UNDERSTANDING SOFTWARE QUALITY ASSURANCE IN OPEN-SOURCE COMMUNITIES: AWARENESS, ADOPTION, AND TOOL UTILIZATION

ALI KHATAMI

Propositions

accompanying the dissertation

Understanding Software Quality Assurance in Open-Source Communities: Awareness, Adoption, and Tool Utilization

by

SeyedAli Khatami

1. Any research in software engineering that relies solely on the static analysis of configuration files to infer usage culture of tools, is methodologically flawed due to the "Configuration-Usage Gap" where a disparity exists between the presence of the configuration and its actual usage. (This Thesis)
2. The impact of tools and methods proposed through mining studies of open-source communities should be evaluated through contributions back to them.
3. The effectiveness of software quality assurance depends on teams executing practices with reflection on their contextual appropriateness.
4. Software engineering needs less focus on inventing new practices and more on systematic methods for analyzing what practices succeed in specific contexts.
5. In the LLM era, schools should put more focus on teaching students how to think critically, create meaningfully, and ask compelling questions rather than merely teaching facts.
6. Professional success depends more on relationships than expertise.
7. Attention is more valuable than time.
8. Global pop culture is gradually fading as a key organizer of collective identity, while algorithmically mediated niche communities are more and more shaping the structure of cultural belonging.
9. Morning routines are overrated. Evening routines are underrated. You can't fix at 7am what you broke at 11pm.
10. Work-life balance IS life balance.

These propositions are regarded as opposable and defendable, and have been approved as such by the promotor Prof. dr. A.E. Zaidman and the copromotor Dr. C.E. Brandt

Understanding Software Quality Assurance in Open-Source Communities: Awareness, Adoption, and Tool Utilization

Understanding Software Quality Assurance in Open-Source Communities: Awareness, Adoption, and Tool Utilization

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus,
Prof.dr.ir. H. Bijl,
chair of the Board for Doctorates
to be defended publicly on
Friday 23 October 2026 at 12:30

by

SeyedAli KHATAMI

This dissertation has been approved by the (co)promotors.

Composition of the doctoral committee:

Rector Magnificus,	chairperson
Prof. dr. A.E. Zaidman,	Delft University of Technology, promoter
Dr. C.E. Brandt,	Delft University of Technology, copromoter

Independent members:	
Prof. dr. ir. D. Spinellis,	Delft University of Technology
Prof. dr. M.T.J. Spaan,	Delft University of Technology
Prof. dr. T. Mens,	University of Mons, Belgium
Prof. dr. G. Robles,	Rey Juan Carlos University, Spain
Prof. dr. S. McIntosh,	University of Waterloo, Canada

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032).

Keywords: Software Quality Assurance, Open-Source Software, GitHub Actions, Mining Software Repositories

Printed by: ProefschriftMaken, https://www.proefschriftmaken.nl/

Cover by: Mehrab Nemati and Ali Khatami

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

ISBN: 978-94-6518-299-5

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

For my mother, the strongest and most loving person I know.
For my father, who gave me the stubbornness to start and the patience to finish.

We are read by what we reached, and written by what we could not.

Contents

Summary **xiii**

Samenvatting **xv**

Acknowledgments **xvii**

1 Introduction **1**
1.1 The Open Source Software Context . . . 2
1.2 Software Quality Assurance . . . 3
1.3 Research Goal . . . 7
1.4 Research Outline . . . 7
1.5 Research Methods . . . 9
1.6 Open Science and Research Ethics . . . 10
1.7 Contributions . . . 11
1.8 Thesis Structure . . . 11

2 State-Of-The-Practice in Quality Assurance in Java-Based Open Source Software Development **13**
2.1 Introduction . . . 13
2.2 Background . . . 16
2.2.1 (CI) Building, Compilability . . . 16
2.2.2 Code Review . . . 16
2.2.3 Testing . . . 17
2.2.4 Continuous Integration . . . 18
2.2.5 Other Elements of Quality Assurance . . . 19
2.3 Study Setup . . . 19
2.3.1 Studied Projects . . . 19
2.3.2 Data Collection. . . . 20
2.4 Results . . . 30
2.4.1 Local Builds . . . 30
2.4.2 Automatic Static Analysis Tools . . . 32
2.4.3 Continuous Integration . . . 32
2.4.4 Code Review . . . 36
2.4.5 Testing . . . 39
2.4.6 Quality Assurance Practices Put Together . . . 42
2.5 Discussion . . . 46
2.5.1 Revisiting the Research Questions . . . 46
2.5.2 Difficulties encountered during our investigation. . . . 48
2.5.3 The Bigger Context of Quality Assurance Practices. . . . 50

2.6 Related Work . . . 51
2.6.1 Quality Assurance Practices in Isolation . . . 51
2.6.2 Quality Assurance Practices in Conjunction . . . 53
2.7 Implications and Future Work . . . 54
2.8 Threats to Validity . . . 55
2.9 Conclusion . . . 56

3 Quality Assurance Awareness in Open Source Software Projects on GitHub 59
3.1 Introduction . . . 59
3.1.1 Information Needs in Software Engineering . . . 61
3.1.2 Quality Assurance Practices . . . 61
3.2 Study Methodology . . . 62
3.2.1 Survey Design . . . 62
3.2.2 Sampling . . . 62
3.2.3 Attracting participants . . . 62
3.2.4 Data Analysis . . . 63
3.3 Results . . . 64
3.3.1 RQ1) How aware are developers of testing in their projects? . . . 64
3.3.2 RQ2) How aware are developers of code reviews in their projects? . 69
3.3.3 RQ3) How aware are developers of automation workflows in their project, and what are the purposes for which they employ these workflows? . . . 71
3.3.4 RQ4) How aware are developers of continuous integration in their projects? . . . 74
3.3.5 RQ5) How aware are developers of guidelines in their project? . . . 77
3.3.6 Maintainers & Contributors Level of Awareness . . . 78
3.4 Limitations . . . 78
3.5 Conclusion, Implications, and Future Work . . . 79

4 Software Quality Assurance Analytics: Enabling Software Engineers to Reflect on QA Practices 81
4.1 Introduction . . . 81
4.2 RepoInsights: A Quality Assurance Analytics (QAA) Dashboard . . . 83
4.2.1 The Global Overview & Comparative Aspect . . . 83
4.2.2 Repository Dashboard Page . . . 83
4.2.3 Prototype Architecture Design . . . 84
4.3 Study Setup . . . 86
4.3.1 Study Design . . . 86
4.3.2 Study Execution . . . 87
4.3.3 Participants . . . 88
4.3.4 Data Analysis . . . 88
4.4 RQ1: How Do Software Engineers Apply QA Practices and How Do They Reflect on Them? . . . 89
4.4.1 Attitude Towards Quality Assurance and Software Analytics . . . 89
4.4.2 Well Justified Variety in Applied QA Practices . . . 90

4.4.3 Reflection on Applied QA Practices . . . 91
4.4.4 Motives to Extend QA Practices . . . 91
4.5 RQ2: What Are Software Engineers' Perspectives on a Quality Assurance Analytics Dashboard Like RepoInsights? . . . 93
4.5.1 Usage . . . 93
4.5.2 RQ2.1.) In What Ways Did the QAA Dashboard Help Software Engineers Reflect on QA Practices?. . . . 94
4.5.3 RQ2.2.) In What Ways Could the QAA Dashboard Be Improved to Help Engineers Reflect on QA Practices? . . . 96
4.5.4 RQ2. Summary Summarizing Software Engineers' Perspectives on RepoInsights . . . 99
4.6 Related Work. . . . 99
4.7 Threats to Validity . . . 100
4.8 Implications and Future Work . . . 100
4.9 Conclusion . . . 101

5 Catching Smells in the Act: A GitHub Actions Workflow Investigation 103
5.1 Introduction . . . 103
5.2 Background . . . 105
5.3 Study Set-Up . . . 106
5.4 RQ1: What are the common patterns of frequent changes in GHA workflow configurations?. . . . 106
5.5 RQ2: What types of smells exist in GHA workflow configurations? . . . 109
5.6 RQ3: Can we automatically detect GHA configuration smells? . . . 112
5.7 RQ4: To what extent do developers find the proposed fixes for identified GHA configuration smells relevant? . . . 113
5.8 Discussion . . . 120
5.8.1 Implications & Future Work . . . 121
5.8.2 Threats to Validity . . . 121
5.9 Conclusion . . . 122

6 Beyond the YAML File: Understanding Real-World GitHub Actions Workflow Adoption 123
6.1 Introduction . . . 123
6.2 Background . . . 125
6.2.1 Impact on Development Practices . . . 125
6.2.2 Ecosystem Structure and Evolution. . . . 125
6.2.3 Security and Vulnerability Management . . . 126
6.3 Methodology . . . 126
6.3.1 Data Collection. . . . 127
6.3.2 Categorization and Sampling Strategy . . . 127
6.3.3 Manual Analysis through Qualitative Coding . . . 129
6.3.4 Answering the research questions . . . 130

6.4 RQ1: What are the observed GitHub Actions utilization patterns? 132
6.4.1 Utilization Intensity Spectrum 132
6.4.2 Workflow Trigger Patterns 134
6.5 RQ2: How do developers react to GitHub Actions workflow failures? . . . 136
6.5.1 Fix-Oriented Response Patterns 136
6.6 RQ3: What relationships exist between project characteristics and GitHub Actions utilization patterns? 139
6.6.1 Project Characteristics and Workflow Utilization Patterns 139
6.6.2 Project Characteristics and Workflow Failure Reaction Patterns . . 141
6.7 Discussion 142
6.7.1 Collaborative vs. Direct Fixing in Workflow Failures 142
6.7.2 Local Optimization vs. Collective Costs in Deferred Fixing 143
6.7.3 The Configuration-Usage Gap 143
6.7.4 Opportunities for Enhanced GitHub Actions Platform 143
6.7.5 Threats to Validity 143
6.8 Conclusion and Future Work 144

7 Conclusion 145
7.1 Revisiting the Research Questions 145
7.2 Navigating the Complex Decision Space of SQA Design 148
7.2.1 Hypothesis Related to Diagnostic Information 149
7.2.2 Hypothesis Related to Awareness Information 150
7.2.3 Hypothesis Related to Behavioral Information 150
7.2.4 Hypothesis Related to Comparative Information 150
7.2.5 Hypothesis Related to Contextual Information 151
7.2.6 Hypothesis Related to Prescriptive Information 151
7.2.7 An Interconnected Framework 152
7.3 Implications 152
7.3.1 Implications: Software Practitioners 152
7.3.2 Implications: Tool Builders and Platform Providers. 153
7.3.3 Implications: Software Engineering Researchers 154
7.4 Future Research Directions. 155
7.4.1 Large-Scale Study of GitHub Actions Workflow Behavior 155
7.4.2 Investigating the Human and Cultural Dimensions. 156
7.4.3 AI-Augmented SQA Infrastructure 156
7.4.4 Quality Assurance for AI-Generated Code 156
7.4.5 Longitudinal Studies of SQA Evolution. 157
7.5 Concluding Remarks 157

Bibliography 159

Glossary 182

Curriculum Vitæ 183

List of Publications 185

Summary

Today's software development ecosystem offers an unprecedented array of quality assurance mechanisms. Continuous integration (CI) platforms, automated testing frameworks, automated static analysis tools, and code review systems are now readily available to developers, particularly in open-source software (OSS) projects hosted on platforms like GitHub. Yet this abundance does not automatically translate into effective adoption. Instead, it introduces a new challenge: the rapid expansion of Software Quality Assurance (SQA) practices and tools has created a complex decision space where teams must determine which practices to adopt, how to configure them effectively, how to integrate them into development workflows, and how to design processes that are relevant for their specific project context. This dissertation aims to construct a holistic understanding of software quality assurance by bringing together different aspects of open-source SQA practices that can inform future decision-making.

We begin by establishing an empirical understanding of how SQA practices are currently employed in the OSS world. Projects tend to adopt practices unevenly across dimensions, while broader investment across SQA practices is more common among mature projects. This raises a deeper question about whether developers actually understand the mechanisms already present in their projects. Our survey of open-source maintainers and contributors confirms this concern, uncovering an awareness gap: while developers broadly know that practices such as CI and testing are in place, their awareness is more limited when asked about specific mechanisms and tools.

To address the awareness gap, we design and evaluate RepoInsights, a software quality assurance analytics dashboard consolidating data from testing, code review, and automated workflow activities. While participants value the overview, they express a need for richer and more fine-grained contextual information. This motivates a deeper investigation into GitHub Actions (GHA), a platform deeply integrated into the GitHub ecosystem and a promising source of such contextual depth. We investigate GitHub Actions through two complementary lenses, revealing two further findings: workflow configurations suffer from recurring quality issues that require active maintenance, and configured workflows are not necessarily used in practice, as configuration does not imply actual utilization.

Taken together, our five studies contribute toward a deeper understanding of how software quality assurance operates in open-source software development on GitHub. The empirical findings and conceptual contributions of this dissertation, including the awareness gap, the configuration–usage gap, and the characterization of developer responses to GHA workflow failures, contribute to the holistic understanding needed to navigate the complex SQA decision space in open-source software development.

Samenvatting

Het hedendaagse softwareontwikkelingsecosysteem biedt een ongekend scala aan mechanismen voor softwarekwaliteitsborging (Software Quality Assurance, SQA). Platforms voor continuous integration (CI), geautomatiseerde testframeworks, geautomatiseerde statische analysetools en codereviewsystemen zijn tegenwoordig breed beschikbaar voor ontwikkelaars, met name in opensourcesoftwareprojecten (OSS) op platforms zoals GitHub. Toch vertaalt deze overvloed zich niet automatisch in effectieve adoptie. De snelle uitbreiding van SQA-praktijken en -tools brengt een nieuwe uitdaging met zich mee: een complex beslissingslandschap waarin teams moeten bepalen welke praktijken zij adopteren, hoe zij deze effectief configureren, hoe zij deze integreren in ontwikkelworkflows en hoe zij processen ontwerpen die relevant zijn voor hun specifieke projectcontext. Dit proefschrift beoogt een holistisch begrip van software quality assurance op te bouwen door verschillende aspecten van open-source SQA-praktijken samen te brengen die toekomstige besluitvorming kunnen ondersteunen.

We beginnen met het opbouwen van een empirisch begrip van hoe SQA-praktijken momenteel worden toegepast in de open-source wereld. Projecten hebben de neiging SQA-praktijken in verschillende dimensies in uiteenlopende mate te adopteren, terwijl bredere investeringen in SQA-praktijken vaker voorkomen bij volwassen projecten. Dit roept de vraag op of ontwikkelaars de mechanismen die al in hun projecten aanwezig zijn daadwerkelijk begrijpen. Onze enquête onder beheerders van en bijdragers aan opensourcesoftwareprojecten bevestigt deze zorg en brengt een bewustzijnskloof aan het licht: hoewel ontwikkelaars in grote lijnen weten dat praktijken zoals CI en testen worden toegepast, blijkt hun bewustzijn beperkter wanneer naar specifieke mechanismen en tools wordt gevraagd.

Om deze bewustzijnskloof aan te pakken, ontwerpen en evalueren we RepoInsights, een software quality assurance-analysedashboard dat gegevens uit testen, code review en geautomatiseerde workflowactiviteiten samenvoegt. Hoewel deelnemers het overzicht waarderen, geven zij aan behoefte te hebben aan rijkere en meer gedetailleerde contextuele informatie. Dit motiveert een dieper onderzoek naar GitHub Actions (GHA), een platform dat diep geïntegreerd is in het GitHub-ecosysteem en een veelbelovende bron is voor dergelijke contextuele diepgang. We onderzoeken GitHub Actions vanuit twee complementaire perspectieven, wat twee verdere bevindingen oplevert: workflowconfiguraties kampen met terugkerende kwaliteitsproblemen die actief onderhoud vereisen, en geconfigureerde workflows worden niet noodzakelijkerwijs in de praktijk gebruikt, omdat configuratie niet automatisch daadwerkelijk gebruik betekent.

Samen dragen onze vijf studies bij aan een dieper begrip van hoe softwarekwaliteitsborging functioneert binnen opensourcesoftwareontwikkeling op GitHub. De empirische bevindingen en conceptuele bijdragen van dit proefschrift, waaronder de bewustzijnskloof, de kloof tussen configuratie en gebruik en de karakterisering van de reacties van ontwikkelaars op falende GHA-workflows, dragen bij aan het holistische begrip dat nodig is

om binnen het complexe SQA-beslissingslandschap van opensourcesoftwareontwikkeling weloverwogen keuzes te maken.

Acknowledgments

A PhD is rarely a straight path. Mine had its ups and downs, and it was shaped by everyone I encountered along the way, whether they chose to be part of it or simply happened to be there. Some of those encounters lasted years, others a single conversation, and I am not sure I could say which mattered more. I could never name everyone who played a part, but I am grateful to all of you for helping me reach this milestone. Here, I would like to thank some of you by name.

Andy, first and foremost: beyond being my PhD supervisor, you have been one of the most influential people in my life. I have enjoyed every one of our meetings, from the very first ones, when I was still thousands of kilometers away, until today. As a supervisor and mentor, you guided me toward becoming an independent researcher, which is exactly what I set out to become when I started this PhD, and I am deeply grateful for that. I learned so much from you, about research and far beyond it. This thesis would not have been possible without you. Thank you.

Caro, you were one of the first people I met after moving to the Netherlands, and you helped me find my way from the very start. You were always great company, and the many projects we worked on together were both fun and insightful. As my co-supervisor, you were a great support as this thesis came together. Whenever I needed help, you were there, and I truly appreciate it.

I would like to thank the members of my doctoral committee, Diomidis Spinellis, Tom Mens, Gregorio Robles, Shane McIntosh, and Matthijs Spaan, for carefully reading my manuscript and for their constructive feedback. Diomidis, although you were not often in Delft, you always felt present and approachable. I enjoyed our conversations, and your work has given me a great deal to think about over the years. Tom, your comments were detailed and generous, and this thesis is noticeably better because of them. Gregorio, Shane, and Matthijs, it is an honour to have you on my committee.

Caro, Mark, Amir, and Baris, you were wonderful officemates. I am grateful for our chats, coffees, and lunches together, as well as for the many research discussions, the feedback we exchanged, and the work we did side by side. You made the office a place I was always glad to come to.

I would like to thank everyone in SERG for creating such a cheerful and supportive environment. Conference trips and shared lunches turned several colleagues into close friends, which is perhaps the part I will miss most. Arie, I enjoyed our runs during the SERG retreats and the conversations that came with them. Annibale and Mitchell, our chats during breaks were always a highlight. Maliheh, Sebastian, Burcu, Luis, Thomas, June, and Mauricio, thank you for the conversations we had and the moments we shared along the way. Pouria, our time at SERG did not overlap, but you helped me a great deal in my early days, and I have not forgotten it.

To the PhD candidates, postdocs, and master's students who shared my time at SERG, thank you for making these years so enjoyable: Xutong, Ali Al-Kaswan, Ali Asgari, Enrique,

Antony, Ege, Daniele, Shujun, Eileen, Jonathan, Ziyou, João, Zahra, Sara, Aral, Aru, Quentin, Lorena, Amir Mir, Imara, Leonard, Fabio, Mahdiyeh, Mehdi, and Marielli. Coffee breaks, lunch conversations, and hangouts outside the office are what made the everyday side of a PhD something to look forward to, and what turned a research group into a community. Cedric, supervising you was a pleasure, and I am glad we got to work on such an exciting project together.

I am also grateful to Minaksie and Kim for the joy and positivity you brought to our office, and for always being there to help us navigate university matters.

To my colleagues at DDS and BK: working with you while finishing my PhD has been a wonderful experience, and the welcoming environment you created made a real difference in bringing this work to completion. Seyran, thank you for welcoming me to DDS and for your support through the final stages. It has been a pleasure working with you, and I feel fortunate to be part of the Data Refinery Lab. Georg, you are an example of what great leadership in academia looks like, and I am glad to be part of the team you lead. Marija, your positive energy and the way you approach work make every collaboration a pleasure. Casper and Fatemeh, I enjoy the projects we are doing together and look forward to many more. Agostino, Angela, Jess, Linda, Michael, Daria, Ömer, and Yagiz, the work culture you have created is something special, and I am glad to have landed among you.

Outside of work, a few people made a big difference during these years, each in their own way. Mark, you have been a good friend, and I learned a lot from you, both about work and from our long talks. Milad, we went from gym buddies to real friends, and you are someone I know I can always rely on. Amir (Deljouyi), we went from master's classmates to the kind of friends who understand each other without needing many words, with plenty of good memories in and out of the office. Moein, your advice has a way of putting things in perspective, and you have been generous with it whenever I needed it. Amir (Ghadiri), there were days, and sometimes weeks, when you were simply there, and that helped me more than you probably realize.

To my lovely friends: Mehrab, Masiha, Patricia, Francis, Mammad (Miro), Faezeh, Saba, Behrad, Shayan, Sanaz, Anita, Roham, Zahra, Ashkan, Aral, Ali Asgari, Ali Kalantari, Ali Nesaei, Julius, Ali Elahinik, Yalda, Parmida, Shirin, Sadegh, Saeed, Amin, Gisou, Mahdi, Kim, Katy, Yaser, Pouria, Sadaf, Khashayar, Sara, Nima, Bahareh, Hafez, Amirhossein, Dorsa, Alessandro, Francesca, Fardad, Entela, and Hamed. We have been through so much together, from conversations that stayed with me and evenings that ran long to building communities, traveling, and countless other moments I will always treasure. Whatever part of this you were there for, thank you for it.

To my friends back in Iran, AmirReza, Milad, Mehdi, Samad, Hamed, Farhad, Mohammad, AliReza, Afshin, Nemat, and Mohsen: you may not realize it, but our chats, however short, helped me stay connected through the hard times of these years away from home.

Maman, Baba, and Abji, you have supported me through every chapter of my life, and this PhD was no exception. Without the wonderful family we have, I would never be standing where I am today.

Ali
Delft, September 2026

1

Introduction

Software quality assurance (SQA) practices are the fundamental mechanisms through which software engineers ensure their software meets the specified requirements and maintains reliability. Today's software development ecosystem offers an unprecedented array of quality assurance mechanisms. Continuous integration platforms, automated testing frameworks, static analysis tools, and code review systems are now readily available to developers, particularly in open-source software (OSS) projects hosted on platforms like GitHub.

Yet this abundance does not automatically translate into the effective adoption of every SQA practice or tool. Instead, it introduces a new challenge: ***the proliferation of SQA practices and tools has created a complex decision space*** *where teams must determine which practices to adopt, how to configure them effectively, how to integrate them into development workflows, and how to design processes that are relevant and beneficial for their specific project context.*

To support these decisions, research needs to explain how SQA practices are adopted together, whether developers are aware of the mechanisms implemented in their projects, and how configured tools are used in practice. ***Existing knowledge remains fragmented across these dimensions****, limiting our understanding of how practice adoption, developer awareness, and tool utilization relate to one another in specific project contexts.*

This dissertation aims to construct a holistic understanding of software quality assurance by bringing together different aspects of open-source SQA practices that can inform future decision-making.

Imagine a software team facing a dilemma: their project suffers from quality issues, or they are starting fresh and must design quality assurance from scratch. Before them lies a daunting array of choices: software testing frameworks, code review processes, continuous integration platforms, and automated static analyzers, each with costs, benefits, and complex interactions. **How should they navigate this decision space?**

A further, very recent development sharpens this problem: autonomous coding agents increase the volume and pace of code production, while entering the decision space as one more tool to adopt, configure, and integrate. Early evidence points to quality implications: agent-authored contributions are associated with more post-merge churn and lower code

survival than human-authored changes [1], and verification is applied unevenly, with some developers merging agentic pull requests without running any continuous integration checks [2]. As this shift emerged after the studies in this dissertation were conducted, we do not examine it here, but flag it as an important factor for future work.

The core problem: While modern software development offers an unprecedented array of quality assurance mechanisms, this abundance has created a paradox. Teams have more options for assuring quality, yet limited guidance on how to select, configure, and integrate them in their specific contexts. Research on individual practices and their relationships provides valuable evidence, but an integrated understanding of adoption, developer awareness, and actual tool utilization remains limited. Which practices are adopted together? To what extent are developers aware of the mechanisms implemented in their projects? How are these mechanisms configured, and how does that configuration relate to their execution in practice?

This dissertation connects these dimensions through empirical studies of SQA practice adoption, developer awareness, analytics, and automation configuration and execution on GitHub.

1.1 The Open Source Software Context

The open-source software ecosystem, particularly projects hosted on GitHub, provides an ideal research setting to build this understanding. GitHub has emerged as the dominant platform for collaborative software development, hosting millions of projects with rich historical records of development activities, team interactions, and SQA configurations.

Why GitHub specifically? Three characteristics make GitHub well suited for investigating SQA practices at scale:

1. **Scale and Diversity:** With 420+ million repositories and 150+ million users [3], GitHub captures diverse adoption patterns across projects of varying sizes, domains, and team compositions. This scale enables comparisons across many projects, although the generalizability of the findings depends on the sampling criteria.

2. **Transparency and Observability:** Public repositories expose development histories, pull request discussions, code review comments, and, where available, CI/CD execution records. These artifacts make it possible to investigate aspects of SQA practice that are often inaccessible in closed-source settings.

3. **Integrated Tooling Ecosystem:** Multiple SQA practices operate within a unified platform: pull requests for code review, continuous integration for automated testing, and integrations for static analysis. This integration allows us to study how practices interact rather than examining isolated implementations.

Moreover, GitHub Actions has created observable integrated SQA automation where execution logs (workflow run records), failure traces, and behavioral patterns are all accessible within the platform itself. This makes systematic empirical investigation of integrated SQA practices possible and practical.

This focus on GitHub also carries limitations. GitHub does not represent all open-source development; projects hosted on platforms such as GitLab or Bitbucket[1] fall outside the scope of this dissertation. Public open-source practice may also differ from practice in closed-source industrial settings. Our observations are limited to publicly available activity, and sampling popular or active repositories can over-represent well-maintained projects [4]. We therefore treat GitHub as a rich but bounded research setting and discuss generalizability in each study's threats to validity.

1.2 Software Quality Assurance

Having established the problem (complex decision space) and the research setting (GitHub), we now detail the specific SQA practices that constitute this decision space. As software pervades modern society [5] and controls critical aspects from economic, safety, security and scientific standpoints, its quality and dependability have become indispensable [6], given the potential of catastrophic consequences [7]. To ensure quality, software engineers employ a range of quality assurance practices, including but not limited to: software testing [8–14], modern code review [15–17], automated static analysis [18, 19], build and CI/CD automation [20–24].

Individual quality assurance practices have received substantial research attention: researchers have analyzed the prevalence of automated static analysis tools in open source software development [18], have performed an empirical analysis of build failures [25], have investigated aspects of code reviews [26, 27], and have studied continuous integration to understand its adoption and usage [24]. Other studies have examined relationships between QA practices, e.g., the impact of continuous integration on code review [28], the impact of continuous integration on other software development practices [29], the interplay between pull requests and CI [30], the complementarity of static analysis tools and code reviews [31], and the relationship between CI and test code evolution [32]. Some of these QA approaches are to be considered complementary to each other, e.g., code reviews and software tests find different types of defects [33].

Understanding how these practices are adopted in combination on GitHub forms the empirical foundation of this dissertation.

SQA Practices: Software Testing

Software testing serves as a fundamental quality assurance practice by detecting faults before they reach end users [8, 34–38]. Tests can be categorized by their scope and granularity, e.g., unit tests verify the correctness of individual functions or methods in isolation; integration tests examine the interactions between multiple components or modules; and end-to-end tests validate complete user scenarios by simulating real user interactions with the software, verifying that entire workflows function as expected from the user's perspective. Each test type targets different aspects of software quality and typically operates at different levels of abstraction. Test coverage measures the extent to which a test suite exercises selected structural elements of the code. Line coverage measures the percentage of executable source lines exercised, statement coverage measures the percentage of executable statements exercised, and branch coverage measures the percentage of branches

[1] GitLab (`https://gitlab.com`) and Bitbucket (`https://bitbucket.org`) are alternative Git-based code-hosting platforms.

