10.1109/BotSE59190.2023.00015.
- 17 Kai Petersen, Cigdem Gencel, Negin Asghari, Dejan Baca, and Stefanie Betz. Action research as a model for industry-academia collaboration in the software engineering context. In *Proceedings of the 2014 International Workshop on Long-Term Industrial Collaboration on Software Engineering, WISE '14*, pages 55–62, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2647648.2647656.
- 18 Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. Monitoring continuous integration practices in industry: A case study. *arXiv preprint*, 2025. arXiv:2503.02610.
- 19 Jadson Santos, Daniel Alencar da Costa, Shane McIntosh, and Uirá Kulesza. On the need to monitor continuous integration practices. *Empirical software engineering*, 30(5):125, 2025. doi:10.1007/s10664-025-10682-6.
- 20 Todd Sedano, Paul Ralph, and Cécile Péraire. Practice and perception of team code ownership. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE '16*, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2915970.2916002.
- 21 Prameet Keshore Sinha, Parma Nand, Sanjay Singh Chauhan, and Smita Tiwari. Study of failures in continuous integration environment. In *2024 International Conference on Electrical Electronics and Computing Technologies (ICEECT)*, volume 1, pages 1–6, 2024. doi:10.1109/ICEECT61758.2024.10739077.
- 22 Mirosław Staron. *Action research in software engineering*. Springer Cham, 2020. doi:10.1007/978-3-030-32610-4.
- 23 Mirosław Staron, Wilhelm Meding, Jörgen Hansson, Christoffer Höglund, Kent Niesel, and Vilhelm Bergmann. Dashboards for continuous monitoring of quality for software product under development. In *Relating system quality and software architecture*, pages 209–229. Morgan Kaufmann, Boston, 2014. doi:10.1016/B978-0-12-417009-4.00008-9.
- 24 Mirosław Staron, Kent Niesel, and Wilhelm Meding. Selecting the right visualization of indicators and measures – dashboard selection model. In *International Workshop on Software Measurement*, pages 130–143, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-24285-9_9.
- 25 Darko Stefanović, Danilo Nikolić, Dušanka Dakić, Ivana Spasojević, and Sonja Ristić. Static code analysis tools: A systematic literature review. In *31st Daaam International Symposium on Intelligent Manufacturing and Automation*, pages 565–573, January 2020. doi:10.2507/31st.daaam.proceedings.078.
- 26 Simin Sun, David Friberg, and Mirosław Staron. “good” and “bad” failures in industrial ci/cd—balancing cost and quality assurance. In Davide Taibi and Darja Smite, editors, *Software Engineering and Advanced Applications*, pages 75–84, Cham, 2026. Springer Nature Switzerland. doi:10.1007/978-3-032-04207-1_6.
- 27 Anders Sundelin, Javier Gonzalez-Huerta, Richard Torkar, and Krzysztof Wnuk. Governing the commons: code ownership and code-clones in large-scale software development. *Empirical Software Engineering*, 30(2):43, 2025. doi:10.1007/s10664-024-10598-7.
- 28 Christoph Treude and Margaret-Anne Storey. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 1*, pages 365–374, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1806799.1806854.
- 29 Francesco Urdih, Theodoros Theodoropoulos, and Uwe Zdun. Architectural design decisions and best practices for fast and efficient ci/cd pipelines. In *Software Architecture*, pages 297–305, Cham, 2026. Springer Nature Switzerland. doi:10.1007/978-3-032-02138-0_19.

- 30 Carmine Vassallo. Enabling continuous improvement of a continuous integration process. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1246–1249, 2019. doi:10.1109/ASE.2019.00151.
- 31 Hironori Washizaki. Guide to the software engineering body of knowledge (swebok guide), 2024. URL: <https://www.computer.org/education/bodies-of-knowledge/software-engineering#about>.
- 32 Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. What to expect from code review bots on github? a survey with oss maintainers. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering, SBES '20*, pages 457–462, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3422392.3422459.