(e.g., if-else conditions) taken. Higher coverage indicates that more of the measured code structure has been exercised, but does not establish the strength of test assertions or guarantee the absence of bugs [39, 40]. Test coverage serves as a widely recognized signal of project maturity and quality [41, 42]. On GitHub, many projects display coverage badges in their repositories. To maintain or improve coverage levels, projects may enforce minimum coverage thresholds for new contributions, using automated checks integrated into their CI/CD pipelines to evaluate pull requests before they can be merged [43].

SQA Practices: Code Review

Code Review is the systematic examination of source code by developers other than the original author or contributor. In modern software development, particularly in open-source projects hosted on GitHub, code review has evolved from formal, heavyweight inspections to a lightweight, tool-supported practice known as Modern Code Review (MCR) [26, 44]. Code review serves multiple purposes beyond defect detection: it facilitates knowledge transfer among team members, increases collective code ownership, ensures adherence to coding standards, and provides opportunities for mentoring less experienced developers [15]. In the pull-based development model common on GitHub, contributors submit changes as pull requests (PRs), which project maintainers and other developers review before merging into the main codebase [45]. Code review quality can be characterized by several factors including thoroughness of feedback, review participation, reviewer expertise and familiarity with the code under review, and the timeliness of review responses [27, 46].

SQA Practices: Continuous Integration & Deployment

Continuous Integration (CI) is a development practice where developers integrate code changes frequently into a shared repository. Each integration is automatically verified by building the project and running automated tests, allowing teams to detect integration problems early. Continuous Deployment (CD) extends CI by automatically deploying software that passes the CI pipeline to production environments. Together, CI/CD practices aim to reduce integration risks, improve software quality through frequent testing, and accelerate the release cycle [29, 47].

In the context of open-source software on GitHub, CI/CD is typically implemented through platforms that automatically trigger workflows in response to repository events such as commits, pull requests, or releases. The outcomes of these automated CI/CD workflows are visible to maintainers and contributors on pull requests, providing immediate feedback on proposed changes that informs the code review process and helps identify integration issues before merging [28, 30, 48].

Among the various CI/CD platforms available (e.g., Travis CI, Jenkins, CircleCI, etc.) GitHub Actions has emerged as particularly significant for our investigation. While we discuss it briefly here, its unique role in the SQA ecosystem warrants deeper examination, which we address later in this section.

SQA Practices: Automated Static Analysis

Automated Static Analysis Tools (ASATs) examine source code without executing it, identifying potential bugs, code smells [49], style violations, and security vulnerabilities through pattern matching and abstract interpretation. Unlike testing, which examines runtime behavior for specific inputs, static analysis examines code without executing it, with the scope

and precision of its analysis depending on the tool and technique [18, 31]. In OSS projects on GitHub, ASATs are commonly integrated into CI/CD pipelines to automatically check code quality for every change. However, the effectiveness of ASATs depends not only on their detection capabilities but also on how developers respond to their warnings: whether they address issues, suppress false positives, or ignore warnings entirely [18, 19, 50]. When integrated into automation workflows, ASAT results become visible alongside other quality signals on GitHub events such as commits, pull requests, and releases.

SQA Awareness

While the previous sections describe specific SQA mechanisms, the effective utilization of these tools relies in part on a fundamental cognitive prerequisite: awareness. Drawing on the concept of *situational awareness* in psychology [51], we define Software Quality Assurance Awareness in this dissertation as the knowledge software engineers possess regarding the specific QA practices active in their projects. This definition focuses on awareness of the practices themselves; broader situational awareness also involves understanding how these practices function and interpreting their signals.

This awareness cuts across testing, code review, ASATs, and CI/CD. Having a tool configured in a repository does not guarantee that developers know it is present, how it functions, when it executes, or how to interpret its signals. The distinction between knowing that a practice exists and understanding how it is functioning is important for navigating the complex decision space of designing QA systems. We therefore investigate developers' awareness of the mechanisms employed in their projects and explore how SQA analytics can support deeper situational awareness.

SQA Analytics

To bridge the gap between the existence of SQA practices and the practitioner's awareness of them, we look toward Software Quality Assurance Analytics (SQAA). Software analytics involves enabling software practitioners to explore data to obtain insightful and actionable information for data-driven decision-making [52]. SQAA applies this concept specifically to SQA data, aggregating information from sources such as testing frameworks, code review discussions, and automation logs into a holistic view.

The motivation for SQAA is to serve as a mechanism for reflection. By visualizing the status of testing, workflows, and guidelines in a unified dashboard, analytics tools aim to enhance situational awareness. This dissertation explores the design of such analytics to determine if providing a global overview and comparative metrics can empower software engineers to assess their project's SQA and make informed decisions about its SQA practices.

This awareness challenge becomes even more complex when considering automation platforms that orchestrate multiple SQA practices simultaneously, a topic we explore in depth below.

GitHub Actions: A Platform for Integrated SQA

Having described individual SQA practices and the cognitive challenges of awareness, we now turn to the platform that has come to orchestrate these practices: GitHub Actions (GHA).

Why GitHub Actions deserves special attention: While all the previously discussed practices are important, GHA occupies a unique position in the modern SQA ecosystem that makes it particularly valuable for investigation:

1. **Platform Integration Hub:** Unlike standalone CI/CD tools that operate externally, GHA is deeply embedded within the GitHub platform itself. This native integration means workflows can directly interact with GitHub's social coding features: automatically responding to pull request reviews, managing issues, controlling merge permissions, and triggering on GitHub-specific events. This makes GHA not just a build orchestration tool, but a platform that unifies technical automation (testing, deployment) with social processes (code review, collaboration) within a single observable ecosystem.

2. **Observability:** While other CI/CD platforms often require separate accounts or hide execution details behind external dashboards, GHA provides rich execution logs, failure traces, and behavioral patterns openly accessible within the same interface where developers already collaborate. This co-location of automation artifacts with source code and social interactions enables unprecedented empirical investigation.

3. **Adoption Dominance:** GHA has rapidly become the dominant automation platform on GitHub [53], making it an important setting for studying how open-source teams implement SQA automation.

4. **Dual Investigation Opportunity:** GHA can be studied from two complementary angles: the static quality of workflow configurations (the YAML files) and the dynamic behavior of workflow executions (the runtime logs).

The evolution of SQA on GitHub has culminated in GHA's widespread adoption, allowing developers to automate not just builds, but social interactions, security checks, and repository management [53, 54]. However, this rapid adoption introduces new challenges:

First, the static configuration challenge. GHA relies on YAML-based configuration files which can become complex and prone to decay. Just as source code suffers from code smells, automation configurations can suffer from *workflow smells* [55, 56]: suboptimal patterns that negatively affect maintainability, security, or performance. Understanding what constitutes a "good" GHA configuration requires empirical investigation of real-world practices (Chapter 5).

Second, the dynamic runtime challenge. The presence of a configuration file does not guarantee active or effective utilization [18]. To fully understand the impact of automation, we must look beyond the YAML file to the execution logs, analyzing how these workflows behave in reality, how frequently they fail, and how developers respond when they do (Chapter 6).

By investigating GHA through the complementary perspectives of configuration quality and runtime behavior, we contribute to a holistic understanding of how automation platforms support SQA practices in specific project contexts. Moreover, because GHA consolidates multiple SQA practices within a single observable platform, studying it provides the rich contextual depth that emerged as essential in our analytics work (Chapter 4).

1.3 Research Goal

The preceding sections reveal a clear gap: despite the proliferation of SQA practices and tools on GitHub, we lack systematic knowledge of how these practices are adopted in combination, whether developers are aware of the mechanisms active in their projects, and how automation platforms behave at runtime. Therefore, the overarching goal of this dissertation is:

Research Goal

To understand the complex software quality assurance (SQA) decision space by establishing a holistic understanding of the SQA ecosystem on GitHub.

To pursue this goal, we examine SQA practices as an interconnected system of practices, human awareness, tools, and automation. Across the studies, we connect evidence on practice adoption, SQA awareness, and tool configuration and utilization to build this holistic understanding.

1.4 Research Outline

This dissertation addresses the overarching goal through five complementary studies, progressing from a broad empirical analysis of SQA practice adoption to investigations of developer awareness, SQA analytics, and GitHub Actions configuration and runtime behavior.

Research Questions

We organize the dissertation around four overarching research questions, RQ I–RQ IV.[2]

RQ I (Chapter 2)

What is the current state of practice regarding SQA adoption in OSS projects, and how are different SQA practices used in conjunction?

We begin by establishing an understanding of how SQA practices are currently employed in the OSS world. We extend research on individual practices and their relationships by examining adoption across several SQA practices within the same set of projects. In this study, we analyze 1,454 popular Java projects on GitHub to investigate the co-usage of software testing, modern code review, automated static analysis (ASATs), and build automation. We examine whether projects that invest heavily in one practice also invest in others.

[2] We label these thesis-level questions with Roman numerals (RQ I–RQ IV) to keep them distinct from the chapter-level research questions (RQ1, RQ2, …): each chapter is a self-contained study that numbers its own research questions independently.

1

RQ II (Chapter 3)

To what extent are OSS developers aware of the SQA practices and mechanisms implemented in their projects?

In this chapter we investigate the human cognitive dimension: to what extent are developers aware of SQA practices in general, and do they know which SQA mechanisms and tools are being adopted within their own repositories? In this study we survey 471 open-source maintainers and contributors. We find that while high-level awareness is common, deep situational awareness is lacking. This "awareness gap" motivates the subsequent studies in this thesis.

RQ III (Chapter 4)

How can software analytics tools enhance developers' understanding of and reflection on SQA practices?

To address the "awareness gap" identified in Chapter 3, we propose and evaluate a concrete intervention: Software Quality Assurance Analytics (SQAA). We design RepoInsights, a dashboard that aggregates data from testing, code review, and automated workflow activities into a holistic view to enable developer reflection. Through interviews with 14 software engineers interacting with a RepoInsights prototype, we assess the viability of this approach. While the dashboard successfully provides a comprehensive overview with comparative metrics, participants expressed a need for more detailed views with deeper contextual information to derive actionable insights. This need for contextual information connects the analytics study to the investigations of workflow configuration and execution in Chapters 5 and 6.

RQ IV (Chapters 5 & 6)

How is GitHub Actions utilized in practice for software quality assurance?

As discussed in Section 1.2, GitHub Actions orchestrates multiple SQA practices within a single platform. By examining both workflow configuration quality and runtime behavior, we investigate how configured SQA mechanisms translate into actual tool utilization in specific project contexts, contributing to our holistic understanding of the SQA ecosystem.

RQ IV.a (Chapter 5)

What quality issues emerge in GitHub Actions workflow configurations that may impact SQA effectiveness?

In this study, we investigate the quality of automation definitions themselves. Just as production code can suffer from "code smells," GHA workflows that underpin SQA automation can exhibit "workflow smells": suboptimal patterns that harm maintainability, reliability, or security. By analyzing the change history of GHA configuration files, we identify potential

workflow smells and validate their practical significance through a contribution study where we propose fixes to active GitHub repositories.

RQ IV.b (Chapter 6)

How do GitHub Actions workflows behave in practice, and how do development teams respond to workflow failures?

Having examined static workflow configurations in Chapter 5, we recognize that the mere presence of a GHA YAML file does not guarantee effective usage. Therefore, this final study moves beyond static analysis to examine dynamic runtime behavior. We analyze workflow execution records from 952 repositories, conducting qualitative analysis of 21 projects to understand how GHA automation behaves in practice: how teams respond to workflow failures, what factors influence these responses, and how these patterns relate to project contexts.

Outline Logic

The research progresses through four interconnected stages, connecting practice adoption, developer awareness, analytics, and automation:

We begin by establishing the **empirical landscape** (RQ I), examining how different SQA practices are adopted and combined in open-source projects. This reveals the state of the practice in adoption of SQA practices but raises questions about whether developers are fully aware of and effectively utilizing these practices.

This motivates our investigation of the **human dimension** (RQ II), where we examine developers' awareness of SQA mechanisms in their repositories. The discovery of an "awareness gap" between high-level knowledge and deep situational understanding reveals a critical challenge in making SQA practices effective.

To address this gap, we propose a **tooling intervention** (RQ III) through a software quality assurance analytics (SQAA) dashboard coined RepoInsights. While this demonstrates the potential of aggregated views, it also reveals the need for richer contextual information, pointing us toward the automation platform itself as a source of deeper insights.

This leads to the deep investigation of **GitHub Actions** (RQ IV), as the automation platform underlying modern SQA practices. By examining both the static quality of workflow configurations (RQ IV.a) and their dynamic behavior (RQ IV.b), we examine how workflow configuration and execution shape SQA in the studied projects. GitHub Actions is particularly valuable for this investigation because it consolidates multiple SQA practices within a single, observable platform, providing the rich contextual depth that emerged as essential in RQ III.

1.5 Research Methods

This dissertation employs a mixed-methods approach combining large-scale repository mining, questionnaire and qualitative surveys, engineering research, and case studies. We characterize each study using the terminology of the ACM SIGSOFT Empirical Standards [57] for empirical software engineering research.

Chapter 2 mines 1,454 Java projects using an automated pipeline that reproduces local builds via Docker, mines static analysis configurations, and injects JaCoCo to measure test coverage. Chapter 3 surveys 471 open-source maintainers and contributors, applying open card sorting to assess their awareness of active SQA mechanisms. Chapter 4 takes a design science approach to build RepoInsights, evaluated through semi-structured interviews and think-aloud protocols with 14 software engineers in a one-group pretest-posttest study. Chapter 5 mines 10,012 commits across 83 projects bottom-up to identify workflow smell patterns, validated by 32 pull requests to active repositories. Chapter 6 analyzes 258,300 workflow run records quantitatively and qualitatively examines developer reactions to failures across 21 repositories.

Research Methods by Chapter

- **Chapter 2:** Repository Mining, 1,454 projects, automated build reproduction, coverage instrumentation
- **Chapter 3:** Questionnaire Survey, 471 participants, descriptive statistics, open card sorting
- **Chapter 4:** Engineering Research (design science), prototype + user study with 14 participants (interviews, think-aloud, questionnaire)
- **Chapter 5:** Repository Mining + contribution study, 10,012 commits mined and analyzed, 32 pull requests analyzed
- **Chapter 6:** Mixed Methods (Repository Mining + Case Study), 258,300 workflow runs (quantitative) + 21 repositories (qualitative)

1.6 Open Science and Research Ethics

Table 1.1: Publications and replication packages for dissertation chapters

Ch.	OA	Study Title (shortened)	Publication DOI	Replication Package
2	Gold	State-Of-The-Practice in SQA	http://doi.org/10.1002/spe.3321	https://doi.org/10.5281/zenodo.7404903
3	Green	SQA Awareness	https://doi.org/10.1109/SCAM59687.2023.00027	https://doi.org/10.5281/zenodo.8139381
4	Green	SQAA: RepoInsights	https://doi.org/10.1109/SCAM63643.2024.00027	https://doi.org/10.5281/zenodo.10961021
5	Green	GHA Workflow Smells	https://doi.org/10.1109/SCAM63643.2024.00015	https://doi.org/10.5281/zenodo.12207164
6	Green	GHA Adoption	https://doi.org/10.1145/3816483.3816499	https://doi.org/10.5281/zenodo.18258226

To ensure rigour, transparency, and reproducibility, this research adheres to open science principles in accordance with the Dutch Research Council (NWO) requirements, which funded this work through the Vici “TestShift” grant. For each study, comprehensive replication packages are publicly available on Zenodo, containing data collection pipelines,

software artifacts (e.g., RepoInsights prototype, smell detection scripts), datasets (1,454 Java projects, 456 anonymized survey responses), and analysis scripts. All peer-reviewed papers forming Chapters 2–6 are available under green (Chapters 3–6) or gold (Chapter 2) open access; Table 1.1 lists the respective DOIs and replication package links.

Beyond protecting participants, we view responsible engagement as an ethical obligation in research that mines open-source communities: rather than merely extracting data from these projects, the contribution study gave back by proposing concrete fixes to the communities we studied (Chapter 5).

1.7 Contributions

The research in this dissertation brings the following contributions:

1. Quantitative characterization of SQA practice adoption intensity across testing, code review, automated static analysis, and continuous integration in 1,454 open-source Java projects, establishing empirical baselines for combined practice utilization (Chapter 2).
2. Curated dataset of 312 Java repositories that were successfully built including their generated test coverage details (Chapter 2).
3. Empirical characterization of awareness gaps in SQA through a survey of 471 open-source maintainers and contributors (Chapter 3).
4. The conceptual design and functional prototype of RepoInsights, a Software Quality Assurance Analytics (SQAA) dashboard designed to aggregate testing, workflow, and review data into a holistic view (Chapter 4).
5. A set of qualitative design recommendations for future SQA analytics tools, derived from user studies, focusing on the need for data granularity and deep automation context to facilitate practitioner reflection (Chapter 4).
6. Identification of seven GHA workflow smells concerning maintainability, security, and resource usage, together with evidence of their practical relevance from a contribution study proposing fixes to active open-source projects (Chapter 5).
7. Empirical characterization of the configuration–usage gap in GHA workflows, showing that the presence of workflow files alone is insufficient to establish how automation is used in practice (Chapter 6).
8. An empirical characterization of developer responses to GHA workflow run failures (categorized as Immediate Fixing, Deferred Fixing, and Ignore/Abandon), derived from analysis of pull requests and commit history (Chapter 6).

1.8 Thesis Structure

This dissertation follows a portfolio-based structure, where Chapters 2 through 6 are all published research papers. Each chapter is self-contained and can be read independently. Chapter 7 synthesizes the findings, connects them to the overarching research goal, discusses implications for research and practice, and outlines future research directions.

The publications forming the basis for Chapters 2–6 are:

- **Chapter 2:** Ali Khatami and Andy Zaidman. "State-of-the-Practice in Quality Assurance in Java-Based Open Source Software Development." *Software: Practice and Experience*, 2024.
- **Chapter 3:** Ali Khatami and Andy Zaidman. "Quality Assurance Awareness in Open Source Software Projects on GitHub." In *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2023.
- **Chapter 4:** Ali Khatami, Carolin Brandt, and Andy Zaidman. "Software Quality Assurance Analytics: Enabling Software Engineers to Reflect on QA Practices." In *Proceedings of the 24th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2024.
- **Chapter 5:** Ali Khatami, Cédric Willekens, and Andy Zaidman. "Catching Smells in the Act: A GitHub Actions Workflow Investigation." In *Proceedings of the 24th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2024.
- **Chapter 6:** Ali Khatami, Carolin Brandt, and Andy Zaidman. "Beyond the YAML File: Understanding Real-World GitHub Actions Workflow Adoption." In *Proceedings of the 30th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2026.

2

State-Of-The-Practice in Quality Assurance in Java-Based Open Source Software Development

To ensure the quality of software systems, software engineers can make use of a variety of quality assurance approaches, e.g., software testing, modern code review, automated static analysis, and build automation. Each of these quality assurance practices have been studied in depth in isolation, but there is a clear knowledge gap when it comes to our understanding of how these approaches are being used in conjunction, or not. In our study, we broadly investigate whether and how these quality assurance approaches are being used in conjunction in the development of 1,454 popular open source software projects on GitHub. Our study indicates that typically projects do not follow all quality assurance practices together with high intensity. In fact, we only observe weak correlation among some quality assurance practices. In general, our study provides a deeper understanding of how existing quality assurance approaches are currently being used in Java-based open source software development. Besides, we specifically zoom in on the more mature projects in our dataset, and generally we observe that more mature projects are more intense in their application of the quality assurance practices, with more focus on their ASAT usage, and code reviewing, but no strong change in their CI usage.

2.1 Introduction

We have grown accustomed to living in a software-filled world: we rely on our smartphone full of apps, or on digitalised services for many of our daily activities [5]. With software pervading modern society and controlling critical aspects from an economic, safety, security and scientific standpoint, the quality and dependability of that software is indispensable [6],

This chapter is based on *A. Khatami and A. Zaidman. State-of-the-practice in quality assurance in Java-based open source software development. Software: Practice and Experience 54, no. 8 (2024): 1408-1446* [58].

because of the potential catastrophic consequences [7]. To ensure the quality of these software systems, software engineers can use a range of quality assurance approaches, e.g., software testing [8–10, 16], modern code review [15, 16, 59], automated static analysis [18, 19], and build automation [21, 22].

Each of these quality assurance techniques have been studied in isolation in-depth. For example, Beller *et al.* have analyzed the prevalence of automated static analysis tools in open source software development [18], Rausch *et al.* performed an empirical analysis of build failures [25], both Beller *et al.* and Rigby *et al.* investigated aspects of code reviews in open-source software development [26, 27], and Hilton *et al.* focused on understanding continuous integration [24]. Other studies have examined the relationship between two quality assurance practices, e.g., Cassee *et al.* have studied the impact of continuous integration on code review [28], Zhao *et al.* have looked into the impact of continuous integration on other software development practices [29], Zampetti *et al.* did a study on the interplay between pull requests and CI [30], Panichella *et al.* looked into the complementarity of static analysis tools and code reviews [31], and Nery *et al.* did an empirical study of the relationship between CI and test code evolution [32].

Some of these approaches are to be considered complementary to each other, e.g., code reviews and software tests find different issues [33]. Precisely because of the complementarity between the approaches, we set out to broadly investigate whether and how software testing, modern code reviews, automated static analysis, and build automation are being used in conjunction in the development of open source software on GitHub. Github is one of the most important development platforms for open source software and has around 28 million users and 79 million repositories [60]. In addition to it being a git-based version control system, GitHub integrates several elements for quality assurance, e.g., pull requests for code review, continuous integration for build automation and test execution, and integrations for automated static analysis and code coverage.

We have performed our investigation into the state-of-the-practice in quality assurance by analysing **1,454** GitHub projects. Our investigation is guided by the following research questions:

RQ1 What is the current state-of-the-practice in quality assurance in open source software development?

The main goal of our study is to see how quality assurance practices are being followed in conjunction. Before we can study how they are used in conjunction, we need to understand how they are being used in isolation. We fully acknowledge that each of the quality assurance practices has previously been studied in isolation, however, we find it essential to understand the *current* state-of-practice reflected in our dataset that comes from GitHub. We see the study of quality assurance practices in isolation as a cornerstone to build towards studying them in conjunction; we do not claim the study in isolation as a contribution of our work.

RQ1.1 What is the prevalence of quality assurance approaches like software testing, modern code review, automated static analysis, and buildability?

RQ1.2 Which quality assurance approaches are being used in conjunction?

Our set of selected projects consist of popular and active projects on GitHub (1,454 in total). However, we hypothesize that projects that have been more active and are relatively more popular than others follow quality assurance practices with higher intensity. Thus, we aim to see how prevalence of quality assurance practices differs in the top 50 most popular and active projects compared to all the projects in our dataset.

RQ1.3 How do mature projects follow quality assurance practices?

While conducting our research, we have faced a number of challenges studying each of the practices, e.g., it was not easy to build all projects out of the box. Further on in this paper, we provide an overview of these challenges and hypothesize how they are important not only for researchers, but also for practitioners.

Getting a broad understanding of how existing quality assurance approaches are being used in open source development is important for software practitioners, software engineering educators, and software engineering researchers. Specifically, this understanding will inform practitioners about the (combinations of) best practices and how they are currently in use. Furthermore, it will apprise educators on what should be taught in software engineering classes, both in terms of the current state-of-practice and how we can improve on the state-of-practice. Finally, it can inspire researchers to improve both quality assurance approaches in isolation, as well as how their combinations can be strengthened.

In summary, this paper makes the following contributions:

- A large-scale analysis of the usage of quality assurance practices in 1,454 open-source Java-based projects, with particular focus on the local buildability, presence of automated static analysis tools, GitHub continuous integration workflows, code review intensity, and test suite availability and resulting test coverage.
- A unique analysis of whether the aforementioned quality assurance practices are used in conjunction in open source software projects hosted on GitHub.
- A list of challenges related to implementing and understanding quality assurance in open-source projects from the double perspective of researchers and practitioners.
- An extensive replication package to reproduce the results of the study, but that can equally be used by researchers to find projects for a specific investigation with regard to quality assurance practices.

The structure of the remainder of this paper is as follows. Section 2.2 presents the related literature. Section 2.3 details our data collection procedure. Section 2.4 reports on our results for each of the quality assurance practices in isolation and in conjunction. Section 2.5 discusses our findings. Section 2.6 surveys related work. Section 2.7 presents implications and future work. Section 2.8 discusses threats to validity, and Section 2.9 concludes the paper.

There is also an associated package provided to replicate and reproduce the results of our study [61].

2.2 Background

We present literature related to each of the practices in our study as the background knowledge in this section.

2.2.1 (CI) Building, Compilability

Tufano *et al.* have examined the compilability of ~220,000 snapshots of 100 Apache Software Foundation projects [62]. They found that 30% (median) of those snapshots could not be compiled. Maes-Bermejo *et al.* revisited Tufano *et al.*'s work by replicating their study and also reproducing a different set of 80 projects with ~300,000 snapshots [63]. They found that the most influential error causing build failures are missing external artifacts, contradicting Tufano *et al.*'s finding that dependency-issues is the most common cause of build failures. Hassan *et al.* have investigated Java projects on GitHub and report that 101 out of 187 projects could be successfully build using Ant, Maven, or Gradle [64]. Their manual inspection provides indications on the main reasons for build failure: 'backward-incompatibility of JDK and building tools', 'non-default parameters in build commands', and 'project defects in code / configuration files'.

In another study Hassan *et al.* have proposed a model to predict CI build results [65]. More specifically, they have built a classifier using the random forest algorithm and applied it to the TravisTorrent dataset [66] of 402 Java projects and 250,000 builds.

Jin and Servant have proposed an approached called SmartBuildSkip to reduce CI build cost by predicting successful CI builds and skipping them [67]. They base their approach on two hypotheses: most builds pass, and build failures usually happen consecutively (52% of build failures are consecutive).

Lou *et al.* have studied 1,080 questions on Stack Overflow related to build issues of Maven, Gradle and Ant build systems [68]. They have observed a high diversity in build failure categories, however about 68% of them can be fixed by applying fix patterns in the build script code related to plugins and dependencies (the most frequent ones being (1) correcting the plugin setting, and (2) adding missing dependencies).

Rausch *et al.* have analyzed build failures in CI from 14 open-source Java projects linking commits to their CI builds results [25]. As a result, they found 14 common error categories and reported test failures as the most common one among projects.

Zampetti *et al.* empirically investigated how developers use CI builds outcome during a code review in a PR discussion, analyzing 69 projects using Travis-CI on GitHub [30]. They found that passed builds increase the chances of merging a PR, and if a build fails, process-factors (like the age of PR, changed lines and files, review comments, etc.) have stronger correlation. Interestingly, sometimes a PR is merged despite a failing build status, e.g., due to minor warnings from static analysis tools.

2.2.2 Code Review

McIntosh *et al.* have provided empirical evidence that higher code review participation and lower code review coverage, i.e., the proportion of changes that have been code reviewed, will lead to less defect-prone, higher quality software. In their study they considered post-release defects as a proxy for software quality. Since their study was conducted on

Gerrit[1] (a web-based code review tool for git-based software projects) and not on GitHub, we could not use their same metrics to measure code review participation in our projects. Moreover, their study is focused on the impact of code review coverage and code review participation on software quality, whereas we study the relation between code review intensity and other quality assurance practices [69].

Cassee *et al.* have explored code reviews of 685 open source projects on GitHub that used Travis-CI as their CI platform [28]. They found that using Continuous Integration (CI) on average saves one review comment per pull request, indicating that adopting continuous integration can save-up valuable time in code review.

Similarly, Rahman and Roy have investigated the impact of CI on code reviews to understand the relation between CI builds and the participation or quality of the code reviews [48]. They indicate that succeeding automated builds have a positive effect on code review participation. Using code review comments count as a proxy for code review quality, they came to the conclusion that automated build frequency has a significant positive impact on the code review quality.

Rigby *et al.* have conducted a case study of the Apache Server peer review practices as a popular open source project [27]. They observe the number of reviewers that are involved in each review, the frequency of reviews, the size of the artifacts under review, the time interval of reviews, and how many reviews find defects.

Beller *et al.* have investigated to see what problems are fixed when doing modern code reviews in open source projects [26]. They report that 10 to 35% of review suggestions do not lead to any code changes, and they observe a 3-to-1 ratio in maintainability versus functional code review comments.

Bacchelli and Bird coined the term *Modern Code Review (MCR)* [15] as the lightweight and less formal version of heavyweight formal code inspections [70]. They explored the expectations, outcomes, and challenges of MCR by manually classifying comments, and observing, interviewing, and surveying practitioners. Their study found that the main motivation of code reviews is not to find defects/bugs, but to increase awareness and do knowledge transfer.

Kononenko *et al.* have looked into developers' perception of code review quality [46]. They found that review quality is primarily associated with thoroughness of the feedback, familiarity with the code, and the quality of the code. Factors such as code quality, presence and quality of tests, and personality of the developer would have impact on the review decision.

2.2.3 Testing

The purpose of software testing is to improve the quality of software by the detection of faults [35, 37, 39, 71, 72], which makes testing a good candidate among the quality assurance practices that we consider in our study.

Hilton *et al.* have performed a large scale study of test coverage evolution over 7,816 builds of 47 open source projects [40]. To gather coverage data of projects they ran test suits themselves and also used the Coveralls service[2] for some of them. Their main findings are that patch coverage is not correlated with overall coverage, and that patches might

[1] `https://www.gerritcodereview.com/`, last visited Nov 23rd, 2022.
[2] `https://coveralls.io`, last visited May 20th, 2022.

have a significant effect on overall coverage. An observation that was previously also made by Elbaum *et al.* [73].

Zaidman *et al.* have observed that testing activities do not always nicely co-evolve with production code engineering activities [74, 75]. Instead, they observe that testing happens in bursts. Importantly, this observation was made before the pull-based development model became popular. Gousios *et al.* have deeply studied the pull-based development contribution model and found that guidelines for pull request creation include feature isolation, but also conformance to source code quality and test coverage guidelines [76].

Beller *et al.* have tracked how developers go about the test engineering process in their Integrated Development Environment (IDE) [77–79]. From the software engineers that they tracked, around 50% did not carry out any test-related activities (writing or executing test) in the IDE during a 5-month period.

Kochhar *et al.* have found in their study that the relationship between test coverage and post-release bugs is either non-existent or unclear [41]. However, in another study by Athanasiou *et al.*, on the relation between test code quality and issue handling performance, they have proposed a test code quality model, and have used code coverage as a metric to assess test code's completeness. They have also showed that there is a positive correlation between test code quality and throughput and productivity of issue handling [80].

2.2.4 Continuous Integration

Vasilescu *et al.* have investigated the effects of adopting continuous integration on quality of the software and also on the productivity of the development team [20]. They report that using CI leads to more pull requests getting accepted, and that CI improves the productivity of project teams.

Vassallo *et al.* have investigated the concept of *Continuous Code Quality* (CCQ) that they take from SonarQube [81]. CCQ is the idea that CI stands central in ensure code quality by running automated tests and performing automated code inspections at every build [23]. Their main finding after a large-scale analysis of 119 Java projects is that only 11% of CI builds continuously monitor code quality.

Hilton *et al.* have studied the usage, costs, and benefits of CI in open-source projects [24]. They observe that popular projects are more likely to use CI, projects using CI release more often than the ones not using it, projects use CI because it helps them catch bugs early.

Nery *et al.* have looked into the relationship between continuous integration and test code evolution [32]. They study test coverage and test ratio (the proportion of test code in a project) and report that projects using CI have a rising trend in test ratio compared to projects that do not use CI, and that test coverage of CI-using projects grows more compared to projects not using CI.

Hilton *et al.* have studied the barriers of CI usage, the needs of developer using CI tools, their motives, and the benefits they experienced when using CI tools [82]. In their study, developers mentioned the need of user interfaces for modifying CI configurations, and easier configuration of CI services. Developers also mentioned maintaining and setting up CI services as barriers of CI usage. It is worth mentioning that this study was conducted before the introduction of GitHub Actions as a CI platform on GitHub. Since GitHub Actions claims to make the automation of software workflows easier, we are interested to

see the prevalence of today's continuous integration usage by studying Java projects on GitHub more than 2 years after the introduction of GitHub Actions.

Vassallo *et al.* have characterized build failures in CI [83]. Firstly, they observe that 26% of the builds that they have analyzed are failing builds, noting specifically that compilation failures are rare when compared to the frequent testing failures.

Gautam *et al.* have used the TravisTorrent dataset [66] to empirically study continuous integration in open source projects. To help attract and onboard new members [84], they identified key actionable practices for maintainers, including explicitly prioritizing test cases, establishing pull-request documentation standards, engaging lurkers (those who watch but do not contribute), managing expectations around core team involvement, and stating cultural values (e.g., "build fast, fail fast").

2.2.5 Other Elements of Quality Assurance

Gousios *et al.* defined the pull-based software development model [45]. In their exploratory analysis on Github projects at that time they discovered that the pull-based model is not popular. Moreover, they found that most pull requests are small (affected a few dozen lines of code), and only a few (13%) of them are rejected due to technical reasons. Furthermore, they reported that providing guidelines for pull request processing, and having a high coverage test suite will attract external contributors and speed up merging of contributions.

Gousios *et al.*'s study [45] resulted in creation of a dataset of 900 projects and 350,000 pull requests called pullreqs [85]. This dataset was initially created by Gousios and Zaidman, then updated by Zhang *et al.* [86] by adding more projects, pull requests, and data features. In both studies there are similar data features that overlap with features collected in our study, using similar heuristics. Besides, the heuristics that Zhang *et al.* used for collecting CI usage and CI build data are similar to our data collection methodology.

Kinsman *et al.* have examined how software developers use GitHub Actions [87]. GitHub Actions is a GitHub feature to automate tasks based on events like commits, PRs, issues, etc. In their dataset only 0.7% of projects adopted GitHub Action spread across 20 categories (CI, CD, utilities, etc.).

2.3 Study Setup

In this section, we first explain how we selected projects for the study, and then describe the data collection steps in detail. Futhermore, we show how we selected the mature projects out of the main dataset. Figure 2.1 details our data collection procedure in 5 steps, each for a specific quality assurance practice.

Step 1: Build projects locally.
Step 2: Determine usage of Automated Static Analysis Tools (ASATs).
Step 3: Investigate CI usage and build result.
Step 4: Collect code review data from last 20 pull requests.
Step 5: Collect test coverage results.

2.3.1 Studied Projects

We used Dabic *et al.*'s GitHub Search (GHS) dataset [88] to select projects from GitHub based on different criteria. As it is our aim to investigate projects that are likely to be in a

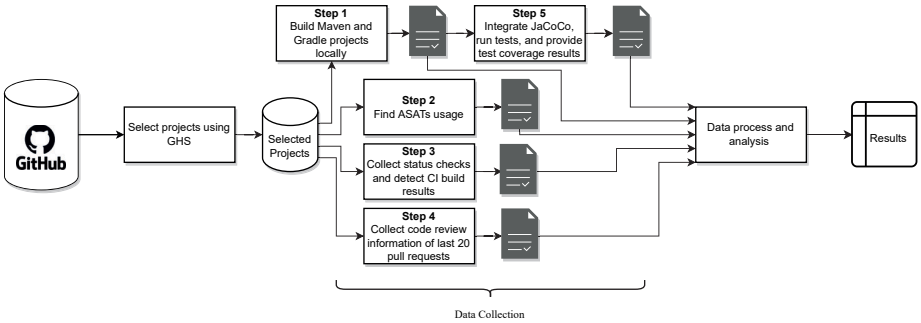

Figure 2.1: An overview of data collection steps.

position to make optimal use of quality assurance practices, we filtered the projects using the following criteria. We selected popular (>100 stars) Java projects that have recently been active (last commit after August 2021). We have settled on one programming language to keep our data analysis pipeline simple, more specifically, we have chosen Java. We have also selected projects to minimally have 10 contributors, as our line of reasoning is that you need to have a sufficient number of developers to make optimal use of quality assurance practices like code reviews. In addition, we have chosen to only study non-forked projects to avoid duplicate projects. Finally, we put a lower limit of 200 pull requests for a project to be part of our data set; this threshold ensures that we study projects that use the pull based development model, and are thus potentially also doing code reviews on the GitHub platform.

We also saved the latest commit hash for each project, and in the next steps, we use this commit hash to run git checkout. This enables (1) us to study each of the projects in a specific state, and (2) to easily replicate our work in future [61].

2.3.2 Data Collection

In this section we go step by step and explain the path we took to collect data related to each quality assurance practice.

Step 1. Local builds

In this step we describe how we have collected information on the buildability (compilability) of projects. This information first shows us which projects can be built out of the box (without any special configuration, reading project specific documents, and environment setup), only using a minimal environment (a Docker container of Ubuntu 21.04) with the Java Development Kit (Open JDK 11) and build tools such as Maven and Gradle. Secondly, we will use this information to filter successfully built projects and generate test coverage results for them in Step 4.

Starting from the collection of GitHub Java projects, first we filtered the projects using Maven and Gradle as their build system. To do this we searched for *gradle.build* and *pom.xml* configuration files inside (the root path of project) repositories by querying GitHub. Next,

Build System	Build Command
Gradle	`./gradlew clean assemble --stacktrace`
Maven	`mvn clean package -DskipTests --batch-mode -e`

Table 2.1: Build commands for different build tools.

we specifically filtered out Android projects (by searching for the *AndroidManifest.xml* file inside repositories) to avoid build failures related to Android SDK configurations.

After removing non-Gradle, non-Maven, and Android projects, we obtain the set of projects to be built locally. In the next step, we automatically clone projects and after doing git checkout on the latest commit hash, build projects locally using a docker container with *Ubuntu 21.04*, *OpenJDK v11*, *Gradle v7.1*, *Python v3.9*, and *Maven v3.8.1* installed. Similar to Sulír *et al.* [89] we skip running tests when building projects in this step to avoid test failures. We add relevant flags to our build commands to have full log information in build output logs (see Table 2.1). Furthermore, we put one hour maximum build execution time to avoid infinite builds (similar to Sulír *et al.* [89]).

Step 2. Automatic static analysis tools

Similar to the previous step, we again use GitHub's API to see if there is a configuration file related to one of the more popular Java static analysis tools (SpotBugs, Checkstyle, and FindBugs) in the repositories, similar to Beller *et al.* [18]. To do this, after running the git checkout on the latest commit hash, we looped through the repository Git tree to check whether it contains a file with a substring of *'spotbugs'* in their name phrase. If that is the case, then the project is marked as (likely) to use the SpotBugs tool. We performed a similar step for Checkstyle and FindBugs.

Step 3. Continuous integration

In this step, we collected the required data to study both the prevalence of CI and the results of the build steps that are part of CI. For the prevalence of CI, we need CI usage data (whether a project is doing continuous integration). In terms of the result of CI, we are interested in two distinct steps: (1) the overall CI build success, and (2) the result of specifically the build phase of the CI workflow. We need the latter result, to be able to compare the result of the CI build phase, to that of a local build result from Section 2.3.2.

Step 3.1. CI usage: To establish whether a project uses continuous integration, we have decided to use a different approach to earlier studies on this topic. In particular, prior studies have used the TravisTorrent dataset [66], or the Travis-CI API to check whether a project is using CI [28]. Also, Vassallo *et al.*'s work only considered SonarCloud as a CI platform to study projects' code quality [23], and Gallaba *et al.*'s study was limited to CircleCI builds [90]. These aforementioned approaches are limited to one particular CI platform, namely Travis-CI, SonarCloud, or CircleCI. To consider different CI platforms together, we followed a different route to establish CI usages, namely the usage of *checks* on GitHub[3].

[3] `https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/collaborating-on-repositories-with-code-quality-features/about-status-checks`, last visited May 20th, 2022.

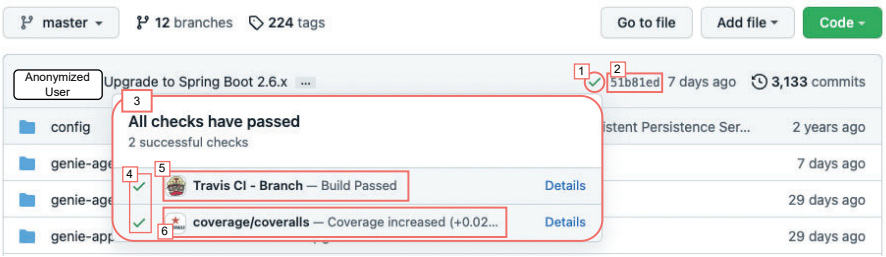

Figure 2.2: Status checks on a commit in the main page of a repository: 1. The combined result of all status checks (in this case successful), 2. Related commit hash, 3. List of all status checks and a summary of their results, 4. Each status check's result, 5. A check by a GitHub App (TravisCI), 6. A status check marked by an external service (Coveralls), these status checks are called "statuses".

Using GitHub Actions[4] (GHA) projects can automate their workflows, including CI/CDs on GitHub. Events like pushing a commit, creating a pull request, opening an issue, etc. can trigger a workflow run (also referred to as a *check*). In this paper we collect information related to *checks* that run after pushing a specific commit, namely the latest commit SHA of projects in our dataset.

There are two types of *checks* that can run after pushing a commit[5]: (1) the ones defined by workflows, which are only available for use with GitHub Apps, and (2) the ones that are marked with external services through the GitHub API (these are called *Commit Statuses* in GitHub's documentation), shown in Figure 2.2. From now on, similar to GitHub's documentation, we will call the first type: "checks", the second type: "(commit) statuses", and both together: "status checks".

What is important to note is that not all status checks are related to CI, and as such can not be unequivocally regarded as a sign of CI usage. To address this, we heuristically filtered CI related status checks. Table 2.2 shows data attributes of status checks depending on their type. This information is collected from the GitHub GraphQL API[6].

To filter CI related status checks, depending on their type, we look for some keywords in *context*, *targetUrl*, and *description* of statuses, and for checks we look for usage of specific GitHub Apps related to CI. For GitHub Actions App[7] (GHAA) checks, the most used GitHub App, we have also looked to see if they contain any "build" keyword in their *name*. This is because using GHAA does not always mean a project is using CI, however if there is the build keyword, there is a strong signal of the project doing CI. Table 2.3 shows the keywords used in filtering CI related status checks.

Step 3.2. Overall CI results: To see what is the final state of CI workflows run in a specific state of the project (latest commit hash), we rely on the status checks information. Status checks data collected from the GitHub API contains a status check attribute that

[4] `https://docs.github.com/en/actions`, last visited May 20th, 2022.
[5] `https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/collaborating-on-repositories-with-code-quality-features/about-status-checks`, last visited May 20th, 2022.
[6] `https://docs.github.com/en/graphql`, last visited May 20th, 2022.
[7] GitHub Actions App is a GitHub App and differs from GitHub Actions which is GitHub's CI/CD platform. In this paper, with GHA we refer to the CI/CD platform on GitHub, and with GHAA we refer to the GitHub Actions App on GitHub's marketplace.

Checks' Attributes		
Name	**Description**	**Value(s)**
conclusion	The conclusion of the check run.	ACTION_REQUIRED / CANCELLED / FAILURE / NEUTRAL / SKIPPED / STALE / STARTUP_ FAILURE / SUCCESS / TIMED_OUT
name	The name of the check for this check run.	String
status	The current status of the check run.	COMPLETED / IN_PROGRESS / PENDING / QUEUED / REQUESTED / WAITING
summary	Summary of the check run.	String
text	Text of check run.	String
title	Title of check run.	String
app	The GitHub App which created the check suite containing this check run.	Name: String Slug: String
workflow run	The workflow run associated with the check suite containing this check run.	Number: Int URL: URI
workflow	The workflow executed in the workflow run.	Name: String

Statuses' Attributes		
Name	**Description**	**Value(s)**
context	The name of this status.	String
description	The description for this status.	String
state	The state of this status.	ERROR / EXPECTED / FAILURE / PENDING / SUCCESS
targetUrl	The URL for this status context.	URI

Table 2.2: Data attributes collected for status checks from GitHub GraphQL API.

Type	Attribute	Keywords
Check	app name	GitHub Actions, Azure Pipelines, ci.jenkins.io, ASF Cloudbees Jenkins ci-builds, Travis CI, CircleCI Checks, Cirrus CI, Confluent-Jenkins-CI
Status	context	ci, continuous-integration, kokoro, bazel
	description	teamcity, kokoro
	targetUrl	ci., screwdriver, buildkite, teamcity

Table 2.3: Keywords used to filter CI related status checks based on their type.

represents the rollup for both checks and statuses for a commit[8] (also referred to as combined status[9], however only for statuses). We also refer to this rollup result as the CI result of each project for the latest commit hash in our dataset. Of course this only applies to the projects having at least one filtered CI related status check.

Step 3.3. Result of build phase in the complete CI workflow: Having local build results in hand from the first step of our data collection (see Section 2.3.2), we want to see

[8] `https://docs.github.com/en/graphql/reference/objects#statuscheckrollup`, last visited May 20th, 2022.

[9] `https://docs.github.com/en/rest/commits/statuses#get-the-combined-status-for-a-specific-reference`, last visited May 20th, 2022.

Type	Attribute	Keyword(s)
Check	title	built, build
	text	build
	name	build
Status	context	build
	description	built, build

Table 2.4: Keywords used to filter CI builds status checks based on their type.

how a CI build outcome of a project matches its local build outcome. Therefore, we first heuristically filter build related CI status checks by searching for "build"/"built" keywords in their attributes (Table 2.4 shows the details). Second, based on the results of each of the filtered CI builds, we determine the rollup result of CI builds. Meaning if all statuses' *state* and all checks' *conclusion* is "SUCCESS", then the rollup status of CI build is success, otherwise it is failure.

Step 4. Code review

To collect information about projects' code review activity, we focus on the user's code review behavior during pull-based development in the GitHub environment. When a contributor creates a pull request to merge their contribution to the main repository, integrators or co-contributors can review the pull request [76]. The contributor who created the pull request and the code reviewer(s) can have a conversation through comments. For our dataset we count the number of reviews and the number of comments for the last 20 merged pull requests of projects (reference data August 1st, 2021). We set the specific reference date to ensure future reproducibility. Reviews are defined as pull request review comments with an optional body comment and a state[10]. Approving changes are also considered as a review action. With comments, we mean other comments in a pull request discussion part that are not posted by GitHub bots [43], and are not made on a unified diff and are usually meant for general communication. We analyze the combination of these two data points to show how code review is taking place for a project on GitHub.

We only consider merged pull requests (based on the GitHub API[11]). To avoid getting into a loop of checking all pull requests to see if they are merged or not, we limit ourselves to the last 100 pull requests. Important to note is that if a project is using Git instead of GitHub to merge pull requests [4], it will not be considered for code reviewing in our dataset, since our approach only monitors code reviews performed through the GitHub web interface.

Furthermore, for our correlation analysis in Section 2.4.6 we collected two other data points for each of the pull requests: total count of lines changed and count of files changed in all commits. We have combined them to formulate our code review rate metric (Equation 2.1). We settled on this equation for two key reasons. The first one being that we want to include both count of comments and count of reviews, as they are both important during the review process. The second reason being that we want to normalize the combined

[10] `https://docs.github.com/en/rest/reference/pulls#reviews`, last visited May 20th, 2022.

[11] `https://docs.github.com/en/rest/reference/pulls#check-if-a-pull-request-has-been-merged`, last visited May 20th, 2022.

count of comments and reviews over the number of changed lines of code divided by the number of changed files. This normalization enables us to more easily compare the review intensity of a changed piece of source code.

$$\frac{\#Reviews + \#Comments}{\frac{\#TotalChangedLines}{\#TotalChangedFiles}} \tag{2.1}$$

Equation 1: Code review intensity metric formula. Please note that the values of both numerator and denominator are aggregated across the 20 last merged pull requests per project.

Validation of the code review intensity metric

We carried out a validation of the code review intensity metric by investigating pull requests that show a wide variety in code review intensity scores. In particular, we look at code review intensity scores that lie at the 20th, 40th, 60th, and 80th percentile; we also consider the minimum and maximum code review intensity scores. We then investigate pull requests of 6 projects for a deeper comparison, in which we also checked the discussions happening during code review to better gauge the intensity of the code reviewing happening.

Project Name	Code Review Intensity	Percentile	PR-ID	Change Requested	# Reviewers	Merged By Other User	# Review Comments	# Comments	# Lines Changed
kiegroup/ kogito-runtimes	57.41	Maximum	1473	Yes	2	Yes	7	9	8
			1482	No	3	Yes	0	3	69
			1465	No	2	Yes	0	17	51
			1488	No	2	Yes	0	3	2
			1476	Yes	4	Yes	31	25	1094
apache/ activemq-artemis	2.44	80th	3679	No	1	Yes	0	0	6
			3678	No	0	No	0	0	2
			3677	No	1	Yes	3	3	74
			3676	Yes	3	Yes	0	15	76
			3674	No	1	Yes	0	0	2
			3673	Yes	2	Yes	15	11	163
			3668	No	1	Yes	0	1	17
			3665	Yes	2	Yes	3	0	67
			3664	No	1	Yes	0	0	9
			3661	No	1	Yes	0	1	57
			3660	No	0	No	0	0	4
			3658	No	0	No	0	0	828

2

Project Name	Code Review Intensity	Percentile	PR-ID	Change Requested	# Reviewers	Merged By Other User	# Review Comments	# Comments	# Lines Changed
			3654	No	1	No	0	2	235
			3651	Yes	1	No	0	2	698
			3650	No	2	No	0	13	5
			3649	Yes	1	Yes	11	0	67
			3648	No	1	Yes	0	0	2
			3647	No	0	No	0	0	145
			3645	No	1	Yes	0	0	10
			3641	No	0	No	0	0	323
radargun/ radargun	1.66	60th	699	No	0	Yes	0	0	6
			698	Yes	1	Yes	1	1	116
			697	No	0	No	0	0	2
			696	No	1	Yes	0	1	268
			695	No	1	Yes	0	1	10
			694	No	1	Yes	0	1	200
			692	No	2	Yes	1	0	4
			691	Yes	1	Yes	2	0	94
			690	No	1	Yes	1	1	296
			689	Yes	1	Yes	4	1	95
			688	No	1	Yes	0	1	329
			687	No	1	Yes	0	1	8
			686	No	1	Yes	0	1	2
			685	No	1	Yes	1	1	595
			684	No	1	Yes	0	0	27
			683	No	0	No	0	0	2
			682	No	1	Yes	0	0	42
			681	No	1	Yes	0	0	4
			680	No	1	Yes	0	1	11
			678	No	0	No	0	0	2
			954	No	1	Yes	0	3	3
			953	No	1	Yes	0	0	2
			950	No	1	Yes	0	0	4
			949	No	1	Yes	0	1	5
			948	No	1	Yes	0	0	2
			946	No	1	Yes	0	0	14
			945	No	1	Yes	0	3	6
			940	No	1	Yes	0	0	2
wocommunity/ wonder	0.54	40th							

Project Name	Code Review Intensity	Percentile	PR-ID	Change Requested	# Reviewers	Merged By Other User	# Review Comments	# Comments	# Lines Changed
			938	No	1	Yes	0	0	4
			937	No	2	Yes	0	1	9
			936	No	1	Yes	0	0	2
			934	No	1	Yes	0	0	3
			933	Yes	2	Yes	0	3	978
			926	No	2	Yes	0	2	16
			925	No	1	Yes	0	3	4
			924	No	1	Yes	0	0	3
			923	No	1	Yes	0	0	13
			920	No	1	Yes	0	0	77
			919	No	1	Yes	0	0	56
			918	No	1	Yes	0	1	2
uwetrottmann/ seriesguide	0.15	20th	787	No	1	Yes	0	1	6
			798	No	0	No	0	1	584
			769	No	0	No	0	0	45
			810	No	0	No	0	1	11456
			790	No	0	No	0	1	986
komamitsu/ luency	0	Minimum	199	No	0	No	0	0	2
			223	No	0	No	0	0	3
			192	No	0	No	0	0	2
			248	No	0	No	0	0	2
			204	No	0	No	0	0	2

Table 2.5: Summary of code review information in a number of selected projects based on their code review intensity. The code review intensity was calculated over the 20 PRs that we have analyzed per project.

For the validation, we consider the following six observations with regard to code reviews. The fifth column of Table 2.5, entitled **Change Requested**, indicates whether any change is requested by the reviewer(s) during the discussion and whether there is a commit after that request to apply the requested change. The sixth column indicates the number of reviewers involved (**# Reviewers**), and **Merged By Other User** indicates whether the pull request is merged by any user other than the creator/author of the pull request. Then, in column 8 we list **# Review Comments**, specifically the comments on a portion of the unified diff made during a pull request review. Column 9 shows the **# Comments** (comments on discussions part, except the first one and the ones by bots), and column 10 tabulates the **# Lines Changed** (in all the commits of a pull request).

Table 2.5 depicts a number of projects and their associated code review metric values. We focus on the minimum, maximum, and the code review intensity values at the 20th, 40th, 60th, 80th percentiles, to look at a good spread in terms of code review intensity values. Accordingly, we investigate whether if the code review intensity value goes up, the other indicators are typically also increasing. As an example, the project with the highest code review intensity (*kiegroup/kogito-runtimes*) has the highest number of reviewers per pull request on average. Moreover, regardless of the amount of changes, it has at least 2 reviewers for each pull request. Whereas, if we look at the project with the lowest code review intensity (*komamitsu/luency*), we see that there is no sign of having any kind of code review before merging the pull requests. A potential explanation can be that they use another platform to do their code reviews (as mentioned in our threats to validity), but for this specific project this seems unlikely, as there is only a single contributor. Also, by looking at the project at the 20th percentile (*uwetrottmann/seriesguide*), we see for 4 pull requests out of 5 that the user creating the pull requests is also merging them. In addition, we see very small numbers of comments being made.

For projects at the 40th, 60th, and 80th percentile, we obtain very similar observations in terms of number of reviewers, number of comments, etc. in the 5 randomly selected pull requests for this analysis. That is why for these categories, we expanded our investigation to 20 pull requests in total. Looking at the number of reviewers, or whether a pull request was merged by another user, we can not make a definite judgement about the code review intensity, since the difference in observations are not obvious. But, if we look at the change requested column we see that for projects in 80th, 60th, and 40th percentile in total have respectively 5, 3, and 1 changes requested, which follows our code review intensity metric values. Moreover, the total number of review comments for projects in 80th, 60th, and 40th percentile are respectively 32, 10, and 0. And the total number of their discussion comments are 48, 11, and 17. Adding up both numbers gives us 80, 21 and 17 review comments and discussion comments in their last 20 pull requests, showing that the intensity of code review decreases as we go from the 80th percentile to the 40th percentile category.

Step 5. Testing

To establish whether software projects do testing and at what level of intensity, we measure branch coverage. In order to do so, in this step we build the projects (that we have already successfully built in Step 1) and run their tests. We did remove some buildable projects from our dataset, in case those projects had no code or no tests. For all build configuration files in a project (`build.gradle` for Gradle or `pom.xml` for Maven) we search for the `src/main` path for source code and the `src/test` path for tests in the same directory. If we could not find any code or test directory for that project then we marked them as projects with no code or no test and removed them for this analysis step.

To establish code coverage, we add the JaCoCo plugin[12] to the projects' build configuration file. We choose JaCoCo because of its popularity, slightly higher visibility, and easier integration [91]. However, integrating JaCoCo with Gradle or Maven projects is not a trivial task. To integrate JaCoCo with a project we need to insert specific configurations in projects' build file (`pom.xml` or `build.gradle`). Depending on the structure of the project (having sub-projects or parent-projects), usage of other plugins, build system of the project

[12]JaCoCo `https://www.jacoco.org`, last visited May 20th 2022.

(Gradle or Maven), and other configurations (in some cases tests were disabled manually in build configuration) we had to change specific lines of the build configuration. Moreover, for Gradle projects there are two different domain specific language (DSL), i.e., Groovy and Kotlin which makes it non-trivial to automatically apply the JaCoCo plugin to projects without causing a build failure.

The process of automatically integrating the JaCoCo plugin for hundreds of projects has proven extremely cumbersome. To the best of our knowledge, we are the first to have created an automatic process for this task. To establish an automated process that works for a majority of projects, we had to finetune our script to cater for different build script structures. Moreover, for Maven projects we usually had to build projects in several steps, since it is quite common for a project to have multiple `pom.xml` files, e.g., a `pom.xml` file per module. On the other hand, for Gradle projects we found it easier to build a project, since when building the parent directory all child modules would be built. In the case of Maven projects, we have encountered projects were we could not produce complete coverage results, because not all modules of a project could be built successfully. Also, there are other reasons for not having complete coverage results for a project, for example, when not all modules of a project have tests, when unconventional paths for source code and test code folders are used, or when our general approach of automatically inserting the JaCoCo commands into the build file conflicts with specific build file configurations of a project. For these special configurations, we would need a manual action to insert the JaCoCo command. This is one reason why we have partial coverage results for some projects.

Mature Projects

In the results section, we will investigate how the maturity of projects possibly affects the results of our analysis. For each of the analysis steps that we perform, we will compare how a practice is either followed similarly or differently among mature projects compared to the overall set of projects.

We first need to establish what we consider to be a mature project. We specifically focus on two indicators of projects, namely *popularity* and *activity*. As a proxy for the activity of a project, we consider the following project attributes: the count of commits, pull requests, issues, and contributors. Projects that we earmark as being more active, have higher counts of the aforementioned attributes. To approximate the popularity of a project, we take the number of forks, stars, and watchers into consideration, again considering higher values for these attributes as an indication that a project is more popular. We consider a mature project as a project that has higher activity and higher popularity.

We have used the following procedure to select mature projects in our dataset. For each of the aforementioned attributes, we first select the projects that have a score for that attribute that is in the top 10%. Since the distribution of the attributes is right skewed, projects that are at the 90th percentile and up have (very) high values for a particular attribute, making it reasonable to consider them as active or popular. However, as an extra condition for establishing that these projects are indeed active or popular, we require them to be at the 90th percentile or above for at least two popularity attributes, *and* two activity attributes.

2.4 Results

In this section we will discuss the results of each of the analysis steps that are depicted in Figure 2.1, which represents our data processing workflow. After presenting the results per analysis step, we elaborate on how the different quality assurance practices are used in parallel in Section 2.4.6.

In order to investigate whether mature projects are implementing the quality assurance practices in the same way as other projects, we follow the procedure of Section 2.3.2. In doing so, we have established 50 out of 1,454 projects as being more mature than most projects. Our subsequent analyses of mature projects is thus based on these 50 projects[13].

The results presented in each part of this section will be used in our discussion in Section 2.5 to present a global answer to our research questions.

2.4.1 Local Builds

As explained in Section 2.3.2, starting from a set of 1,454 projects in our study, we filtered out non-Gradle, non-Maven and Android projects. Moreover, we also removed Gradle and Maven Projects that did not have any build configuration file in their root directory. Furthermore, in cases where we recognized both Gradle and Maven configuration files inside a project, we have labelled them as Gradle projects, our reasoning being that the Maven configuration file might be a legacy element, as the Gradle (°2008) build system is newer than Maven (°2004). The aforementioned steps left us with a total of 1047 projects that we could build locally (~72% of the original dataset).

To filter successful and failed builds we searched for BUILD SUCCESSFUL / BUILD SUCCESS and BUILD FAILED / BUILD FAILURE phrases in respectively Gradle and Maven build logs. However, in some cases none of the aforementioned phrases were found in build logs due to: (for Gradle projects) missing libraries that prevented running the build process, encountering errors when processing Maven's `pom.xml` (Project Object Model) file, and encountering errors because of missing environment variables configuration. Even though there was no explicit build failure phrase in the build log, we labeled these builds as failed since no successful build was achieved. Moreover, we also consider the following cases as failed builds: builds in which the build command could not be executed (e.g., due to a missing commit in git tree), and builds that could not finish their execution (reaching the one hour time limit). The summary of local build results is shown in Figure 2.3. From this graph, we observe that ~63% of projects could be built successfully out of the box. We do observe that that despite Gradle being a newer and more popular build system [92], among our selection of projects, it exhibits a relatively lower successful build percentage (56%) compared to Maven (66.6%).

We analysed a selection of failed builds to better understand why builds fail. To this end, we first randomly selected 50 projects from the total of 388 projects with build failures, then, we manually labeled them according to their build log outputs. In Table 2.6 we list the most frequently found build failure categories from our sample. A key observation that we can make is that build failures related to issues with dependencies and compilation errors account for around half of build failures. For 11 projects the cause of failure was unclear due to limited build logs, which were excluded.

[13] List of these projects and the details on the analysis are available in our replication package.

Category	#	%
Compilation	13	26%
Dependency	10	20%
Configuration	6	12%
Parsing	5	10%
Assembling	3	6%
Other	2	4%

Table 2.6: Top categories of local build failure errors

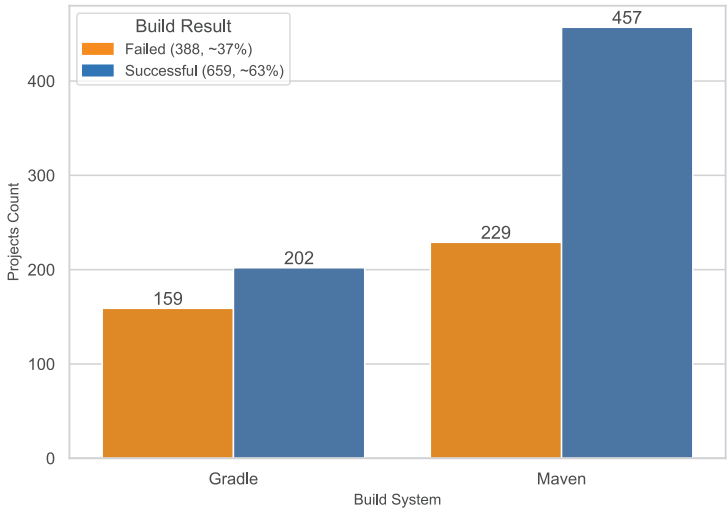

Figure 2.3: Summary of local build results based on build systems.

Observation 1

~63% of projects build successfully and Maven projects show a higher percentage of successful builds (~67%) compared to Gradle projects (56%).

Local Builds in Mature Projects

Looking at the local build results among the 50 mature projects, we see that 21 projects build successfully out of the box, from the total number of 33 eligible projects to build locally. Meaning, 64% of mature projects build locally, which is nearly the same ratio when comparing to other projects in our dataset.

All Projects	Use CheckStyle	Use FindBugs	Use SpotBugs	Use no ASAT	Use 1 ASAT	Use 2 ASATs
1,454	519 (~36%)	3 (~0%)	104 (~7%)	906 (~62%)	470 (~32%)	78 (~5%)

Table 2.7: ASATs usage results.

Observation 1*

Mature projects have roughly the same percentage of local build success (64%) comparing to all the projects in our dataset (~63%).

2.4.2 Automatic Static Analysis Tools

Overall, we observe that in 38% (548) of all 1,454 projects in our study we found evidence of Checkstyle, Findbugs, or SpotBugs Automatic Static Analysis Tools (ASATs). Table 2.7 shows the frequency of ASAT usage, and also indicates that no project uses more than 2 ASATs in parallel. Among our set of studied projects, Checkstyle is the most popular ASAT with ~36% projects using it.

Observation 2

38% of projects use at least one ASAT.

ASATs Usage Among Mature Projects

Unlike local builds, we observe a great difference in the ratio of mature projects using ASATs. We see evidence of ASATs usage in 70% of mature projects (35 out of all 50). This shows mature projects tend to invest more in ASATs.

Observation 2*

70% of mature projects use at least one ASAT.

2.4.3 Continuous Integration

We set out to study two aspects of Continuous Integration: firstly, the prevalence of the use of CI in our study set, and secondly, a comparison between CI build results and local build results. As stated in Section 2.3.2, data about CI related activities are derived from GitHub status checks. We will study the usage of status checks separately.

Prevalence of CI

As explained in Section 2.3.2, we study the use of CI for a project at a specific point in time, namely the latest commit hash in our dataset. At that point, we have results stemming from 1131 projects using external services that mark their *Commit Status* or are using GitHub Actions to run *Checks* in the form of workflows. Overall, we found that checks (6,906 checks) are more frequently used by projects compared to statuses (1,224 statuses).

# All Projects	# Projects *Not* Using CI	# Projects Using CI		
1,454	542 (37%)	912 (63%)	Use Checks	613
			Use Statuses	200
			Use Both	99
			Successful State	695 (76.2%)
			Unsuccessful State	219 (23.8%)

Table 2.8: Counts of projects using CI and the rollup result of CI status checks. Projects might be using checks or commit statuses (or both), either way they are considered as projects using/following CI practice.

However, not all checks and statuses can be considered as a use of CI. So, we came up with a heuristic to filter CI related ones (explained in Section 2.3.2). After this filtering we established that 912 (62.7%) projects use status checks in their CI pipelines: 613 projects only used CI checks, 200 projects only used CI commit statuses, and 99 projects used both (more details in Table 2.8).

Looking at the rollup state of status checks of projects using CI, we obtained an overview of CI results of our projects. In doing so, we observe that 695 (76.2%) of the 912 projects using CI have a successful state. The other 219 (23.8%) projects had an unsuccessful state (2 pending and 217 failure).

Moreover, when we look at counts of status checks on the latest commit of projects, we observe that projects do not have the same approach when following CI practice: there are projects with more than 100 CI checks and also projects with only one CI check. Figure 2.4 depicts the frequency of the number of status checks in projects using CI (median = 3, and average = 7.35).

To have an idea of why some of the projects have a high number of status checks and what is the purpose of it, we selected one project with more than 50 status checks, namely the Checkstyle project which has 60 status checks[14]. The project runs many GitHub Apps (such as GitHub Actions App, Cirrus CI, Travis CI, Azure Pipelines, and CloudBees). Additionally, Checkstyle uses many external services to mark a commit's status (such as Drone CI, Semaphore CI, AppVeyor, Circle CI, and Wercker). When analysing these status checks, we observed some of the motives for running them, including: checking broken links / closed issue references, code analysis to discover vulnerabilities across the codebase using CodeQL, running mutation analysis for different modules using PIT, and building the project with different versions of the JDK and OS (so-called multi-environment builds, see Beller et al. [21]). Outside the scope of our current investigation, it would be interesting to study the various checks that software projects use in more detail in future work to (1) better understand what quality aspects developers care about, and (2) why several CI services and GitHub apps are used in conjunction.

With the CI usage results at hand in Table 2.8, and with our results from local builds available in Section 2.4.1, we set out to investigate the relation between using CI and buildability. Specifically, we aim to understand whether projects that use CI in their development process are more likely to build locally without failure. Figure 2.5 shows the

[14] `https://github.com/checkstyle/checkstyle/search?q=4ce315d27d387b48befd667c7ed8426a1bad6dbb&type=commits`

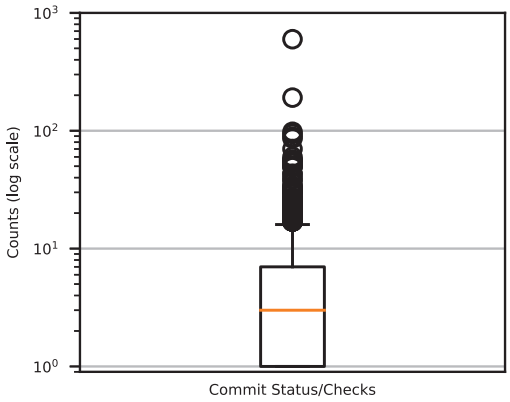

Figure 2.4: Counts of status checks among projects following CI practice.

summary of CI usage based on our local build results. Using the Chi-square test ($\chi^2 \simeq 6.5$, p-value $\simeq 0.01 < 0.05$, and Cramer's V $\simeq 0.08$), there is evidence that a weak, yet statistically significant correlation between CI usage and building locally exists[15].

Observation 3

~63% of projects use CI with ~76% of them having a successful state. There is a weak, yet statistically significant positive correlation between CI usage and building successfully locally, meaning that using CI does increase the likelihood of a successful local build.

CI Builds vs. Local Builds

In Section 2.4.1 we have reported on the local build results of projects. Now with the CI results available, we select CI-related checks from the entire collection of checks to compare them with our local builds. The goal of this comparison is to see whether having a successful/failed build on CI is correlated with a successful/failed build on a local machine. Furthermore, we will dive in to see if there is any relation between having more CI builds and a successful build on a local machine. Our reasoning here is to see if building several times on CI, which we assume to be multi-environment builds, can increase projects' buildability on a local machine with common settings.

After filtering build-related CI status checks (explained in Section 2.3.2) we obtain 2846 build status checks for a total of 843 projects. It is obvious that there exists more than one CI build for some projects, thus we looked into the data to see what are the reasons: building with different versions of JDK / operating systems (e.g., multiple integration environments [21]), building different modules separately, building and running tests in different blocks, and building with/without cleaning cache.

Next, we have combined CI build results for each of the projects to then compare the overall CI build result to the build results of the local machine (explained in Section 2.3.2).

[15]The effect size interpretation for Cramer's V is: $0 \leq V \leq 0.2$ is weak, $0.2 < V \leq 0.6$ is moderate, $V > 0.6$ is strong

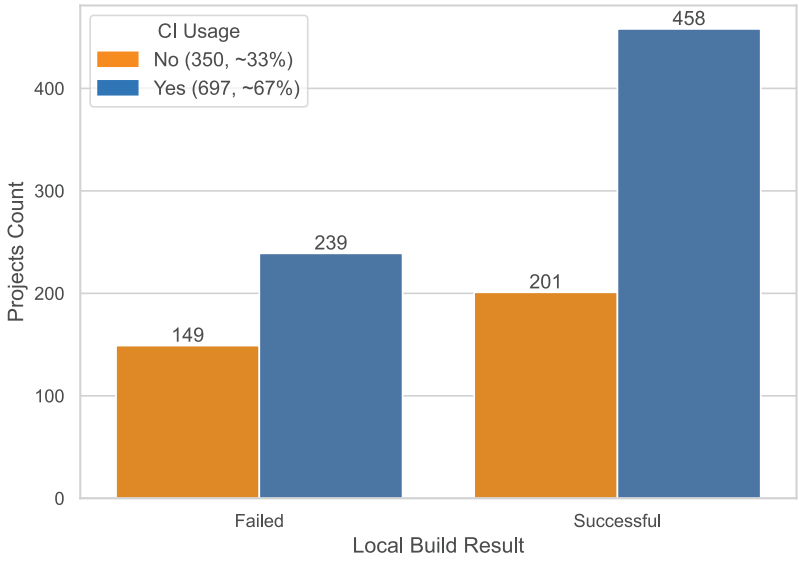

Figure 2.5: Summary of CI usage in projects based on their local build results.

Overall, we observe 83.5% successful CI build results among 843 projects using CI build status checks. Next, we have compared CI build results with local build results using the Chi-square test. We have obtained values $\chi^2 \simeq 6$, p-value $\simeq 0.014 < 0.05$, and Cramer's V $\simeq$ 0.10. This indicates a weak, yet statistically significant correlation. This test and the one reported above form a family of two tests over the same set of projects; after applying the Holm-Bonferroni correction within this family, both remain significant (adjusted p-value $\simeq 0.022$ for both), so Observations 3 and 4 are unaffected [93]. To put things into context, in 64% of projects (from a total of 644 that we had both CI/local build results) we found the same CI and local build results (shown in Figure 2.6).

Observation 4

For ~64% of projects we found matching CI build related status checks and local build outcomes. Using the Chi-square test, we observe a statistically significant weak correlation between successful local builds and CI builds.

Prevalence of CI in Mature Projects

The CI usage ratio among mature projects is almost the same as what we observe in all the projects in our dataset. 62% of mature projects are using CI (using the same heuristic method to filter CI usage). Also, among those using CI, 16 (out of 31) projects had a successful rollup state. This is ~52%, which is less than the ~76% successful CI state among all the projects. One question worth raising here is why mature projects have less successful CI states? A preliminary investigation shows that mature projects have more status checks in their CI workflows. Specifically, on average they have 10 (median = 4) status checks,

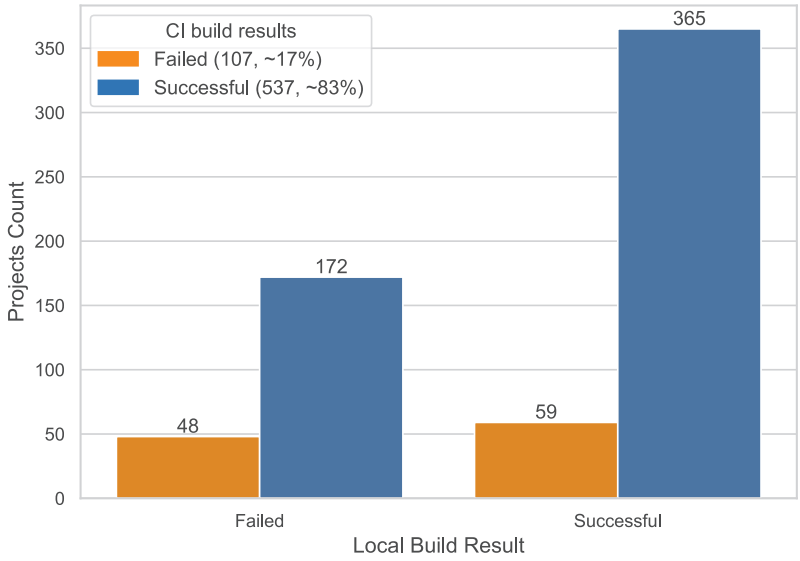

Figure 2.6: Summary of CI build results based on local builds.

compared to other projects that have on average 7 (median = 3) checks. However, the number of projects in our mature set is not big enough to show statistical significance.

Observation 4*

We observe that 62% of mature projects use CI with ~52% of them having a successful state. When we compare mature projects against the general set of projects, we see that a similar ratio of projects use CI, however mature projects have a lower percentage of successful CI states, ~52% compared to ~76%.

2.4.4 Code Review

As a quality assurance practice code review has been shown to be able to improve the quality of software, quality of changes, and quality of system design [44]. To study how code reviewing is done in our set of GitHub projects, we investigate the users' activities that are related to code reviewing during the merging of pull requests (PRs) (also see Section 2.3.2). More specifically, we collect the count of comments, reviews, total changed lines, and total changed files for the last 20 merged PRs of 1398 projects. Another 56 projects did not meet our data collection condition of having at least 20 merged PRs while checking the last 100 PRs.

To study the state of code review as a quality assurance practice, we start without making any assumptions with regard to what can be expected in terms of the intensity of code review activities. To keep our assumptions in check, we use a benchmarking approach that is taken from Alves *et al.* [94]. Specifically, the benchmarking enables us to categorise

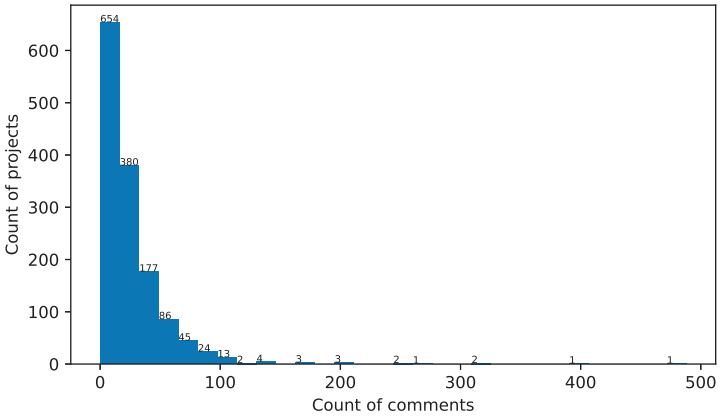

Figure 2.7: Distribution of count of comments in projects' last 20 merged PRs.

projects into four different groups: projects with very high, high, moderate, and low code reviewing. To determine the thresholds for the 4 categories we use the 70th, 80th, and 90th percentiles of the sum of count of reviews and comments in each of projects as the boundaries. These specific boundaries have previously been empirically established and used in the SIG quality model from Alves *et al.* [94] and the test code quality model of Athanasiou *et al.* [80]. An important prerequisite to being able to use this benchmarking approach is that the distribution of values follows a power-law distribution. In order to determine whether the distribution of count of reviews and comments indeed follows a power-law, we plot the values for our metrics for the last 20 merged PRs in the projects that we consider in Figure 2.7 and 2.8. Both metrics seemingly follow a power-law-like distribution: count of reviews and comments are mostly low, but there are a few cases with higher values. As such, we can apply Alves *et al.*'s calibration method to derive different categories of code reviewing intensity [94].

Considering the last 20 merged PRs, projects with less than 38 reviews fall in the low level of code reviewing, projects with review counts between 38 and 50 are classified as doing a moderate level of code reviewing, the ones with review counts between 50 and 73 are considered as projects with high levels of code reviewing, and a review count higher than 73 makes us consider the project doing a very high level of code reviewing. Similar, for comments less than 29, between 29 and 38, between 38 and 56.3, and greater than 56.3 we have low, moderate, high, and very high level of code reviewing. The mentioned thresholds corresponding to percentiles are shown in Table 2.9.

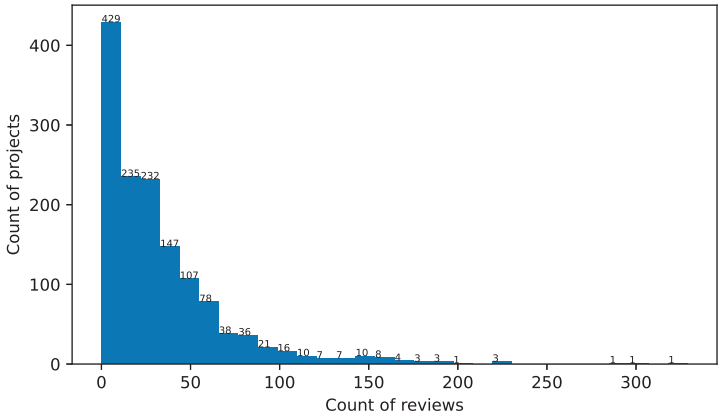

Figure 2.8: Distribution of count of reviews in projects' last 20 merged PRs.

Metric	Low Level	Moderate Level	High Level	Very High Level
Reviews Count	38	50	73	>73
Comments Count	29	38	56.3	>56.3

Table 2.9: Thresholds for code review metrics.

Observation 5

Projects having more than 50 reviews or 38 comments in their last 20 merged pull requests are among the projects doing more intense code reviewing. Similarly, projects having less than 38 reviews or 29 comments can be considered as projects doing relatively less code reviews.

Code Review in Mature Projects

We observe that 48% of mature projects have either a high or very high level of code review intensity based on the thresholds measured in Table 2.9. Around 26% of them fall in the category of *low* code review for both reviews and comments count. If we do a comparison with the code reviewing intensity in all the projects, we see that 30% of projects having high or very high level of code reviews and 54.5% of them falling below the low level. This shows mature projects tend to invest more in their code reviews as a quality assurance practice.

Observation 5*

Around 48% of mature projects have either a high or very high level of code review intensity. Circa 26% of the projects have a low level of code review intensity.

2.4.5 Testing

Software testing is a practice to improve the quality of software by detecting faults in the source code [39]. The goal of this section is to see how well projects are doing in terms of software testing. In contrast to code reviewing, we have established metrics for measuring the quality of tests, namely code coverage metrics [95]. In addition, to gauge how well software projects are tested in terms of code coverage, we can rely on the existing classification scheme for unit test coverage proposed by Heitlager *et al.* [42]. This classification ranks projects in 5 different levels according to their unit test coverage and makes it easier to reason about testing (see Table 2.11). In our investigation, we rely on *branch coverage* as calculated by the JaCoCo tool.[16]

As mentioned in Section 2.3.2 to see the state of testing in our projects, we selected 599 projects that (1) could be built successfully in the first step of our study and (2) had conventional paths for their source code and test code (at least in one of their subprojects / modules). Having conventional paths for source code and tests means that we could detect their path in a project. We also repeated this for all subprojects / modules inside a project before applying JaCoCo. If source code and tests paths were detected, then we would apply JaCoCo plugin and build them to generate coverage results. The strict interpretation of the necessity to have conventional paths for the tests does imply that there might be projects that do have high quality tests, but that are not part of our analysis.

For the 599 projects that we applied JaCoCo to, we obtain JaCoCo test coverage results for 310 of them (51.75%). From these 310 projects, we observe complete coverage results for 172 (28.7%) projects, i.e., we did manage to get coverage results for all of their modules. For another 138 (23%) projects, we could only collect partial coverage results, i.e., we have coverage results for some of the modules. We did not manage to obtain coverage results for the remaining 289 (48.3%) projects. An overview of the aforementioned process can be witnessed in the bottom part of Figure 2.9.

Projects with no coverage results. To investigate the reasons of 289 projects having no JaCoCo output, we randomly selected 50 of them and manually checked their log files. These projects had at least one module with code *and* tests detected, but we could not see any coverage results after applying the JaCoCo plugin. The most common reasons for this were: (1) JaCoCo failing or skipping its execution due to some (mis)configuration in the project's build setting, or conflicting with other settings of project, and (2) build failure. See Table 2.10 for a global overview of reasons. Moreover, for 12 projects the build logs were very sparse, and we could thus not establish the reason of having no coverage result.

Projects with partial coverage results. For 138 projects we have obtained partial coverage results: 113 of them had at least one module with no test. In terms of why not all modules have JaCoCo output, we observe similar reasons as to our prior case of projects

[16] `https://www.jacoco.org`, last visited May 20th, 2022.

Reason	#	%
JaCoCo skipped/failed	17	34%
Build failure	16	32%
Parsing error	3	6%
Test failure	2	4%

Table 2.10: Reasons of having no JaCoCo coverage result.

Rank	Code Coverage	# of Projects	% of Projects
Very High (++)	95-100%	9	2.9%
High (+)	80-95%	31	10%
Moderate (o)	60-80%	58	18.7%
Low (−)	20-60%	129	41.6%
Very Low (−−)	0-20%	83	26.8%

Table 2.11: Ranking projects based on their code coverage using the classification scheme for unit test coverage proposed by Heitlager *et al.* [42]

without any coverage result. If we compare the coverage levels of projects with partial and complete coverage results in Figure 2.10, we observe a similar distribution.

We also observe that from the total of 599 projects 389 (65%) have at least one module with no tests (this is minus the projects without any tests since we removed them in the beginning).

Table 2.11 provides an overview of how well projects are tested. Relying on the classification scheme from Heitlager *et al.* [42], we observe that 2.9% of projects fall in the category "very high" coverage, while a majority of projects are placed in the category "low". In fact, combinig the categories "low" and "very low", we see that 68.4% of projects fall into these two categories, giving an indication that not all projects are maximising their testing.

Observation 6

68.4% of projects among the ones we could calculate code coverage for, are doing very low or low testing. Also, only ~13% of them are doing very high or high testing. Moreover, 65% of projects doing testing, do not do it for all modules of their project.

Testing in Mature Projects

We seek to understand whether and how following *testing* as a quality assurance practice differs in mature projects compared to other projects in our dataset. Six projects out of the total 50 mature ones have code coverage results (6 out of 310 projects with JaCoCo results in Figure 2.9 are mature). Since this is a very low number to be representative of all mature projects, we manually looked all of 50 repositories to find their code coverage information.

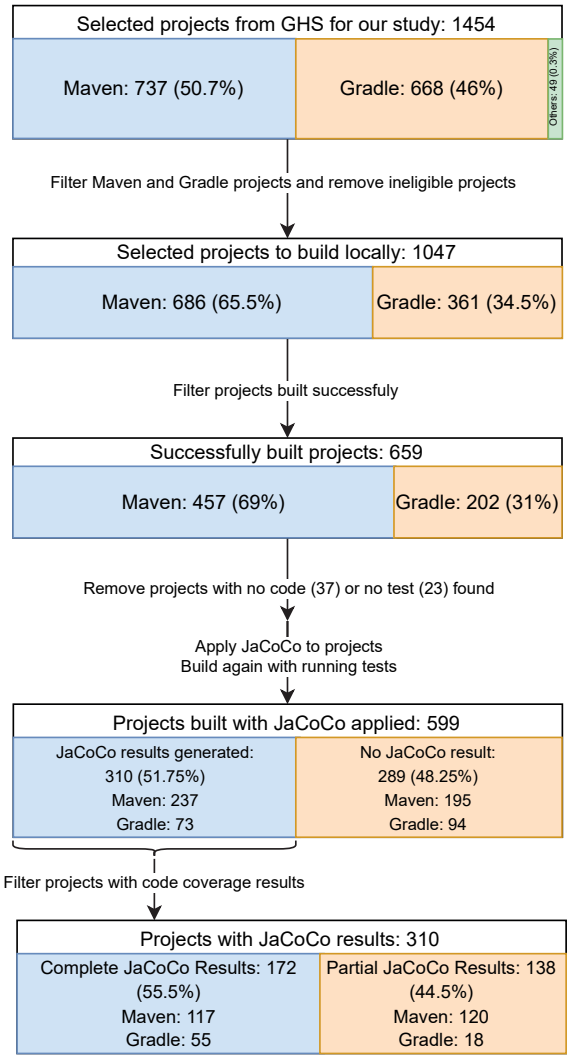

Figure 2.9: Overview of studied projects in building and testing.

To this end, the first author checked repositories' *README* files for a coverage badge[17], scanned through pull request comments, and looked for CI status checks on a project's GitHub page for coverage information. Overall, we found code coverage information for 12 projects using this approach [18]. If we combine the information of these 12 projects with the JaCoCo results, we now have coverage information for 14 projects, i.e., for 4 projects

[17]For more information on badges on GitHub, see: `https://github.com/dwyl/repo-badges`

[18]Except one project using Jenkins service to measure branch coverage, other projects use Codecov service to measure their code coverage based on Codecov coverage metric: https://docs.codecov.com/docs/about-code-coverage, last visited September 7th, 2022.

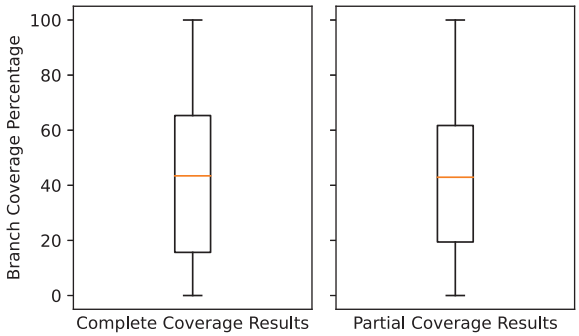

Figure 2.10: Branch coverage of projects with complete and partial coverage result.

we have both results of JaCoCo, and code coverage information from the GitHub page. We picked the latter one for further analysis. Also, we checked to see if they have test suites in their repositories, and for all the mature projects we found test files indicating they have developed a test suite for their project.

We observe a mean of 65.6% code coverage among mature projects, with a median value of 63.2%. This falls into the moderate level of testing, which is higher when we compare it to the mean of all other projects in our dataset, as they reach a low level of testing.

Observation 6*

All mature projects have test suites in their repositories. Also, on average they have 65.6% code coverage (with median of 63.2%) among the 14 projects that we could establish code coverage for.

2.4.6 Quality Assurance Practices Put Together

In this section we set out to answer **RQ1.2**: *Which quality assurance approaches are being used in conjunction?* In order to answer this question, we use statistical tests to determine which quality assurance approaches that we previously discussed in isolation in Sections 2.4.1 through 2.4.5 are used in conjunction in open source projects. More specifically, we conduct a statistical analysis to calculate the correlation between each of the practices using the results from previous sections: code coverage of 310 projects, code reviewing information of 1398 projects, ASAT usage and CI usage of all 1,454 projects. This analysis does raise the issue of data imbalance, as we have less projects for which we have code coverage data (310) than we have projects for which we have code reviewing information (1398). To address this issue, we have performed the statistical test on projects that are in the intersection for two quality assurance practices, i.e., those projects for which we have data of both.

The correlation analysis enables us to better understand whether projects that use a quality assurance practice (intensively), are also more inclined to employ another quality assurance approach (intensively). Of importance to note here is that for two quality

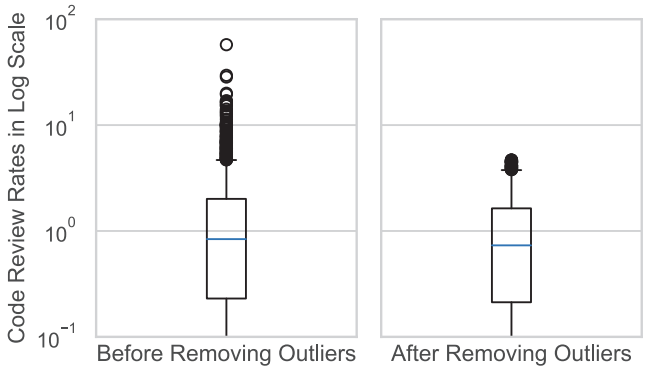

Figure 2.11: Code review rates before and after removing outliers.

assurance practices, namely ASAT usage and CI usage, our data is categorical, i.e., a project *is* or *is not* using them. The other two practices, namely Testing and Code Reviews, are presented as continuous variables based on respectively the level of code coverage reached and the counts of comments/reviews that we observe. However, while for software testing we can make use of an established code coverage metric (branch coverage), for code reviewing we had to define a new metric to express the intensity of the code reviewing process (Equation 2.1 explained in Section 2.3.2).

In terms of the statistical tests, we use Cramer's V to measure correlation among categorical variables. Before applying Cramer's test, we first tested the independence of the variables using the Chi-square test. For the continuous variables vs. continuous variables and vs. categorical variables, we use respectively the Pearson and the Point Biserial correlation coefficient. Because we conduct several tests over the same population of projects, we further apply a family-wise error correction to limit the inflation of false positives that multiple testing induces. We treat each research question as a separate family: the two tests relating CI to buildability in Section 2.4.3 form one family, and the nine tests in Table 2.12 another. Within each family we apply the Holm-Bonferroni correction [93]. Table 2.12 reports the results of the correlation analysis, including both the raw and adjusted p-values.

Table 2.12 shows the results of the statistical tests. The results that are statistically significant are highlighted in light-grey. For our code review metric, since there were quite a lot of outliers (Figure 2.11 shows the distribution of code review metric values before and after removing outliers), we used the IQR method to identify and then remove outliers [96], which resulted in removing 106 projects (from a total of 1398 projects with code review rates). Therefore, there are two values for code review correlation analysis. Moreover, the same method was then used to remove the outliers of the code review metric in Figure 2.12. In contrast, the range of values for testing are fixed (0 to 100), and they are binary for ASAT/CI usage, so no outlier exists for other variables.

	Code Review	Code Review Without Outliers	Testing	ASAT Usage
Testing	Corr = 0.29 p-value <0.0001 p_{adj} <0.0001	Corr = 0.15 p-value = 0.011 p_{adj} = 0.056		
ASAT Usage	Corr = 0.019 p-value = 0.48 p_{adj} = 1.0	‘Corr = 0.12 p-value <0.0001 p_{adj} = 0.00014	Corr = 0.0157 p-value = 0.79 p_{adj} = 1.0	
CI Usage	Corr = 0.0317 p-value = 0.24 p_{adj} = 0.94	Corr = 0.10 p-value = 0.00024 p_{adj} = 0.0014	Corr = 0.0231 p-value = 0.69 p_{adj} = 1.0	Cramer’s V = 0.10 p-value <0.0001 p_{adj} = 0.00070

Table 2.12: Correlation analysis of quality assurance practices. p_{adj} denotes the p-value after Holm-Bonferroni correction across the nine tests in this table. Cells highlighted in light-gray are correlations that remain significant at $\alpha = 0.05$ after correction.

Code reviewing and testing show a positive correlation with small effect size $r = 0.29$[19] (also $r = 0.15$ when we remove outliers from code review rate, although this sensitivity check no longer reaches significance once corrected for multiple comparisons, adjusted p-value = 0.056). To better understand the small effect size, consider Figure 2.12. For both the projects that use ASATs and that do not use ASATs, we observe a concentration of projects that have a code review rate between 0 and 0.5 (the X-axis). Yet, that group of projects in that specific (relatively low) code review rate range show a widely dispersed code coverage level, going from 0% all the way to 100% branch coverage. This wide dispersal on the testing axis and clustering on the code reviewing axis (Y-axis) explains the small effect size. It also gives us the insight that projects that heavily invest in high levels of code coverage do not necessarily invest as strongly in code reviewing.

Code reviewing and CI usage. We observe a small positive correlation between code reviewing (without outliers) and CI usage $r = 0.10$. When we look at Figure 2.12, we need to consider both the left and right graphs and look at the X-axis for the code review rate and the spread of blue dots and orange crosses, signalling respectively the *use* and *non-use* of CI. For both the left and right graphs, we see an overall scattering of projects that use CI over the entire range of the code review rate. For the more dense code review rate interval [0,1.5], we visually observe a mix of projects that use or do not use CI. This scattering helps explain the small correlation that we obtained.

Code reviewing and ASAT usage. We observe a small positive correlation between code reviewing (without outliers) and ASAT usage $r = 0.12$. When we look at Figure 2.12, we need to consider both the left (projects not using any ASATs) and right graphs (projects using ASATs) and look at the X-axis for the code review rate. We see an overall scattering of projects in the entire range of the code review rate. However, the density of projects with a code review rate between 0 and 2 is relatively higher in the left graph, On the other

[19] For interpreting the strength of Pearson’s correlation, we use $0 \leq r < 0.3$ for *small*, $0.3 \leq r < 0.5$ for medium, and $0.5 \leq r < 1.0$ for large effect size [97].

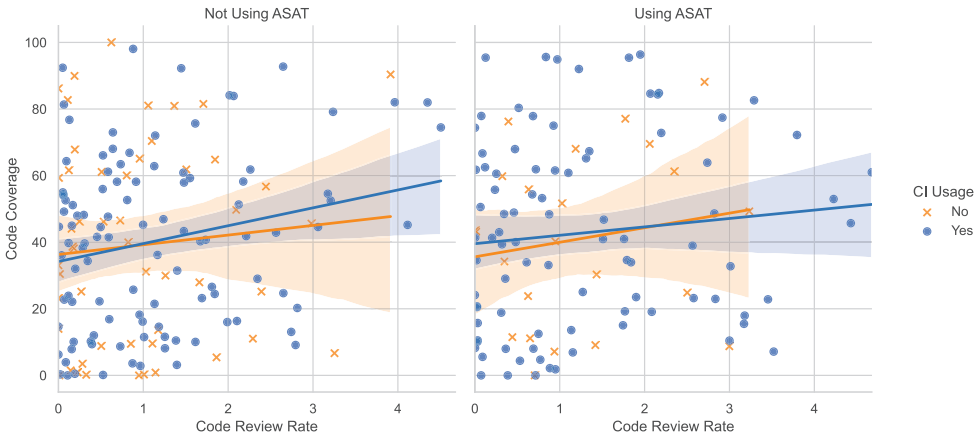

Figure 2.12: Relation between Testing and Code Reviewing with respect to usage of CI and ASAT. The outliers of code reviewing metric are excluded.

hand, we see more projects having a code review rate higher than 3 in the right graph. This density of projects in different ranges explains the small correlation that we have observed.

CI usage and ASAT usage exhibit a weak, yet significant correlation $V = 0.10$[20]. When we turn our attention to Figure 2.12, we need to compare the distribution of blue and orange (respectively CI usage and no CI usage) dots/crosses in the left (not using an ASAT) and right (using an ASAT) graphs. Both left and right, we observe a multitude of projects that do and that do not use CI, with visually more orange crosses (for non CI usage) in the left most graph, signalling that no ASAT usage is more common amongst projects that do not use CI. To also see this numerically we divide the count of dots in the right graph by the total number of projects using an ASAT, and also divide the count of dots in the left graph by the total number of projects not using an ASAT. The results are respectively 0.79 and 0.67, confirming our earlier intuition that we have a higher percentage of projects that use CI among projects using ASATs, compared to projects not using any ASAT.

Summarising the other relations from Table 2.12, we observe no other statistically significant correlations between the other quality assurance practices.

Observation 7

Having high code coverage weakly correlates with doing more code reviewing in projects. We also observe weak correlations between (1) the usage of ASATs and CI usage, (2) the code review rate and CI usage, and (3) the code review rate and ASAT usage.

[20]The effect size interpretation for Cramer's V is: $0 \leq V \leq 0.2$ is weak, $0.2 < V \leq 0.6$ is moderate, $V > 0.6$ is strong

2.5 Discussion

2.5.1 Revisiting the Research Questions

In this section, we revisit the research questions from Section 2.1. In order to answer the over-arching research question

> **RQ1** *What is current state-of-the-practice in quality assurance in open source software development?*

We first concentrate on providing a global overview of some of the most popular quality assurance practices in use, thereby answering **RQ1.1**

> **RQ1.1** What is the prevalence of quality assurance approaches like software testing, modern code review, automated static analysis, and buildability?

As we mentioned before, RQ1.1 is aimed squarely at understanding the current state-of-the-practice with regard to the quality assurance practices of which we want to understand whether they are used in conjunction. We will now discuss our findings for each of the quality assurance practices in isolation.

Testing. From the original set of 1,454 GitHub projects, we have been able to establish code coverage for 310 projects (21.3%). Our results show that the median of their branch coverage is ~43%. Moreover, using Heitlager's classification scheme [42] of test coverage for projects, we establish that ~68% of them do *very low*, or *low* levels of testing. On the other hand, ~13% of projects reach *high*, or *very high* levels of code coverage.

The reason why we could only establish code coverage for 21.3% of the projects in our original dataset can be seen in Figure 2.9 and can be summarized as a race of attrition because of projects' inherent attributes that make them ineligible for our study (28%), issues of buildability (26.7%), non-presence of tests and/or source code (4.1%), and our inability to successfully establish code coverage results (19.8%).

Code Review. For 1398 projects in our dataset we looked at the count of comments, and the count of reviews in their last 20 merged PRs. We have created a code review intensity metric, that expresses the amount of code reviewing normalised by lines of code changed in the PR. Subsequently, using Alves *et al.*'s calibration method [94] we derived different categories of code reviewing intensity. We observe that projects having more than 50 reviews or 38 comments over their last 20 merged pull requests are among the projects doing more intense code reviewing. In contrast, projects having less than 38 reviews or 29 comments can be considered as projects doing relatively less code reviewing.

ASAT Usage. To study the usage of ASATs in GitHub projects, we make the assumption that the presence of an ASAT configuration file is an indication of actual ASAT usage. Under this assumption, we have observed that 38% of the projects under observation use at least one ASAT. However, the presence of a configuration file does not causally imply the actual usage of these ASATs.

Local Buildability. We set out to investigate whether the GitHub projects under study can be built locally out of the box. Our results indicate that ~63% of projects build successfully locally.

Continuous Integration. We have used a new approach of studying usage of CI on GitHub, namely using GitHub's checks and statuses. This approach enables us to have a wide range of CI platforms in our study compared to previous studies that mainly focused on a single CI provider, e.g., [21]. We find evidence that ~63% of overall 1,454 projects use CI with ~76% of them having a successful state. This also indicates that for 24% of the projects, the final commit that is part of our study led to a failing CI build. Moreover, we observe different strategies in making use of GitHub features (GitHub Actions, and Statuses) to implement CI status checks among projects (e.g., having a few vs. many CI status checks with many different CI platforms or services). In future work, we intend to study the reasons for using multiple CI status checks and platforms.

RQ1.1 Summary

We found that the majority of projects exhibited low testing levels. The intensity of code reviews varied, with some projects engaging in more intensive reviews. About 38% of the projects had configured automated static analysis tools. Approximately 63% of the projects had the capability to be built locally, and 63% of them used Continuous Integration, with 76% having a successful state. Additionally, we observed various strategies and the use of different CI platforms for implementing CI status checks in the projects.

Now that we have an understanding of how these quality assurance practices are used in isolation, we turn our attention to:

RQ1.2 Which quality assurance approaches are being used in conjunction?

Code Review and Testing. We found statistically significant evidence that higher levels of branch coverage correlate with higher levels of the code review intensity metric with small effect size. This indicates that projects that show a higher level of branch coverage tend to have more code reviewing performed.

Code Review and ASAT/CI usage. Using our defined categories of code reviewing intensity, if we look at projects with a very high level of code reviewing, e.g., more than 73 reviews or 56 comments in their last 20 merged PRs (239 projects), for 49% of them we find evidence of ASAT usage, and for 65% we see evidence of CI usage. On the other hand if we look at projects with a low level of code reviewing, e.g., less than 38 reviews and less than 29 reviews in their last 20 merged PRs (781 projects), for 32% of them we find evidence of ASAT usage, while for 61% we see evidence of CI usage.

This observation shows that projects with a higher code review intensity tend to use ASAT and CI more when comparing to the ones with lower code review intensity. We also saw statistically significant correlation between our code reviewing intensity metric and ASAT/CI usage when outliers are removed, which strengthens this observation.

ASAT usage and CI Usage. Finding evidence of ASAT usage in projects does not imply that these ASATs are actually used and integrated in the CI workflow. However, we did observe a statistically significant, yet weak correlation between their usage, which potentially indicate that projects use both at the same time.

RQ1.2 Summary

We found statistically significant correlations with small effect size among Code Review and Testing, Code Review and ASAT/CI usage, and ASAT usage and CI usage, which demonstrate their usage in conjunction.

RQ1.3: How do mature projects follow quality assurance practices?

To summarize our findings with regard to the quality assurance practices in more mature projects, compared to our overall dataset, we can state that:

- **local buildability** is almost the same, which also indicates that keeping the latest version buildable is also a challenge for mature Java projects,
- **ASAT usage** is 22% higher in the more mature projects,
- the **CI usage** ratio is similar over both sets, but CI states are less successful in the more mature projects,
- the **code review** intensity is either high or very high for around half of the more mature projects, while over the entire dataset the projects with the 20% highest code review intensity are in these categories (see Table 2.9 for thresholds). This is an indication that more mature projects typically do code reviews at a higher intensity,
- finally, test suites of more mature projects seemingly have higher **code coverage**. Moreover, based on the existence of unit tests in *all* mature project repositories, we assume they all do testing.

RQ1.3 Summary

We observed that mature projects demonstrate a higher level of adherence to quality assurance practices, or at least comparable to other projects in our dataset. However, they exhibit less successful CI states.

2.5.2 Difficulties encountered during our investigation

We now give an overview of the challenges and obstacles that we have encountered during our study. We hypothesize that understanding and potentially alleviating these challenges in the future can be useful for researchers, and perhaps also practitioners.

Difficulty of creating an environment for local build success. We could not build ~ 47% of projects out of the box. Compilation, dependency, and configuration errors are the top three reasons of the build failures. Also, other studies have reported that building Java projects is not always straightforward. Hassan *et al.* could not build 46% of the projects

in their study [64]. Sulir *et al.* have respectively observed a 38% and 59% build failure percentage in their local build results dataset in their 2017 and 2020 studies [89, 98].

This number of build failures shows that it is non-trivial to create an easy-to-build environment for Java projects. This has far-reaching consequences for both **practitioners** and **researchers**. For **practitioners**, this means that developers that want to contribute to an open source project might find it difficult to build the project on their development machine. On the other hand, **researchers** might be hindered to conduct their studies on a large variety of projects, or on multiple versions of a project. Several studies have investigated the causes of build failures [25, 68], while other studies have tried to devise approaches to tackle the underlying issues [68, 99–101]. In an ideal world, creating an environment (e.g., a Docker container) for building a large number of projects would benefit **researchers** to conduct studies that need to build projects as a prerequisite (e.g., dynamic analysis, measuring code coverage, etc.).

Interestingly, our results indicate that if project maintainers want more confidence about their projects' buildability on local machines, they should consider doing continuous integration, because CI increases the likelihood of a successful local build (Cramer's V $\simeq$ 0.08). And also consider having a successful CI build for the latest release of the project. Because, CI builds match with local build outcomes in 64% of projects, and there is a statistically significant correlation between them (Cramer's V $\simeq$ 0.10).

Measuring code coverage. Measuring code coverage of many projects automatically is a challenge for **researchers**. One prerequisite of measuring code coverage of a Java project is building it, and we already discussed that creating an environment for building a large number of projects successfully is a challenge. The next step in the measurement of code coverage is to use a code coverage measurement tool. Our observations are that, first, most projects do not have their code coverage details available in their repository. So, in most cases researchers need to measure projects' code coverage themselves. Second, automatically generating code coverage information for a large number of projects is a cumbersome task. This is because of the difficulty of integrating a code coverage tool like JaCoCo to generate code coverage results, and also successfully building the project with running all tests. Particularly, this is why we were unable to collect coverage measurements for ~48% of projects when trying to measure their code coverage. Third, many projects have several modules (subprojects), and they do not have unit tests for all of them. This issue makes it difficult to have an idea of overall code coverage since some parts of the software project miss tests. This third issue in particular, is also an issue for **practitioners**, as it leaves them blind with regard to the level of test coverage of specific submodules.

Measuring the quality of code review. In this paper we suggested a code review metric to have an idea of the intensity of reviews taking place during pull requests' discussions. Then we looked at the correlation between testing, ASAT usage, and CI usage with code review intensity. However, the intensity of code review is different from the quality of the code review. Despite the fact that code review quality has been studied in many studies, there is no obvious way to quantitatively measure it [46, 101, 102]. Having a measurement of the quality of the code review would help **researchers** when comparing projects' code review and when studying its correlation with other quality assurance practices. Also, with a code quality measurement **practitioners** have a concrete idea of the state of the

quality of the code reviews taking place in their project, and it enables them to improve their code review practices.

ASATs actual usage As we pointed out before (see Section 2.5.1), the presence of a configuration file does not causally imply the *actual usage* of these ASATs. Vassallo *et al.* found that developers use ASATs in different contexts: local programming, code review, and CI [103]. So, to have a notion of ASATs actual usage we need to look at them in all three different contexts, which is a challenge for researchers, keeping in mind that different projects use different CI platforms. We have looked at how other studies approach the study of actual usage of ASATs: Beller *et al.* investigated the use of ASATs in 122 projects by manual analysis of configuration files and then surveying the contributors of those projects, asking about the actual usage of ASATs [18]. In another work Zampetti *et al.* studied the actual use of ASATs (Automated Static Analysis Tools) in the CI environment [104]. In their study, they analyzed Travis CI builds and jobs of 20 projects. Amankwah *et al.* investigated the precision by which ASATs can find bugs in Java source code and found precision values between 83% to 90.7% based on the studied datasets and using 8 different ASATs [105].

For **researchers**, observing and understanding the actual usage of ASATs at scale remains a challenge. A key challenge is that ASATs can be used in multiple contexts (per the observation of Vassallo *et al.* [103]), thus requiring multiple observation points. Another challenge is that in order to obtain meaningful insights, the dataset would need to be sufficiently large, implying that we need an automated observation tool to perform a longitudinal analysis on multiple projects.

GitHub Actions best practices To study the use of CI, we put our focus on the status checks of a project's latest commit. We found that projects have different approach when configuring their CI using GitHub features: GitHub Actions (checks run using GitHub's apps) & Commit Statuses (statuses marked by external services). There are projects with more than 100 status checks, and many others with only 1 check / status. Also, in some cases, many GitHub apps are used in parallel to run checks. Here we highlight the challenge for **practitioners**: configuring their CI using checks (GHA) and statuses. Including choosing the right (combination of) GitHub app(s) for these checks.

Kinsman *et al.* have shown that developers have a positive perception of GHA [87], and in a related work about CI services on GitHub, Golzadeh *et al.* found that GHA has become the dominant CI service 18 months after its introduction [106], which strengthens the need to address the mentioned challenges by offering best practices to use GHA.

In another work, Chen *et al.* took a different path to study the usage of GHA focusing on the projects' GHA YAML file (under the workflow path of projects adopting GHA). They similarly found a median of 3 for steps (median of 3 for status checks in our study) used in jobs (with average of 4.7 vs. average of 7.35 status checks in our study), and highlighted the challenge for less experienced developers to configure a GHA workflow when facing with workflows, jobs, and steps with many alternatives.

2.5.3 The Bigger Context of Quality Assurance Practices

In Section 2.5.1 we have revisited research questions **RQ1.1**, **RQ1.2** and **RQ1.3** and gave concrete answer to them. However, we have not answered the over-arching RQ1 yet. In order to do so, we need to look at the bigger context.

RQ1 *What is current state-of-the-practice in quality assurance in open source software development?*

First of all, we have observed a very diverse level of adoption, or intensity of adoption, of the individual quality assurance practices among our set of 1,454 open source software projects on GitHub. We have observed projects that do not invest in any of the practices, but we have also seen projects investing in all, or at least most practices with relatively high intensity. Also, when we look at mature projects we see that they generally tend to invest more in their quality assurance practices.

When it comes to using quality assurance practices in conjunction, we have observed weak correlation among some quality assurance practices in projects. But generally speaking, projects do not follow all quality assurance practices together with high intensity. This raises interesting questions for follow-up research that is more qualitative in nature, because some of the studied quality assurance practices are said to be complementary, using them in conjunction would strengthen the overall quality perspective, and might also reduce quality assurance efforts (e.g., an ASAT might pick up code review comments, which would ease the burden of the reviewer [31]).

RQ1 Summary

We observed diverse adoption levels of individual quality assurance practices, ranging from no investment to high-intensity adoption. Mature projects tend to invest more in their quality assurance practices. While some weak correlations among these practices exist, most projects do not simultaneously employ all quality assurance practices with high intensity.

2.6 Related Work

In this section we compare the results of our study with other related studies.

2.6.1 Quality Assurance Practices in Isolation

Most of the related work has studied each of the quality assurance practices in isolation. Here we look how our results for the usage of each of these quality assurance practices in isolation compare to other related studies.

Buildability. Hassan *et al.* could build 54% of the projects in their dataset using the default build commands of build tools. This ratio is 9% lower than our 63% of projects that build successfully out of the box [64]. While their methodology for establishing buildability is very similar to ours, their dataset comprises fewer projects (200 vs. 1456). Of importance to note is that their dataset also comprises projects that use the Ant build system. Moreover, they have discussed that 52 of the build failures can be resolved automatically. Following this work, Hassan and Wang created HireBuild as an automatic approach to fix reproducible build failures (which is a history-driven approach, thus could not be used in our study to address build failure issues) [65]. Both of the aforementioned studies highlight the challenge of facing build failures. Sulir *et al.* have respectively observed a 38% and 59% build failure percentage in their local build results dataset in their 2017 and 2020 studies [89, 98]. Similar

to our observations, both of their studies report *'dependencies'* and *'Java compilation'* as two of the top categories of errors leading to build failure of the projects.

Testing. Our goal of measuring code coverage of projects was to see how testing is being done in conjunction with other quality assurance practices. To the best of our knowledge no other studies have done such large-scale study of code coverage of open-source projects' unit tests for this particular goal. Other studies have looked at code coverage of projects to study its evolution [39], or its relation to software quality [41]. Hilton *et al.* have taken a similar approach collecting code coverage information for 29 projects using JaCoCo [40]. However, their study was aimed at studying the coverage evolution, and thus they focused on obtaining 250 successful builds per project for a limited set of projects, instead of single successful builds of many projects in our study. Overall, for the 29 Java projects that they have considered, six of them had <60% coverage and only one had <20%. The average code coverage percentage for all projects in their study was 76% [39]. Kochhar *et al.* have used Sonar to measure code coverage of 100 open-source Java projects. 8% of the projects in their dataset had higher than 75% coverage. Also, 69% of them had a coverage of less than 50%. The results of the coverage of their projects are similar to ours, however they used a different coverage metric which is a combination of branch and statement coverage. One of the main reasons of the difference in results of our study comparing to others is that we have used a relatively larger set of projects. Addressing the difficulty of obtaining code coverage measures for researchers, would open up the way to more studies of the state of the practice in testing.

Continuous Integration. One of the main differences between our CI usage study and other related works lies in our approach to detect CI usage. Many other studies have used either the TravisTorrent dataset [66], or the Travis-CI API for this matter [20, 24, 28, 29, 32, 107, 108]. Other studies have looked at Jenkins [20, 108], CircleCI [24, 90, 108], SonarCloud [90], AppVeyor [24], or Werker [24] as the CI services configured with GitHub projects in their studies. However, the introduction of GHA aligns with a decreasing growth rate of CI usage for Travis and CircleCI, two of the most popular CI platforms before the introduction of GHA [106]. Considering this fact, we have used a new approach to detect CI usage in GitHub projects regardless of the CI platform that they use. This approach gave us the ability to study more projects.

Hilton *et al.* have observed a 40% CI usage in their corpus of ~34K projects. When they only considered the most popular projects in their corpus, they observe a 70% usage ratio [24]. In order to make a fair comparison, we only considered the Java projects in Hilton *et al.*'s dataset when comparing to our results: they had a 35% CI usage among their 3371 Java projects, while we have observed a 63% usage rate some five years later. This point also underlines the need for more research on GHA adoption among projects as the most popular CI platform among open-source projects to highlight best practices as mentioned in Section 2.5.1.

Code Review. McIntosh *et al.* have found that reviews without discussion are associated with higher post-release defect counts [69]. Accordingly, we use count of comments and reviews in our study as a proxy for code review intensity. They have also studied the effect of code review coverage metrics on software quality, however this contributes less than code review participation metrics to software quality defect models. Bosu *et al.* have identified a set of factors affecting the usefulness of code reviews [109]. They recommend

submitting smaller and incremental changes. Therefore, using this insight, we normalized count of comments and reviews in code review discussions by the number of lines of code changed in each pull request. To the best of our knowledge no other study has suggested any quantitative code review intensity / quality metric for projects on GitHub.

ASAT Usage. Beller *et al.* did an analysis of the state of ASAT usage in open source software. They have found that 60% of the popular OSS projects have automated static analysis configured, however they usually only use one ASAT and do not integrate it into their workflows [18]. Comparing to their results, our study shows 38% of ASAT usage among projects which increases to 70% by considering only the mature ones. Checking whether ASATs are integrated into the workflows of projects is out of the scope of our study. But since we also see an increase in CI usage among mature projects, integrating ASATs in CI workflows can be a factor in this observation. Other related studies have looked at the reasons why developers use ASATs [110], how they engage with them in different contexts [19], and how developers employ them in their CI pipelines [104]. We will discuss more about their results in the next part when discussing related work of different quality assurance practices being used in conjunction.

2.6.2 Quality Assurance Practices in Conjunction

To the best of our knowledge our work is the first study into how a set of quality assurance practices are being used in conjunction. Other related work has looked at the influence of one practice on another by focusing on studying the relation between them. Here we look at the results of these studies and compare them to ours.

Literature on ASATs suggests to configure ASATs in development workflows, e.g., CI pipelines [18]. In particular, Zampetti *et al.* have studied how 20 popular Java open source projects use ASATs in their CI. Specifically, they have investigated which ASAT tools are being used and how they are configured for CI, and what types of issues make the build fail or raise warnings. They have observed that CI builds typically break due to ASATs detecting non-adherence to coding standards [104]. In our analysis, we have observed a statistically significant, yet weak correlation between CI usage and ASAT usage, indicating that ASATs are indeed potentially configured in CI pipelines. Also, the ratios of CI and ASAT usage both increase when we filter for mature projects.

In another related work, Zampetti *et al.* studied the interplay between pull request reviews and CI builds. They found that pull request discussions mainly focus on testing, static analysis problems, and also about CI configurations [30]. This seemingly confirms that some quality assurance approaches are used in conjunction, in particular code reviewing and CI builds.

Cassee *et al.* found CI to be a silent helper of code reviews by saving time. They specifically found that the number of comments in code reviews decreases after adopting CI, without any change in the number of changes per code review [28]. Our study has indicated a statistically significant, yet weak correlation between CI usage and intensity of code reviews, which means that projects with high code review intensity tend to use CI more when comparing to the ones with lower code review intensity. So from Cassee *et al.* we learn that using CI can be a time saver, while our observations point to projects with high code review intensity already to be using CI.

In another similar study, Rahman *et al.* found that automated builds very likely affect

code review participation. They stated that automated builds along with their outcomes might trigger further code reviews [48]. This can also be a reason why we see a statistically significant correlation between code review intensity and CI usage. We encourage future research to qualitatively study the relation between CI usage and code review intensity.

Wang *et al.* studied the combined use of testing and CI in open source projects by focusing on how test automation maturity can improve product quality. They found product quality improvement and release cycle acceleration as the potential benefits of test automation maturity [111]. In support of our own investigation, the study by Wang *et al.* confirms the importance of using specific quality assurance practices in conjunction.

In a survey study [21], Khatami and Zaidman evaluated the awareness of open-source software maintainers and contributors regarding quality assurance practices within their projects [112]. Through a qualitative study they observed a prevalent high-level awareness among the study participants concerning various quality assurance approaches. However, their research also unveiled areas of notably low awareness, including aspects such as knowledge gaps related to test coverage information, the advantages of code review in projects, the usage of bots, and the when and how of checking CI results in projects. This study complements our quantitative approach by shedding light on the qualitative dimension, specifically highlighting the awareness gaps surrounding quality assurance in open-source projects.

2.7 Implications and Future Work

We envision that within the software engineering community there are three groups that can benefit from the results of our study, namely researchers, educators, and practitioners.

Researchers. Our investigation has shown that among most quality assurance practices no or only a weak correlation can be observed. This calls for a deeper understanding of the complementarity of the approaches. Simultaneously, we need to better comprehend why practitioners choose between practices, and not do multiple of them at a similar level of intensity. We hypothesize that practitioners need a better grasp of how effective each of the quality assurance practices is to reach a certain level of quality. In this area, we also refer to the work of Cassee *et al.* which shows that CI can reduce the effort required from code reviews [28]. We make an explicit call to arms, to further investigate the effectiveness and efficiency of using quality assurance practices in conjunction.

Another important and timely step for researchers to make is to establish and measure the quality and intensity of code reviewing. Additionally, with the growth in popularity of GitHub Actions, a deeper investigation into how this platform is used and should be used is also an urgent concern for future research. Furthermore, we propose to replicate our study for other programming languages, e.g., Python, JavaScript, to get a better understanding of the state-of-practice in other communities. We also envision that it can be interesting to investigate how practitioners' behaviour in applying quality assurance practices changes, when they are better informed in terms of the effectiveness of the quality assurance approaches, both when used in isolation and in conjunction. Finally, we consider that another interesting path for future research is to investigate the challenges that open-source developers encounter when trying to contribute to open-source software projects.

[21] Chapter 3 of this dissertation.

Educators. Based on our observations, we would like to emphasize the importance of teaching about all of these quality assurance practices, thereby also highlighting their complementarity so that these future practitioners can make well thought through trade-offs in terms of quality assurance. Of importance to underline towards students is our observation that mature projects tend to combine quality assurance practices more intensely. Another important element to highlight here is that students should be made aware of the importance of leaving the project in a buildable state, as to enable newcomers to easily contribute to the project.

Practitioners. Our study underscores the importance of raising awareness among software practitioners about the benefits of different quality assurance practices and emphasizes the potential for these practices to complement each other. Although we acknowledge the challenges associated with obtaining information on the adoption of different quality assurance practices (e.g., measuring code coverage), practitioners, being intimately familiar with their projects, may find it more feasible to overcome these challenges. By assessing the extent to which different quality assurance practices are implemented in their projects, practitioners can address quality issues (e.g., maintaining a buildable state) and gain valuable insights into the overall quality assurance status. Additionally, comparing this assessment with the quality assurance state of more mature projects, which tend to exhibit higher adoption intensity of various quality assurance practices, can serve as motivation for practitioners to strive for similar levels of adoption in their own projects. Furthermore, we encourage future research to explore practitioners' level of awareness regarding quality assurance in their projects and investigate ways to enhance their awareness.

2.8 Threats to Validity

External validity. We have used the GHS dataset [88] to select 1,454 projects from GitHub that are written in the Java programming language, had a sufficiently large number of pull requests (200), minimally 10 contributors, and are popular (>100 stars). For some of our investigations, we could not use the entire dataset (e.g., because there was no test code, or we could not measure test coverage). While we have taken great care not to introduce any bias in the projects that we study, the filtering conditions that we use might impact the external validity of our results. We urge the research community to replicate our studies, e.g., with more software projects, also in the context of different programming languages.

Another threat is that we only consider 3 different ASATs. While these are the more popular ASATs for the Java programming language [18], we cannot claim that our findings generalize to other ASATs or ASATs in other programming languages.

While we have looked at multiple CI platforms, and went beyond the TravisTorrent dataset in terms of scope [66], we have only considered CI tools that are configured in GitHub. We cannot generalize our findings to CI systems that are used outside of GitHub.

Construct validity. In Section 2.4.5 we study test quality through the *branch test coverage* metric. While test coverage metrics provide a good image of which parts of a code base are not covered by tests, it does not provide insight into how good the tests actually are at detecting faults. Running a full mutation analysis would give us more insight here, but is also a computationally very costly operation.

In Section 2.3.2 we create a code review intensity metric, that normalises the number of code review comments per line of code. While we have validated this metric, we do work under the assumption that more code review comments per line of code are better, but we should also take into consideration that a team of experienced contributors might produce higher quality code in the first place, thus requiring less suggestions on how to improve. Future research should look into the relationship between developer experience and code review intensity.

Another threat to consider in our code review intensity analysis is the possibility of finding comments made by bots in pull requests. We did not filter out these bot-generated comments. Therefore, such comments might be present in our dataset, and should be considered when interpreting our research results.

Our approach for detecting ASAT usage, by relying on file name substrings has limitations. Since there might be cases in which the configuration files exist in repositories, but the ASATs are not used in practice. Also, developers may use ASATs within their IDEs or have CI workflows for ASATs without explicit configuration files in the repository of their project.

In order to determine local buildability, we relied on a "minimal environment" for building the projects out of the box (without any special configuration, reading project specific documents, and environment setup). More specifically, we used a Docker container of Ubuntu 21.04 with the Java Development Kit (Open JDK 11) and build tools such as Maven and Gradle. We do realise that some projects have special needs in order to be built, but in our aim of investigating a large number of projects automatically, this was a conscientious trade-off.

2.9 Conclusion

In this study, we set out to understand the state-of-the-practice in quality assurance in open source software development. In particular, we have focused on software testing, modern code review, automated static analysis, and build automation. We acknowledge that each of these practices have been studied in-depth in isolation, and as such, the major contribution of this study is that we shed light on how the quality assurance practices are used in conjunction. We have collected quality assurance related information from 1,454 open source Java projects from GitHub. This information includes their local buildability outcomes, code coverage results, discussion & review details at the pull request level, usage of automated static analysis tools, and continuous integration related status checks.

In our main research question **RQ1** *(What is current state-of-the-practice in quality assurance in open source software development?)* we first looked into each of the quality assurance practices in isolation, observing a wide variety of their intensity of usage. Moving the goalpost to understand how these quality assurance practices are used together, we observe weak correlation in our set of projects when it comes to (1) code coverage and code review intensity, (2) usage of ASATs and usage of CI, (3) code review intensity and CI usage, and (4) code review intensity and ASAT usage. We also specifically zoomed in on the more mature projects in our dataset, and in general we observe that these are more intense in their application of the aforementioned quality assurance practices, with more focus on their ASAT usage, and code reviewing, but no strong change in their CI usage.

In our investigation of the combined usage of quality assurance practices, we have

overall observed weak correlations. This calls for a future investigation that deeply tries to understand the complementarity of the quality assurance approaches that we consider, and the trade-offs that developers face when choosing between them.

3

Quality Assurance Awareness in Open Source Software Projects on GitHub

Software engineers employ a variety of approaches to ensure the quality of software systems, including software testing, modern code review, automated static analysis, build automation, and continuous integration. To make effective decisions regarding quality assurance (QA), software engineers need to have an awareness of (1) the QA approaches that are in use in a project, and (2) how they are used. Through an exploratory, mixed-methods investigation we set out to better understand the awareness of software engineers in open-source software (OSS) development with regard to QA practices. This involved a large-scale survey of 471 maintainers and contributors on GitHub. Our findings indicate that a high-level awareness among the respondents is common, but also that the respondents are less certain about how the practices are adopted; we further consider the perspective of both the contributor and the maintainer.

3.1 Introduction

As we have grown accustomed to living in a software-filled world, the quality assurance of that software is indispensable [6], because of the potential catastrophic consequences [7]. To ensure the quality of these software systems, software engineers can use a range of quality assurance approaches, e.g., software testing [8, 9, 16, 72, 74, 75, 77, 78, 113, 114], modern code review [15, 16, 26, 115], automated static analysis [18, 19, 103, 116–118], and build automation [21, 22, 24, 83, 119, 120].

In order to make effective decisions with regard to quality assurance, we postulate that software engineers need to have an awareness of the quality assurance (QA) approaches that are in use in a project. This corresponds with the *situational awareness* (SA) theory from psychology, that describes that an understanding of an environment, its composing

This chapter is based on *A. Khatami and A. Zaidman. In IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 174-185. IEEE, 2023.* [112].

elements, and its evolution over time, is important for effective decision making in many environments [51]. This study projects situational awareness onto the domain of QA in software engineering [58]. Our motivation to study the awareness that software engineers working in the Open Source Software (OSS) domain have, is driven by the fact that while many QA tools and principles are available, and many OSS projects adopt them, it is as yet unclear how *aware* software engineers are about the use of QA in the project. Building up our understanding of their awareness, can be a stepping stone towards understanding the decision making process surrounding QA, and potentially improving existing tools to stimulate awareness.

While there have been investigations into the information needs in such contexts as agile software development [121], software development analytics [122], collocated software development teams [123], contemporary code review [124], and CI & CD [125], there is a notable research gap concerning the *overall awareness* of quality assurance practices among software engineers in the context of OSS development. An additional dimension of interest is that GitHub as a social coding platform offers several features to enable or stimulate quality assurance, e.g., the pull-based development model [85], modern code review [15], code coverage reporters [126], and the utilization of GitHub Actions [87], Apps, and Bots [127],

We carry out an exploratory investigation to find out more about OSS developers' awareness about quality assurance in their projects on GitHub. For our investigation, we conduct an online survey with 471 maintainers [76] and contributors [128]. As a means of measuring awareness, we include questions regarding their adoption of various QA tools and practices, but we equally gauge for their knowledge regarding what actually happens during these QA practices, e.g., how many code review comments did they get, or what type and level of code coverage is reported for their project? Our investigation is steered by the following five research questions.

RQ1 How aware are developers of testing in their projects?

Our first research question investigates how aware OSS maintainers and contributors are when it comes to automated testing, a QA practice that was earlier identified to be important in the pull-based development model [128].

RQ2 How aware are developers of code reviews in their projects?

In our second RQ we gauge how aware the software engineers that we survey are with regard to code reviewing, a QA practice that has been shown to decrease functional and maintainability defects in code [15, 26, 44].

RQ3 How aware are developers of automation workflows in their project, and what are the purposes for which they employ these workflows?

Third, our investigation delves into the awareness of maintainers and contributors regarding *automation workflows* [129, 130]. These encompass various tools such as GitHub Actions (GHA)[1], GitHub Checks[2], GitHub Apps[3], and bots, and can help ensure the quality of software during development.

[1] `https://docs.github.com/en/actions`, last visit on July 2023.
[2] `https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/collaborating-on-repositories-with-code-quality-features/about-status-checks`, last visit on July 2023.
[3] `https://github.com/marketplace?type=apps`, last visit on July 2023.

RQ4 How aware are developers of continuous integration in their projects?

Next, we zoom in on the awareness of maintainers and contributors in OSS of CI, as an established QA practice that checks builds, tests, static analysis, etc. [19, 21, 131].

RQ5 How aware are developers of guidelines in their project?

Lastly, we ask OSS maintainers and contributors to find about whether they are aware of their projects' contributing guidelines (e.g., a CONTRIBUTING.md file). We mainly aim to investigate whether projects have contribution guidelines for newcomers, and if projects have any guidelines to keep their software buildable after making changes [58, 64, 132].

Our research findings indicate that respondents are generally aware of the quality assurance practices in use in their OSS projects, however, when it comes to more detailed questions like when a specific QA should be used, or to what level a QA practice is (or should be) done, e.g., levels of code coverage, there is generally a lack of awareness. Similarly, we also noted a lack of awareness surrounding the use of automated workflows, and the benefits of code reviews. Overall, maintainers of projects are more aware of QA practices than contributors, with the notable exception of the presence and use of contribution guidelines.

3.1.1 Information Needs in Software Engineering

Several studies have explored the information needs of software engineers. Baysal *et al.* have investigated the information needs of software engineers when working with bugs, and have made suggestions for improving issue tracking systems [133]. Buse and Zimmermann have studied the information needs of software development managers and have distilled a set of guidelines for software analytics tools [122]. Pascarella *et al.* highlighted seven high-level information needs in modern code review [124], while Ahmad *et al.* have categorized information needs in the context of Continuous Integration [134].

3.1.2 Quality Assurance Practices

Key QA practices include testing, code reviews, continuous integration (CI), and automated static analysis. These practices have been extensively studied both individually and in combination [58].

Kochhar *et al.* found that open-source projects with more lines of code, and a higher number of developers tend to have a greater number of test cases [135]. Vassallo *et al.* investigated automated testing and source code quality inspection in OSS and found that only 11% of builds do quality control in CI [23]. Hilton *et al.* zoomed in on usage, costs, and benefits of CI in OSS. They found popular projects are more likely to use CI, and projects using CI, release more often, accept pull requests faster, and have developers who are less concerned about build breakage compared to projects not using CI [24]. Cassee *et al.* investigated the impact of CI on code reviews. They found adopting CI leads to an average reduction of one review comment per pull request, highlighting the time-saving benefits of adopting CI in code review [28]. McIntosh *et al.* studied the impact of modern code review on software quality. Their results indicate the negative impact of poorly reviewed code on software quality [69]. Beller *et al.* found that 60% of the most popular open-source projects seem to use automated static analysis tools (ASATs) [18]. Zampetti et al. found that open-source software (OSS) build breakages resulting from automated static

analysis tools (ASATs) failures mostly result from failure to adhere to code standards and missing licenses; vulnerabilities contributed less frequently to build failures [104]. In the context of automated workflows, studies looked at expectations of code review bots [136], usage of GHA [87, 137, 138] and its influence on the adoption of other CI services on GitHub [106]. However, there remains a knowledge gap regarding the awareness of all these tools within the QA context. Khatami and Zaidman investigated the state of QA practices in OSS quantitatively, revealing that QA techniques are mostly *not* being used in conjunction [58][4]. Complementing their study, our study will provide insights into the awareness of software engineering professionals regarding QA practices.

3.2 Study Methodology

With approval from our university's ethics committee, we conducted a mixed-methods exploratory study, utilizing multiple-choice and open-ended questions in our survey. We used both qualitative and quantitative approaches to analyze the collected data to assess the level of awareness among 471 contributors and maintainers of open-source software projects on GitHub regarding quality assurance in their projects.

3.2.1 Survey Design

Our survey follows a specific structure, encompassing the following components: informed consent, 7 demographic questions, and 24 questions regarding quality assurance practices within GitHub (see Table 3.2).

Prior to the main survey, we conducted a pilot run with six colleagues to gather feedback on the survey. Their feedback contributed to refining the phrasing of questions and answer options, and adding a back button to our survey.

We run our survey on the Qualtrics[5] platform.

3.2.2 Sampling

In our study, we focused on engaging with active and popular open-source software projects on GitHub. To ensure a representative sample, we utilized GitHub Search [88] to select projects. We selected projects with more than 100 stars, over 50 contributors, and more than 100 pull requests, while excluding forks. We also only considered recent projects with a last commit made after January 1, 2023. This yielded a total of 5,273 projects. We used this set of projects to invite participants for the subsequent phase of our study.

3.2.3 Attracting participants

In addition to promoting the online survey through our Twitter accounts to invite open-source developers to participate, we proactively reached out to the developers of our selected projects on GitHub (Section 3.2.2) to encourage their involvement in our survey. We specifically targeted three groups of developers in the selected projects: assignable users[6] of projects, authors, and assignees of their most recent pull requests. In doing so, we aimed to ensure a balanced representation of both contributors and maintainers.

[4] Chapter 2 of this dissertation.
[5] `https://www.qualtrics.com/`, visited on July 2023.
[6] https://docs.github.com/en/graphql/reference/objects

We collected 18,287 data records containing developers' contact information. We removed any redundant records, resulting in a final dataset of 12,337 unique records.

To invite these developers to voluntarily participate in our survey, we sent personalized emails to 12,337 email addresses. In each email, we included relevant information such as their GitHub user account, the repository they have contributed to, a clear explanation of our research objectives, and estimation of the time needed to complete the survey (15 minutes). Besides, we emphasized the anonymity of their participation by providing an anonymous link to the survey.

Because of the anonymous link to the survey, we are not able to provide an exact response rate for each source of recruitment (targeted recruitment versus Twitter).

We conducted our survey from February 19 to March 10, 2023, and we received 644 non-empty submissions. After filtering out incomplete submissions, we obtained a final total of 471 complete responses. However, this does not mean that all questions were answered since the survey questions were optional. Precise information regarding how many participants answered each question is presented in the results section.

3.2.4 Data Analysis

Out of the 24 non-demographic and non-consent questions in the survey, 11 were open-ended. To analyze these open-ended questions, we employed an open card sorting approach [139]. Initially, the first author independently performed coding and categorization, manually sorting the responses into meaningful groups [140]. To establish consensus, a discussion meeting was conducted between both authors to finalize the code names and categories for each question. We opted for open card sorting to allow codes and categories to emerge naturally, without imposing pre-defined groups.

For questions primarily focused on naming tools or indicating the presence or absence of specific practices, we categorized the responses into a limited number of emerging categories. Conversely, for questions allowing more open-ended responses, we employed a combination of open coding and card sorting techniques to identify themes.

Additionally, we included 10 text-entry fields for certain answer options in multi-option questions. We did this to clarify the "Other" option and to collect specific information such as comment counts based on selected options, or relevant links. We also used open card sorting.

Table 3.1 shows an overview of the participants to our survey. Regarding participant roles, the majority (60.2%) identified themselves as maintainers, integrators, project (co-)owners, or other similar positions. We refer to this group as *maintainers*. The remaining participants self-identified as *contributors*, and only one respondent left this question open.

Of all respondents, 451 provided the handle (`username/repository`) of the most recent GitHub project they had contributed to. There were 415 distinct OSS repositories mentioned in total.

Table 3.1: Participant responses to demographic questions.

	<1	1—3	3—6	7—10	>10
Experience in Software Development (in years)	2 (0.4%)	38 (8.1%)	101 (21.9%)	76 (16.1%)	252 (53.5%)
OSS Experience (in years)	25 (5.3%)	82 (17.6%)	122 (26.1%)	83 (17.6%)	157 (33.3%)
Occupation(s) (multiple choice)	Academia 15.4%		Industry 65.2%	OSS 34%	Other 12.8%
Role in Project	**Contributor** 187 (39.8%)				**Maintainer** 283 (60.2%)
Contributing to an OSS project is the main professional activity?	**Yes** 174 (36.9%)				**No** 297 (63.1%)

3.3 Results

In this section, we present an analysis of the survey data, structured along the quality assurance practices and our research questions, see Table 3.2.

In our investigation, we aimed to gather insights into the current landscape of QA awareness among OSS developers by examining both their awareness and adoption of consolidated QA resources on GitHub. We evaluated respondents' awareness level based on five types of answers to our survey questions: (1) "I don't know'' answers, these responses explicitly indicate a lack of awareness. (2) "No or Zero'' answers: these answers indicate that the practice is not followed in the project or no information is available in that area, which we use as a proxy for lack of awareness. (3) "No Answers'': when respondents leave a question blank, we also consider it as a proxy for lack of awareness, although it is a limitation of our work as some respondents might be aware but chose not to answer. (4) "Invalid Answers'': these are answers that are not related to the question or are too general or vague, implicitly indicating a lack of awareness about the topic. (5) "Other Answers": these are informative and valid responses that explicitly indicate the respondent's awareness of the topic. A summary of all answers is shown in Figure 3.1; we will use this information to answer each of the research questions.

3.3.1 RQ1) How aware are developers of testing in their projects?

T-Q1) Does the project have automated (scripted) tests?

The majority of respondents (87.5%) reported having automated tests in their project, while 10% said they do not have tests, and 2.5% did not know the answer to this question.

Table 3.2: Survey Questions Details

Code	QA-related Category	Question	Type
T-Q1	Testing	Does the project have automated (scripted) tests?	multiple-choice
T-Q2	Testing	What kind of (functional) tests do you have in the project? (leave open if you don't know)	multiple-choice
T-Q3	Testing	What kind of test coverage information is available for the project? (leave open if you don't know)	multiple-choice
T-Q4	Testing	Considering your answer to the previous question, do you know how much is the code coverage of the project?	multiple-choice
T-Q5	Testing	What kind of tooling do you use for testing the project? (frameworks, libraries, etc.)	text entry
CR-Q1	Code Review	How many comments did you receive (as the author of the pull request), or give (as the maintainer)?	text entry
CR-Q2	Code Review	The comments that you received/gave were about:	multiple-choice
CR-Q3	Code Review	How many code reviewers were involved in the pull request? (Not considering code review bots)	multiple-choice
CR-Q4	Code Review	Does the project use GitHub's Checks in pull requests?	multiple-choice
CR-Q5	Code Review	What kind of reports do you get from GitHub's Checks' comments on your pull requests?	multiple-choice
CR-Q6	Code Review	How does code reviewing help you?	text entry
AW-Q1	Automation Workflows	Do you use GitHub Actions and/or GitHub Apps in the project?	multiple-choice
AW-Q2	Automation Workflows	Which type of GitHub Actions do you use in the project? (Please name them)	text entry
AW-Q3	Automation Workflows	Which GitHub Apps do you use in the project? (Please name them)	text entry
AW-Q4	Automation Workflows	Do you use bots in the project?	multiple-choice
AW-Q5	Automation Workflows	Which bots do you use in the project? (Please name them)	text-entry
AW-Q6	Automation Workflows	Do you use ASATs in the project?	multiple-choice
CI-Q1	Continuous Integration	Does the project have CI workflows configured on GitHub?	multiple-choice
CI-Q2	Continuous Integration	How do you (or other people in the project) configure CI workflows on GitHub?	text entry
CI-Q3	Continuous Integration	When and how do you check the project's CI results on GitHub?	text entry
CI-Q4	Continuous Integration	Describe a situation in which CI workflows helped you in your contribution to the project.	text entry
G-Q1	Guidelines	Does the project have a checklist/guideline to check before contributors submit or maintainers review a pull request?	multiple-choice
G-Q2	Guidelines	Does a newcomer get any guidelines about how to contribute to the project? If yes, can you provide a link to it?	text entry
G-Q3	Guidelines	Does the project have specific guidelines (e.g., CONTRIBUTING.md) to make the latest version of the project buildable/compilable?	text entry

Respondents who did not select "Yes" as their answer to this question were excluded from getting any subsequent questions about testing in the survey.

T-Q2) What kind of (functional) tests do you have in the project?
Out of all respondents, 84.7% reported knowing the specific type(s) of testing used in their project (e.g., unit testing, integration testing, etc.). A large subset of the population indicated to have unit tests in their project (92%), while 72% mentioned having integration tests (72%). A smaller group of respondents reported using validation (29%) and system testing (24%). Another 30 respondents (7.5%) selected the "Other" option and specified additional testing methods not included in the answer options. Among those end-to-end testing (7), fuzzing, performance, and UI testing (3) were the most frequent ones.

T-Q3) What kind of test coverage information is available for the project?
Of the respondents who answered this question, 80.8% reported that statement/line coverage information was available in their project. Branch coverage was indicated to be available by 41.4% of respondents, while 8.8% reported using mutation score to assess their tests. Additionally, 43 respondents (17.3%) selected the "Other" option and specified additional types of code coverage; out of those responses, 32 were valid, with 25 indicating that no code coverage information was available to them, while 2 were unsure. Other responses included using *Coveralls.io*[7] (1), class coverage (1), line coverage (2), and branch coverage (1).

T-Q4) Considering your answer to the previous question, do you know how much is the code coverage of the project?
Around half of the participants (53.7%) reported that code coverage information was not available on the project's GitHub page. In contrast, 32.6% of respondents indicated that they knew their project's code coverage, and 13.7% reported that the information could be found on the project's GitHub page.

We asked the participants who indicated that they knew the value of code coverage to provide this value through a text input field. Out of 124 entries received, 106 contained valid code coverage information. For participants that reported that the code coverage information was available on their project's GitHub page, we requested the specific location of the information through a separate text input field; of the 52 entries received, 11 provided a valid location for the code coverage information, while 21 provided the actual code coverage. The participants identified various locations to access code coverage information, including the README on Github, GitHub Actions, external tool reports (e.g., *Codecov*[8], pull requests, CI status/pipeline, and build artifacts.

Further analysis based on the classification scheme proposed by Heitlager *et al.* [42] showed that most (61.3%) respondents' projects have either high (39) or very high code coverage (26), while 24.5% had a moderate level of code coverage (26), and the remaining 14.2% of projects had either a low (12) or very low (3) code coverage levels.

[7] `https://coveralls.io/`, last visit on July 2023.
[8] `https://about.codecov.io/`), visited on July 2023.

Participants were more responsive when asked about the type of code coverage in their project (T-Q3 with 249 responses) compared to when asked about its value (T-Q4 with 176 responses saying either they know the value or they can find it on project's GitHub repository).

T-Q5) What kind of tooling do you use for testing the project?

Of all responses, 343 provided valid inputs. Upon analysis (explained in Section 3.2.4 – Data Analysis) of responses, we found that all respondents mentioned the name of at least one tool, framework, library, or approach for testing in their project, with some mentioning up to 15 such tools. The mean number of tools mentioned was 2.16, and the median was 2. From these responses, we selected the most commonly mentioned tools, which are shown in Table 3.3.

Table 3.3: Top 50% tooling, frameworks, libraries, or approaches used for testing (T-Q5). All websites visited July 2023.

Categories	**URL**	**# of mentions**	**%**
PyTest	https://pytest.org/	50	6.8%
PHPUnit	https://phpunit.de/	35	4.8%
GitHub Actions	https://docs.github.com/en/actions	35	4.8%
Go test	https://pkg.go.dev/testing	30	4.1%
JUnit	https://junit.org/	26	3.5%
Cargo (Rust)	https://doc.rust-lang.org/cargo/	24	3.3%
Custom		22	3%
Rust test	https://doc.rust-lang.org/cargo/	20	2.7%
Jest	https://jestjs.io/	19	2.6%
CodeCov	https://about.codecov.io	15	2%
Rspec (Ruby)	https://rspec.info/	12	1.6%
Scripts (bash, etc.)		12	1.6%
Unittest (Python)	https://docs.python.org/3/library/unittest.html	11	1.5%
Google test	https://github.com/google/googletest	10	1.4%
Coveralls	https://coveralls.io	9	1.2%
Xunit (.Net)	https://xunit.net/	8	1.1%
Coverage.py	https://coverage.readthedocs.io/en/7.2.7/	8	1.1%
Playwright	https://playwright.dev/	8	1.1%
Jenkins	https://www.jenkins.io/	7	1%
Cypress	https://www.cypress.io/	7	1%
Others (246 different categories)		366	50%
Total		734	100%

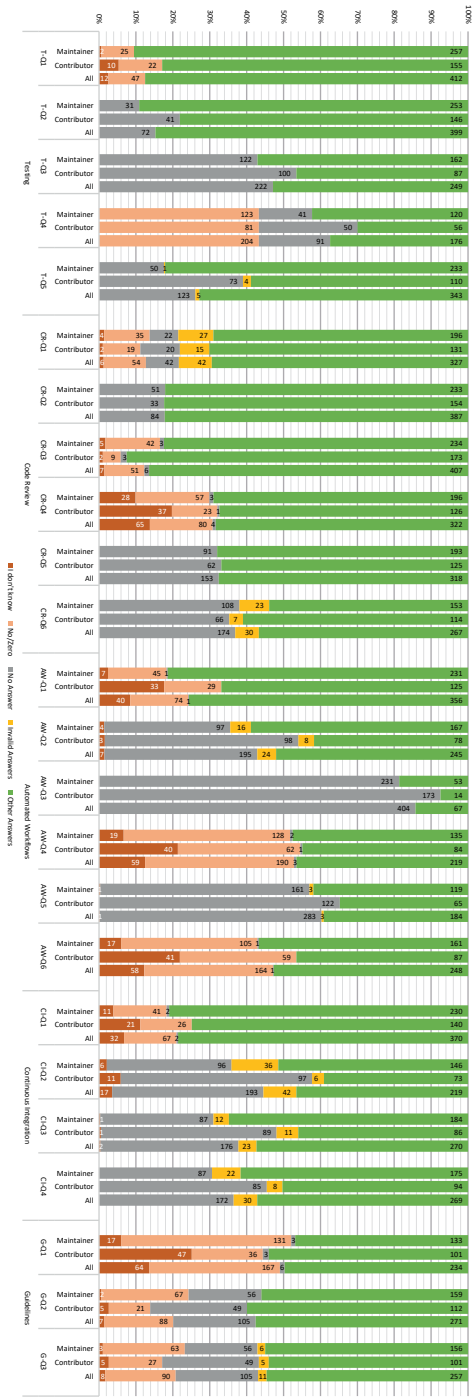

Figure 3.1: Summary of answers to all questions.

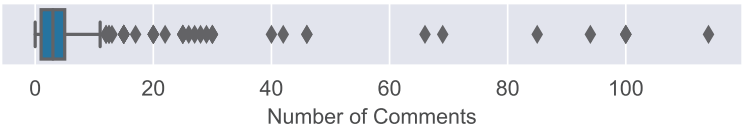

Figure 3.2: Comments received/given in a pull request.

RQ1 Summary

Figure 3.1 shows that ~87% of the participants to our survey indicate to know whether the project they most recently contributed to has test scripts (T-Q1). Similarly, ~85% of the respondents know the type(s) of tests a project employs (T-Q2), and ~74% know the testing tools in use (T-Q5). However, the awareness surrounding the type of test coverage (T-Q3; ~53%) and the actual level of test coverage (T-Q4; ~37%) is considerably lower.

3.3.2 RQ2) How aware are developers of code reviews in their projects?

For the following questions, we asked the survey participants to reflect on their most recent involvement in a pull request (PR) when responding.

CR-Q1) How many comments did you receive (as the author of the pull request), or give (as the maintainer)?

Figure 3.2 displays a box plot based on the number of comments provided by 373 respondents. Also, 57 responses were excluded due to imprecise or vague numerical ranges (e.g., thousands, hundreds, or less/more than 10, etc.). Only exact values or specific ranges were included, and in cases of ranges, the average was calculated. Accordingly, the median number of comments per PR is 3. Furthermore, 50% of the respondents' PRs had between 1 and 3 comments, while the remaining 50% ranged from 3 to 11 comments.

CR-Q2) The comments that you received/gave were about?

We received 387 responses for this question. The majority of respondents (63%) reported that their comments were related to *Code Quality or Maintainability Defects* [26], while 35.7% of respondents indicated that their comments were about "Functional Defects". A slightly lower percentage indicated the "Tests" option (32%). Moreover, 39.5% selected the "Other" option to specify additional topics for their comments in a PR.

For this question, we asked participants to indicate the number of comments they had per specific topic they selected from the answer options. The comment count distribution for each topic is depicted in Figure 3.3. The median number of comments for the "Tests" and "Functional Defects" categories was 1, which was lower compared to the other two categories with a median of 2. The number of responses provided for each topic were as follows: "Tests" (52), "Code Quality or Maintainability Defects" (100), "Functional Defects" (62), and "Others" (49).

We present the qualitative analysis (see Section 3.2.4) of the "Other" responses in Table 3.4. The "functional aspects" of change encompass comments related to business logic, code behavior, design decisions, architecture, features, functionality, implementation, solution/approach, compatibility, dependencies, CI, build, testing, bugs, defects, and API. The "maintainability aspects" involve comments related to code quality of the contribution, textual content, formatting, code style, documentation, change log, and typos. The "non-functional aspects" of change include mentions of performance, accessibility, and appearance. In the category of "social and communication", respondents highlighted comments that involve pinging someone to review the PR, expressing gratitude towards contributors, and providing encouragement. Lastly, the "suggestions" category includes comments that offer improvements, suggestions, clarification, verification of changes, and discussions about unnecessary changes.

Of interest to note here is that respondents used the "Other" category to list defects that we would classify as either *functional* or *maintainability* aspects (also see the work of Beller *et al.* [26]). However, the respondents were perhaps not aware of the exact terminology. We explicitly chose to not reclassify these answers.

CR-Q3) How many code reviewers were involved in the pull request? (Not considering code review bots)

Among the respondents that provided an answer, 11% reported that no reviewers were involved in their PR, 46.6% reported one reviewer, while 28.4% had two reviewers, and

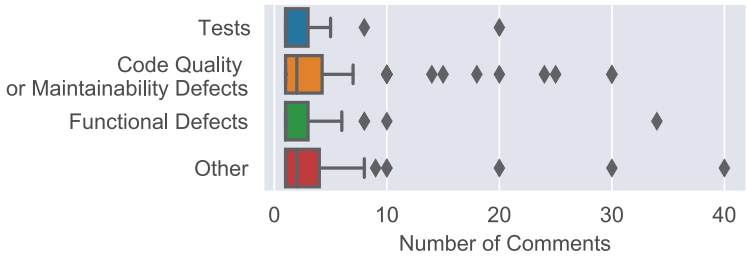

Figure 3.3: Count of comments per topic in a pull request.

Table 3.4: Categories of other topics of comments in a PR (CR-Q2)

Categories	**# of mentions**	**%**
Functional, logical, behavioral aspects of change	51	38%
Maintainability aspects of change	29	22%
Social and communication	21	16%
Suggestions, improvements, and verification of change	18	13%
Non-functional aspects of change	3	2%
Other	12	9%
Total	134	100%

12.5% selected the "more than 2 reviewers" option. With 1.5% of the respondents indicating to be unsure of how many reviewers were involved, we observe a high level of awareness.

CR-Q4) Does the project use GitHub Checks in pull requests?
GitHub Checks provide information on build outputs, static code analysis results for the changes made in a PR. This information is provided on GitHub's PR page, and contributes to the general QA awareness. We asked respondents whether they use this feature in their projects.

Half of the respondents (50.3%) indicated to use *Checks*, while 35.9% said they do not, and 13.8% are not aware of it; four did not provide an answer.

CR-Q5) What kind of reports do you get from GitHub's Checks' comments on your pull requests?
Participants who answered *Yes* to CR-Q4 were asked about the types of reports they receive from the *Checks* feature. The most commonly selected ones were "Test results" (57.8%) and "Build checks" (55%); these two options were frequently selected together because tests are run during the build. Other options selected were "Code quality warning" (39.4%) and "Code coverage reports" (23.3%). Only a small percentage (4%) selected the "Other" option. Of those, 22 respondents provided additional information about the types of checks they use, including DCO (Developer Certificate of Origin)/commit format checks, licensing checks, compatibility checks, and security checks.

CR-Q6) How does code reviewing help you?
A total of 297 responses were collected for the question. However, 30 of these responses were excluded as they were considered too general (e.g., "yes," "no," "it doesn't") or unrelated to the question. We conducted a qualitative analysis (see Section 3.2.4) of the remaining responses and identified several high-level categories. The frequency of these categories is summarized in Table 3.5. It should be noted that some respondents mentioned multiple reasons why code reviewing was helpful to them. The most frequently mentioned category was "improving code quality" (173 mentions), followed by "error/bug prevention" (124 mentions), and "knowledge sharing" (65 mentions).

RQ2 Summary

Figure 3.1 shows that ~69% (CR-Q1) and ~68% (CR-Q4) of respondents are aware of the number of code review comments and the usage of *Checks* during PRs. Similarly, ~68% of survey respondents indicate to be familiar with GitHub Check reports (CR-Q5). Most respondents are aware of the number of reviewers involved (~86%; CR-Q3) and the type(s) of comments they give (~82%; CR-Q2). We observe a lack of awareness regarding the benefits of code reviewing (~57%; CR-Q6).

3.3.3 RQ3) How aware are developers of automation workflows in their project, and what are the purposes for which they employ these workflows?

Table 3.5: Categories of How Does Code Reviewing Help You (CR-Q6)

Categories	# of mentions
Non-Functional (173)	
Code Quality	161
Code Performance	8
Security	4
Functional (124)	
Error/Bug Prevention	120
Verification	14
Knowledge Sharing (65)	
Learning and Knowledge Sharing	65
Other (29)	
Design (code, architecture, and software)	12
Testing	6
Speed-up Workflow	3
Community	3
Collective Decision-making	1
Static Analysis	1
Team Confidence	1
Think More	1
Project Policy	1

AW-Q1) Do you use GitHub Actions and/or GitHub Apps in the project?
GitHub Marketplace offers two distinct types of tools for automating development workflows: GitHub Apps (Apps) and GitHub Actions (GHA). Both offer similar functionality, including the ability to automate tasks, customize and extend GitHub's features, and integrate with third-party services. However, there are also differences between these two tools. GHA is built into GitHub's platform, whereas Apps are integrated through the GitHub API. As a result, GHA is primarily focused on automating workflows within a repository, while Apps are more versatile and can be used to interact with multiple repositories and third-party services.

We included this survey question to determine participants' awareness and usage of these two tools. Out of 470 responses, 56.6% reported using only GHA, while 17.2% used both GHA and Apps, and 15.7% used only Apps. Also, 15% reported not using either tool, and 8.5% were unaware of the answer.

AW-Q2) Which type of GitHub Actions do you use in the project? (Please name them)
From the 347 respondents that indicate to use GHAs in AW-Q1, 276 provided input for this question. After excluding 24 invalid inputs, we qualitatively analyzed the responses (see Section 3.2.4), resulting in the categories presented in Table 3.6. CI/CD, environment setup, and code quality analysis are among the top Actions types used by the respondents of our survey.

These findings offer insights into the awareness of our survey participants about types of GHAs used in their projects.

Table 3.6: Categories of GHA Types According to Responses (AW-Q2)

Categories	# of mentions
Continuous Integration and Deployment	
Testing	45
CI	37
Build	36
Build/Test	31
Release/Publish	26
Deployment	18
Versioning	3
Dependency Management/Review	3
Compile	2
Close Milestones	1
Code Quality and Analysis	
Code Quality/Code Style	32
CodeQL	22
Codecove/Code Coverage	15
Static Code Analysis	5
Security	2
Documentation and Management	
Upload/Generate Artifacts	20
Issue/PR Management	12
Wiki/Documentation	5
Notifications	3
Update Changelog	2
UI Translation	1
License Analysis	1
Other	
Environment Setup	77
Link to GHA Configurations	12
Custom	7
Check File Structure	1

AW-Q3) Which GitHub Apps do you use in the project? (Please name them)
Similar to AW-Q2 we surveyed the 90 respondents who reported to use Apps in AW-Q1, and 67 of them (74.4%) provided input to this question. The respondents mentioned a wide variety of Apps, with *Codecov* (19) , *Renovate* (9), *CodeQL* (5), *Slack* (4), *Travis CI* (4), *Dependabot* (3), *DCO*[9] (3), *Azure Pipelines* (3), *AppVeyor* (3), *Vercel* (3), and *SonarCloud* (3) being the most commonly mentioned ones.

AW-Q4) Do you use bots in the project?
Before gauging the awareness surrounding bots, we made it clear to respondents that we were referring to bots such as *Dependabot*. We received 46.8% positive answers, while 40.6%

[9]This App enforces the Developer Certificate of Origin (DCO) on pull requests. From https://github.com/apps/dco, visited on July 2023.

responded that they do not use bots in their projects; 12.6% selected "I don't know".

AW-Q5) Which bots do you use in the project?
From the 219 respondents who indicated to use bots in their projects, 188 participants provided at least one bot. In total, we identified 54 distinct bots. Some notable bots are: Dependabot, which was mentioned 111 times, something that could potentially be attributed to the fact that we cited Dependabot as an example in the question, *Stale* (14 mentions), *Renovate* (9 mentions), *Bors*[10] (9 mentions), and *Codecov* (8 mentions). Moreover, 8 respondents specified to use custom bots.

AW-Q6) Do you use ASATs in the project?
After providing an explanation of automated static analysis tools (ASATs) to the participants of our survey, we inquired whether they utilize them in their projects. Out of the 470 respondents who answered this question, the results were as follows: 52.8% responded with "Yes", 34.9% replied with "No", and 12.3% selected the "I don't know" option.

RQ3 Summary

Figure 3.1 shows that ~76% of the survey respondents indicate that they use GHA or GitHub Apps in their project (AW-Q1). However, there is lack of awareness about the specific type of Apps they use (~14%; AW-Q3) compared to Actions (~52%; AW-Q2). Additionally, we observe relatively low awareness of ASATs among respondents, with ~53% indicating to know about ASAT usage in their projects (AW-Q6). Also, less than half of survey participants are aware of bot usage (~46%; AW-Q4) and which ones they use (~40%; AW-Q5) in their projects.

3.3.4 RQ4) How aware are developers of continuous integration in their projects?

CI-Q1) Does the project have CI workflows configured on GitHub?
In response to AW-Q2, the respondents mentioned Testing, CI, and Build as the most commonly used categories for GHA types in their projects. This finding is consistent with the purpose of GitHub's automated workflow tools, which aim to integrate continuous integration (CI) into the software development process. These tools automate tasks such as building and testing code changes[11]. In this question, we asked participants whether they had configured CI workflows for their projects using GitHub's automated workflows.

Of the 469 responses received, 78.9% stated they had CI workflows, while 9.2% reported configuring CI workflows on platforms other than GitHub, 6.8% said they did not know about it, and 5.1% indicated they did not have CI workflows for their projects.

[10] `https://github.com/apps/bors`, visited on July 2023.
[11] From https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration, visited on July 2023.

CI-Q2) How do you (or other people in the project) configure CI workflows on GitHub?

We posed an open-ended question to the 370 participants who reported using CI workflows on GitHub, asking them to share how they or other people in their project configure these workflows.

We received 278 responses, with 93 respondents reported using GHA (GitHub Actions) for configuration, while 108 mentioned editing YAML configuration files. It is important to note that some responses were either invalid or vague (42 in total), and 17 participants indicated that they were unsure about the answer. Additionally, 20 respondents mentioned services or tools outside of GitHub.

We received further insights from some of the respondents regarding their approach to configuring CI workflows:

1. 14 respondents said they manually edit the (.yml) configuration files,

2. 6 respondents mentioned "copying" configurations from other projects or Stack Overflow,

3. 3 respondents mentioned using GHA examples and shared workflows (of GitHub) when configuring their CI.

CI-Q3) When and how do you check the project's CI results on GitHub?

We posed an open-ended question to participants, asking them about *when* and *how* they check CI results on GitHub. We received a total of 295 responses, out of which 23 were deemed invalid or vague, and 2 indicated not knowing the answer to this question. After qualitatively analyzing the remaining 270 responses (described in Section 3.2.4), categories presented in Table 3.7 were emerged.

We observe that Pull Requests (PRs) play a crucial role in the evaluation of CI workflows. They emphasized the importance of reviewing CI results for each commit and push, particularly in cases of failures. Moreover, respondents mentioned receiving notifications on GitHub or via emails to stay informed about these failures.

CI-Q4) Describe a situation in which CI workflows helped you in your contribution to the project.

In total, we received 269 responses (excluding 30 invalid and vague ones) and qualitatively analyzed them (as described in Section 3.2.4).

We summarized the responses in Table 3.8. The majority emphasized the benefits of CI in terms of testing and bug prevention. Additionally, participants highlighted the importance of building the project in different environments, to mitigate the "works-on-my-machine" mentality. They also mentioned the impact of CI on code quality, which was mentioned almost as frequently as building in different environments.

Table 3.7: When/how developers check CI results on GitHub (CI-Q3)

	Categories	# of mentions
When	Per/during/after/before each **PR/merge/change**	154
	Per each **commit**	42
	In case of **failure/error**	28
	Getting **email/notification**	26
	Per each **push**	25
	Per/during/before **(code) reviewing**	19
	On a **regular basis**, daily or monthly	8
	Others: per/during deploy, after manual test, part of branch workflow, after workflows run	5
How	On **PR page** and **PR Checks tab**	73
	By checking **emails/notification**	17
	On **Actions tab**	11
	On **commit Checks**	9
	On **(badges in) README file**	7

RQ4 Summary

Based on the summary of answers presented in Figure 3.1, the majority of respondents indicate to have knowledge about CI workflows configured on GitHub (~79%; CI-Q1). Also, ~57% of them show awareness of how CI helps in contributing to a project (CI-Q4). Similarly, ~57% know how and when to check GitHub's CI workflows results (CI-Q3), however, there is less awareness on how to configure them (46.5%; CI-Q2).

Table 3.8: Categories of how CI helps developers (CI-Q4)

	Categories	# of mentions
How & in what areas?	Testing	106
	Bug/failure/mistake/breakage prevention	98
	Build with different versions/configurations/environments & avoid works-on-my-machine mentality	47
	Static code analysis/code style/formatting/code quality	46
	Build/compile	29
	PR/contribution verification	18
	Automation of manual tasks: reduce workload, local environment limitation, commits' sign-off message, and maintaining repo files	12
	Prevention of regressions	11
	Deployment/release/publishing	11
	(Code) Reviews	11
	Improve contribution confidence	6

3.3.5 RQ5) How aware are developers of guidelines in their project?

G-Q1) Does the project have a checklist/guideline to check before contributors submit or maintainers review a pull request?

Of the total number of respondents, 234 (50.3%) answered *yes* to this question, while 167 (35.9%) answered *no*. Another 64 (13.8%) answered that they do not know, and 6 did not answer.

Out of 234 respondents who answered positively, 173 (74%) provided input on the location of the contribution checklist/guideline. The CONTRIBUTING.md file was the most popular location (81 respondents), or the similar DEVELOPER.md file (3 respondents). Another 42 respondents pointed to an issue or pull request template, while 29 respondents pointed to a separate document, and 5 persons indicated the README.md file. Lastly, two respondents said the information is provided privately to contributors, and five did not provide valid responses.

G-Q2) Does a newcomer get any guidelines about how to contribute to the project? If yes, can you provide a link to it?

We manually reviewed a total of 366 responses to this question. Out of the respondents, 270 (74.8%) confirmed the presence of a contribution guideline in their project. While 216 participants provided a link to the guideline, 87 respondents indicated that their project did not possess such a guideline, and 8 participants were uncertain about its existence.

Interestingly, respondents shared various additional practices they employ to support newcomers in their projects. Six participants mentioned leveraging communication platforms like Discord or Matrix to keep in touch with and guide newcomers. They also highlighted the significance of directing newcomers to *good first issues* [141, 142] and encouraging them to initiate discussions regarding their motivations for contributing.

G-Q3) Does the project have specific guidelines (e.g., CONTRIBUTING.md) to make the latest version of the project buildable/compilable?

Similar to the previous question, we inquired about the existence of guidelines regarding making the latest version of participants' projects buildable or compilable. We manually reviewed a total of 366 text-entry responses to gather insights on this topic.

Around 70% of them indicated that guidelines are in place to ensure buildability or compilability, pointing to the CONTRIBUTING.md, or the README.md file for these guidelines. Another 90 respondents (24.5%) indicate to not have these guidelines for their projects, while 8 respondents were unsure whether these guidelines exist for their project. Moreover, 5 respondents claim that building is not applicable to their project, and 6 answers were considered invalid.

Additionally, (4) respondents pointed out the importance of their CI build and test workflows in ensuring that the latest version of their software builds successfully.

RQ5 Summary

Figure 3.1 shows a summary of survey respondents awareness about three types of guidelines in projects: for newcomer contributions to project (G-Q2; ~58%), keeping the latest version of software buildable/compilable (G-Q3; ~55%), and as a checklist when reviewing or submitting pull requests (G-Q1; ~50%. In all, we observe an average level of awareness about project guidelines in OSS.

3.3.6 Maintainers & Contributors Level of Awareness

As we compare the awareness of maintainers and contributors, we turn our attention to Figure 3.1.

In general, we observe that maintainers are more aware of QA activities in their projects compared to contributors. This is in line with earlier findings from Gousios *et al.* who established that maintainers are more involved in ensuring a project's quality [76]. This is especially true for questions related to *testing*, *the automation workflows in use*, and *CI*.

However, we see a mixed picture for *code reviewing*, where we see indications of higher awareness among contributors compared to maintainers, in particular when it comes to the number of reviewers active in a pull request (CR-Q3) and the benefits of code review (CR-Q6).

Also, contributors are (slightly) more aware of newcomer guidelines and pull requests' checklist/guidelines than maintainers (~60% vs. ~56%; G-Q2, and ~57% vs. ~47%; G-Q1).

3.4 Limitations

We conducted a survey to investigate the level of awareness among OSS developers regarding the quality assurance (QA) practices in their GitHub projects. To ensure unbiased responses, we carefully formulated the survey questions, avoiding leading or ambiguous language. However, it is important to acknowledge that our study has certain limitations that could potentially impact the validity of our findings.

Generalizability. Our findings may not apply to other populations of OSS developers, or OSS projects. While we tried to avoid constraints in our sampling, e.g., limiting the programming language, we acknowledge the need for replication.

Researcher bias may introduce categorization bias when using qualitative research methods to categorize answers of open-ended questions in our survey. Additionally, this bias could potentially influence the wording of the questions. To address this limitation, we took measures to mitigate it by conducting a pilot round of the survey (see Section 3.2.1).

Survey responses validity. The order of questions, the presence of open-ended questions, and respondents' inclination to present themselves in a positive light (e.g., by claiming awareness of all QA activities in their project) may have influenced the accuracy of the provided answers. Similarly, some answers might be estimations, e.g., T-Q4 about test coverage, instead of very precise numbers.

Measuring awareness. While our approach described in Section 3.3, serves as an approximation to measure respondents' awareness, it has limitations. Specifically, using "No Answers","No/Zero", and "Invalid Answers" as proxies of lack of awareness may not

always be accurate, e.g., respondents may leave a question unanswered despite having the knowledge.

3.5 Conclusion, Implications, and Future Work

We set out to better understand the situational awareness of software engineers when it comes to quality assurance practices in their open source software projects on GitHub.

We found that while there is awareness surrounding test scripts (~87%; T-Q1), types of tests (~85%; T-Q2), and testing tooling (~74%; T-Q4), awareness is lacking when it comes to test coverage (~53%; T-Q3), either because coverage reports are unavailable or contributors are not familiar with them [**RQ1**].

In the context of code review, our findings indicate that respondents demonstrated a higher level of awareness compared to other practices (~69%; CR-Q1, ~68%; CR-Q4 & CR-Q5, ~82%; CR-Q2, and ~86%; CR-Q3). However, there is lack of awareness surrounding the benefits of code review (~57%; CR-Q6) [**RQ2**].

We have observed that while 76% of the respondents indicate that a workflow automation is in use in their project, they generally lack awareness on the precise type of GitHub Action (~52%), or App (~14%). When it comes to bots, ~46% indicates to know that a bot is in use, with less respondents indicating to know which bots are in use (~40%) [**RQ3**].

We see that ~79% of the respondents are aware that there is a CI configured for their project, yet only around 57% knows how and when to check the CI results [**RQ4**].

When we turn our attention to the awareness of guidelines in GitHub projects, we observe that only ~50% to ~58% of the respondents are aware of such a set of guidelines existing. Either creating such guidelines, or increasing the awareness surrounding them, can likely improve the onboarding and developer experience [**RQ5**].

When we compare participants' roles, we observe that in general maintainers are more aware of QA practices in their projects. However, when it comes to code reviewing or guidelines, like the number of reviewers in a pull request (CR-Q3), the benefits of code review (CR-Q6), newcomer guidelines (G-Q2), and PRs' checklist/guidelines (G-Q1) contributors show a higher level of awareness.

Implications. The identification of awareness gaps through this analysis can pave the way for **educators** to emphasize the importance of situational awareness among the next generation of software engineers. **Tool makers** can improve their tooling to stimulate awareness among software engineers, similar to what TestAxis does for testing [143]. **Researchers** should study more deeply how and why software engineers either gain or lack knowledge of specific QA practices.

Future work. A publicly available data set [144] with 456 anonymized survey answers of our study makes it possible to replicate our study (15 respondents did not give consent to archive their answers). Also, the dataset can support other potential future research directions like: (1) using respondents' repository handles in our data set to see how reported answer compare to the reality of QA in OSS projects, (2) expand the work on private repositories or industrial projects, (3) using GitHub projects with different characteristics to see if and how QA awareness changes among those, (4) building tools to help increase OSS developers knowledge of QA in their projects, particularly in areas that we observed lack of awareness.

4

Software Quality Assurance Analytics: Enabling Software Engineers to Reflect on QA Practices

Software engineers employ a variety of approaches to ensure the quality of software systems, including software testing, modern code review, automated static analysis, build automation, and continuous integration. Previous research has shown that software engineers lack situational awareness about the quality assurance (QA) practices in their projects. We propose software quality assurance analytics *to help software engineers become aware of their QA usage, and reflect upon it. We have developed a prototype quality assurance analytics dashboard coined* RepoInsights *that provides both a global overview and a comparative aspect of the different QA practices. Through a qualitative study involving 14 participants who have completed an assignment using our* RepoInsights *dashboard, we elicit their perspective on quality assurance analytics. We observe that the dashboard has enabled the software engineers to reflect on the QA practices of software projects. Additionally, we distill a set of recommendations for future quality assurance analytics.*

4.1 Introduction

Because of the growing importance of software in our society, the *quality assurance* (QA) of that software is indispensable [6]. This view is supported by countless reports of software failures that have caused severe harm to businesses, people, or society as a whole [7, 8]. Software engineers can use a range of QA mechanisms to ensure the quality of the software systems they produce, including software testing [8, 16, 77–79, 113], modern code review [15, 16, 26] automated static analysis [18, 19, 103, 116, 117], and build automation [21,

This chapter is based on *A. Khatami, C. Brandt, and A. Zaidman. In IEEE 24th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 189-200. IEEE, 2024.* [52].

22, 24, 52]. However, a recent study has indicated that software engineers lack situational awareness [133] when it comes to the QA efforts of the projects they are maintaining or contributing to [112].

An established approach to raise awareness is software analytics (SA), which makes software data available to engineers and managers, empowering them to gain and share insights leading to better decisions [145]. The combination of insights that: quality assurance is generally considered important, the existence (and use) of a plethora of QA mechanisms, and the lack of situational awareness of whether, how, and to what degree these QA mechanisms are used, leads to our idea to explore software analytics focused on QA data. We conjecture that providing so-called *quality assurance analytics* (QAA) to software engineers will increase their awareness about the quality assurance situation in their projects. This in turn can be useful to maximize the use of (a combination of) QA mechanisms, and hopefully lead to less quality incidents.

In this paper we investigate **how to design a quality assurance analytics dashboard in order to empower software engineers to reflect on the QA practices of their projects**. To this end, we design a QAA dashboard that provides a global overview of quality assurance-related activities in open-source software repositories and allows for comparisons with other projects. We develop a prototype of a QAA dashboard, RepoInsights, and evaluate it through a pretest-posttest study including semi-structured interviews with 14 software engineers from open-source, industry and academia. Our study is guided by the following research questions:

RQ1: How do software engineers apply QA practices and how do they reflect on them?

RQ2: What are software engineers' perspectives on a quality assurance analytics dashboard like RepoInsights?

RQ2.1: In what ways did the QAA dashboard help software engineers reflect on QA practices?

RQ2.2: In what ways could the QAA dashboard be improved to help software engineers reflect on QA practices?

During our study, we first explore how the participants of our study currently follow and reflect on QA practices (**RQ1**). We let them judge the QA of two software projects with RepoInsights to observe their use of a QAA dashboard. Based on their satisfaction with the dashboard, their verbalized thoughts and the plethora of feedback they provided, we elicit software engineers' perspective on our QAA dashboard (**RQ2**) and characterize the ways in which RepoInsights succeeds in letting engineers reflect on QA (**RQ2.1**), and the ways in which the QAA dashboard can be improved (**RQ2.2**).

In summary, this study contributes:

- A novel, conceptual design of a quality assurance analytics dashboard, and our prototype RepoInsights

- Insights from software engineers on how they currently adopt and manage QA practices in their projects.

- Insights on the successful aspects of our design and recommendations on how to improve the dashboard, based on the feedback from software engineers.

4.2 RepoInsights: A Quality Assurance Analytics (QAA) Dashboard

In this paper, we introduce the concept of a software quality analytics dashboard, designed to enhance software engineers' situational awareness surrounding QA practices [146]. It gives a ***global overview*** of quality assurance practices by collecting information and giving metrics and details related to the state of testing, automated workflows [147], code reviews, and guidelines, in one place. It also provides detailed views showing specific information on, e.g., pull requests. Additionally, it incorporates a ***comparative aspect***, enabling users to view similar information and statistics across a larger set of repositories. The dashboard is intended for software engineers that contribute to or maintain a software project. It serves as a vehicle to reflect on how adequate their project currently applies QA practices.

4.2.1 The Global Overview & Comparative Aspect

The **global overview** is designed around two main concepts: (1) situational awareness and (2) the complementarity of quality assurance practices. *Situational awareness* theory, rooted in psychology, refers to "the continuous extraction and integration of information to form a coherent mental picture, and then use this information to direct future actions" [148] and is important for effective decision-making in many environments (e.g., health care, air traffic control, etc.) [51]. A prior study explored open-source developers' situational awareness [133] of quality assurance aspects within their projects. The study identified areas of deficiency and compared awareness across various quality assurance practices [112]. The complementarity of quality assurance practices highlights the fact that using them in conjunction can enhance the overall quality perspective while at the same time might also reduce quality assurance efforts [31, 58].

The **comparative aspect** of the dashboard lets software engineers evaluate the information provided for their project in comparison to *other* projects. It thus enables them to independently judge their project's QA by comparing metrics to the average of other popular and active software projects.

4.2.2 Repository Dashboard Page

The repository dashboard page on RepoInsights has four primary sections: Testing, Automated Workflows (AWs), Pull Requests (PRs), and Guidelines (see Figure 4.1). These four sections comprise the **global overview** of the project's quality assurance. Beneath each section, there is a collapsible segment that presents similar information averaged across a set of projects, providing the **comparative aspect** (④ and ⑥ in Figure 4.1).

The provided information, including metrics, statistics, and graphs for each section of the dashboard are as follows: **Testing**: displays available test coverage information ①

(Codecov[1] and Coveralls[2]) and provides a link to a detailed page with trend of test coverage over time ②. **Automated Workflows**: presents details regarding checks on commits of the latest 100 PRs ③, including: top failed checks (number of failures and their associated GitHub App handle), top-used GitHub Apps, success rate of checks, average checks runtime, and average number of checks per commit. **Pull Requests**: displays information on the last 100 PRs, including the average number of line additions and deletions, average number of files changed, average number of participants in PRs, and average review time. Additionally, it provides a link to a details page ⑤ with the following: top active authors, distribution of pull requests' authors and mergers, pull requests with the highest number of comments, and trend graphs for review time, line/file changes, and comments per pull request, along with a detailed table with pull request information. **Guidelines**: indicates the presence of contributing guidelines, code of conduct, and templates for issues and pull requests. If any of these guidelines exist, the dashboard page provides a link to them.

4.2.3 Prototype Architecture Design

We adopt an engineering research approach, in which we develop a tangible prototype of a QAA dashboard. This prototype facilitates our study, enabling us to gather feedback on the concept of RepoInsights from software engineers.

The RepoInsights prototype follows a hybrid software architecture including a MERN[3] stack [149] for the app itself and a set of Python scripts that work as crawlers to collect the data through APIs of GitHub[4] (PR details, repository information, automated workflows, and guidelines), Coveralls[5], and Codecov[6] (test coverage information). The collected data is saved to MongoDB, retrieved by the MERN app to calculate statistics, and then display on the dashboard.

[1] `https://about.codecov.io/`, last visit: April 11, 2024
[2] `https://coveralls.io/`, last visit: April 11, 2024
[3] MongoDB, Express.js, React.js, and Node.js.
[4] https://docs.github.com/en/graphql, last visit: April 11, 2024.
[5] https://docs.coveralls.io/api-introduction, last visit: April 11, 2024.
[6] https://docs.codecov.com/reference/overview, last visit: April 11, 2024.

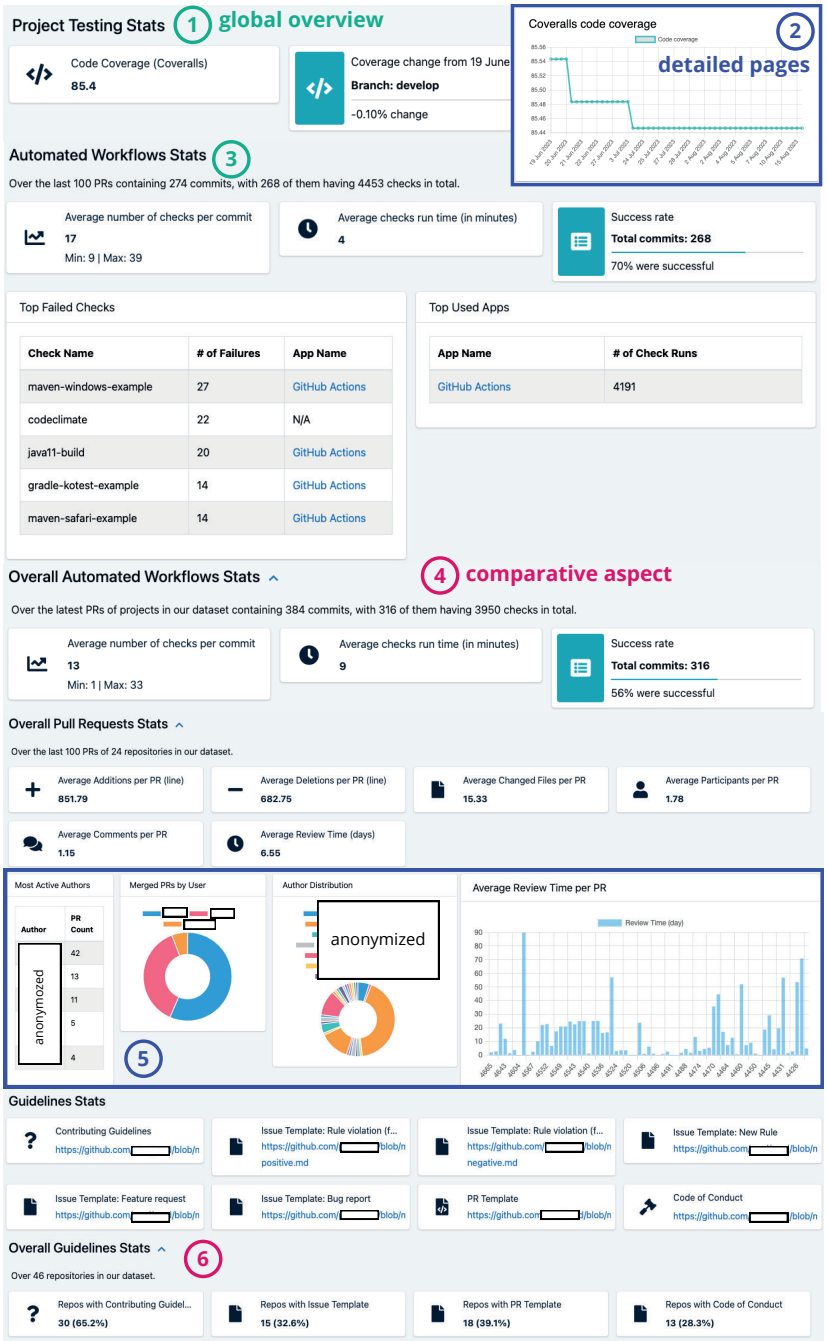

Figure 4.1: An illustration showcasing various parts of the RepoInsights dashboard.

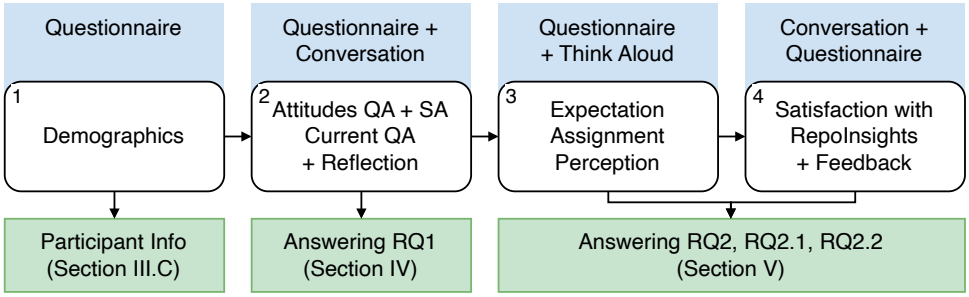

Figure 4.2: Structure of our interview sessions.

4

4.3 Study Setup

To evaluate our concept of a QAA dashboard, we follow a one-group pretest-posttest design with semi-structured interviews involving software engineers. We discuss how they currently reflect on quality assurance in their projects to get more insights on how a QAA dashboard can facilitate this current state (**RQ1**). To elicit concrete feedback on and suggestions for improvement for our design of a QAA dashboard, we developed the prototype RepoInsights. During the interviews, we ask the participants to judge two projects with RepoInsights, while thinking aloud. Then we ask them to evaluate different aspects of the tool, provide feedback, and sketch their potential usage. Through analyzing the participants' ratings and comments, we elicit software engineers' perspectives on a QAA dashboard like RepoInsights (**RQ2**). We are interested in the ways RepoInsights and our general concept of a QAA dashboard help participants in reflecting on QA practices (**RQ2.1**), as well as identifying the ways it can be improved (**RQ2.2**).

4.3.1 Study Design

We developed a questionnaire to guide our interview sessions. The steps of the interview align with the sections of the questionnaire and are shown in Figure 4.2. The questionnaire starts with informed consent and the participants' demographics, including questions about their experience in software development and whether they primarily work in open source, industry or academia (Step 1, Figure 4.2).

In the next step, we ask our participants to rate statements regarding their attitudes towards quality assurance and software analytics. To ensure clarity, we also provide a definition along with examples for each concept. Understanding our participants' attitudes is important, as these could influence their evaluation of the tool they are about to interact with. Then, the interviewer asks about the concrete quality assurance practices they use in terms of testing, code reviews, automated workflows, and other guidelines. Additionally, the interviewer asks how they judge whether quality assurance practices are sufficiently used (Step 2, Figure 4.2).

The following step consists of a one-group pretest-posttest study design [150] with the participant interacting with the dashboard in between. This type of study is called quasi-experimental to indicate that it does not meet the scientific standards of experimental design [151], yet it allows reporting on facts of real user-behavior, even those observed

in under-controlled, limited-sample experiences. First, we provide an explanation of our design of a QAA dashboard and ask the participants to rate statements regarding their expectations of such a tool (Step 3, Figure 4.2). After they complete the assignment with RepoInsights, we present the same set of statements and ask the participants to rate them based on their concrete experience with the dashboard. With this we gauge the participants' perception of the dashboard.

To encourage participants to engage with the dashboard, we devised a concrete task to solve during the assignment: *"assess two real-world open-source projects from GitHub across four quality assurance aspects: testing, automated workflows, pull requests, and guidelines."* Additionally, the participants were asked to judge the projects overall. The two projects, `fluentlenium/fluentlenium` and `pmd/pmd`, were carefully selected from our initial dataset of selected OSS projects on GitHub. This dataset included active and popular projects with ample data on pull requests (sampled using GitHub Search [88]), automated workflows, and available test coverage information. The two projects were chosen to have varied code review activity and contributor counts, enabling comparisons and observations of participants' reactions to different quality assurance levels. During the assessment, participants were encouraged to narrate their thoughts and provide reasons behind their judgments, with the interviewer actively prompting them for further explanations.

In the last section of the interview, we ask the participants to rate their satisfaction with the dashboard, provide feedback on how it can be improved and whether they would use such a QAA dashboard in their work (Step 4, Figure 4.2).

We obtained approval from our local ethics board and asked for informed consent to use and share the collected data for research purposes from the participants. For access to our full questionnaire, please refer to our provided artifact [152].

We conducted a pilot round of two interviews to estimate the interview duration and to identify any unclear questions. As a result, we increased the Likert scale range with 2 points for two questions and resolved an issue regarding the questionnaire's user interface. Subsequently, we requested the pilot participants to review and update their responses to the modified questions, allowing us to incorporate their revised answers into the final data set.

4.3.2 Study Execution

We recruited 12 study participants through e-mails to software engineers who expressed interest in follow-up research after participating in a previous study [144]. In addition, we recruited two more participants through convenience sampling in our professional networks to ensure a balanced representation of expertise in open-source software (OSS), industry, and academia.

We conducted the interviews via Zoom sessions, granting participants remote access to the interviewer's computer to interact with both the questionnaire and the dashboard. We recorded the interviews to be able to transcribe them for the qualitative analysis. The first author, who conducted all 14 interviews, took notes during the interview, especially during the open conversation regarding the participants' current quality assurance practices (Step 3, Figure 4.2). We also surveyed participants to assess their satisfaction with the interview, including questions, assignments, and the overall experience. Responses indicated either a high or medium level of satisfaction.

Participant	Software Dev. Experience	First GitHub Contribution	Academia	Industry	Open Source	Primary Role Projects	University Education	Age range
P1	7 to 10 y.	3 to 7 y. ago	x	-	x	Maint.	PhD	25-34
P2	3 to 7 y.	1 to 3 y. ago	x	-	-	Maint.	PhD	25-34
P3	3 to 7 y.	3 to 7 y. ago	-	x	x	Maint.	BSc	18-24
P4	3 to 7 y.	3 to 7 y. ago	x	-	x	Maint.	BSc	18-24
P5	3 to 7 y.	3 to 7 y. ago	-	x	x	Maint.	MSc	25-34
P6	10+ y.	7 to 10 y. ago	-	-	x	Contrib.	-	35-44
P7	10+ y.	7 to 10 y. ago	x	-	-	Maint.	PhD	55-64
P8	10+ y.	7 to 10 y. ago	-	x	x	Maint.	-	45-54
P9	10+ y.	10+ y.	-	x	-	Maint.	BSc	35-44
P10	10+ y.	3 to 7 y. ago	x	-	-	Contrib.	PhD	55-64
P11	10+ y.	3 to 7 y. ago	-	x	x	Maint.	BSc	35-44
P12	10+ y.	3 to 7 y. ago	x	-	x	Maint.	PhD	35-44
P13	10+ y.	10+ y. ago	x	-	x	Maint.	PhD	25-34
P14	3 to 7 y.	7 to 10 y. ago	-	x	-	Maint.	PhD	25-34

Table 4.1: Participants demographics.

4.3.3 Participants

A summary of the participant demographics is in Table 4.1. The majority of participants (85%) considered themselves to be in a maintainer role in the software projects they are involved in. Our participants represented a wide age range (~20 to ~60 years old) and mostly had education in computer science or related fields. They are often actively involved in OSS projects (64%), alongside working in academia or industry. One participant is exclusively working in OSS.

4.3.4 Data Analysis

The data analysis from the interviews was two-fold. We collected data from the questionnaire that the participants filled out during the interview, and applied open and axial coding [153] to analyze the recordings and transcriptions of the interviews. The first two authors started by analyzing one interview separately, and then discussed their experiences to set guidelines for how to assign open codes to any interesting observations from the interview transcripts. Then each of the two analyzed half of the interviews separately before merging the codes together and grouping them into higher level codes. The authors discussed the emergent groups, employing constant comparison [154] with the original transcripts, until they reached a negotiated agreement [155]. These higher level codes, together with the data collected through the questionnaire, are the basis for our answers to the research questions. In the following section, each statement based on our codes is followed by parentheses clarifying which participants made statements that led to this observation. We publish the full codebook as part of our replication package [152].

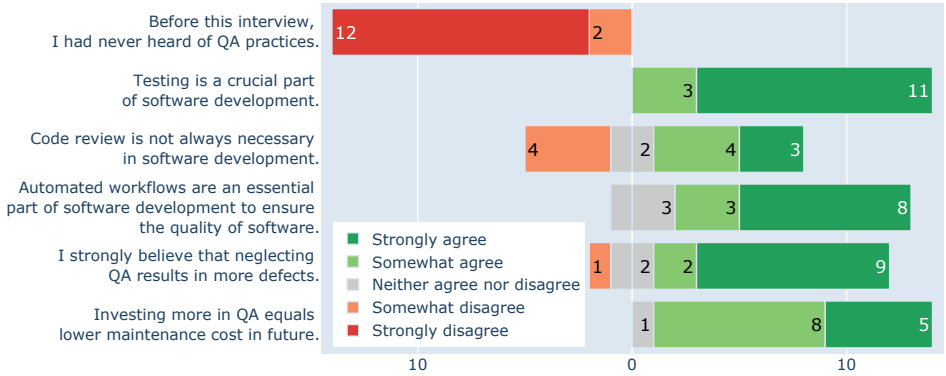

Figure 4.3: Attitudes towards quality assurance practices.

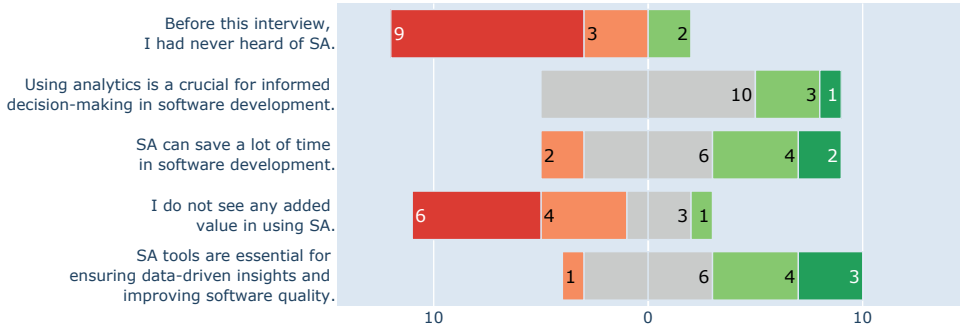

Figure 4.4: Attitudes towards software analytics.

4.4 RQ1: How Do Software Engineers Apply QA Practices and How Do They Reflect on Them?

In this section, we summarize our insights from the first part of the interviews, executed as part of the pre-test. We asked about the participants' attitudes on QA practices, software analytics, and their current approaches for QA. We delved into how they assess the adequacy of their approaches and explored their motivations for extending QA efforts. Based on the high-level codes that arose during the open and axial coding of the interviews, we answer our first research question (**RQ1**).

4.4.1 Attitude Towards Quality Assurance and Software Analytics

Overall, our participants are aware of and positive towards the value of quality assurance. They are aware of software analytics but the majority does not make use of it in their projects, and they are neutral to positive about whether it is necessary for a successful project.

The answers to the questionnaire on QA are shown in Figure 4.3. All participants were

familiar with the term "quality assurance" prior to the interview. There was unanimous agreement on the importance of testing. However, opinions differed regarding the necessity of code reviews, with half of the participants expressing either a moderate or high level of agreement that code reviews may not always be required. Concerning the impact of quality assurance, most participants held positive views about its ability to reduce defects and maintenance costs.

The participants had a mostly positive view of software analytics. They were neutral and leaning towards positive regarding its importance and necessity (Figure 4.4). When asked if they used software analytics in their projects, five participants (36%) said yes. These participants gave examples of using custom scripts and internal tools for software analytics. Four participants reported checking these analytics daily, while one participant reported checing them every two weeks.

4.4.2 Well Justified Variety in Applied QA Practices

The most prominent insight from our interviews is the variety in adopted QA practices. This variety is so large that reporting the single practices goes beyond the scope of this paper. Nevertheless, we want to give a few examples.

With regards to adding tests, P2 explained that in their projects the responsibilities for testing lies with the individual developers and there is no particular enforcement of adding tests. In P11's project only the single maintainer adds tests to the project, while in P4's project the pull request reviewers are responsible to add tests. P13 shared that the level of testing in their projects strongly depends on how easy it is to test the application they are developing. While a library has defined inputs and outputs and can easily be unit tested, for applications that depend on a lot of other interfaces not under P13's control they often opt for "production debugging", i.e., seeing in production if something goes wrong and fixing it.

With regards to automated workflows, several participants mentioned using GitHub Actions (P3, P8, P9, P11, P12, P13, P14). Some combine this with other continuous integration services (P3, P11) or set up their own build infrastructure (P14). P3 mentioned that they have a custom solution for continuous integration, as *"It's nothing we can find on the market. So the framework is customized to our own need."*

When talking about code reviews, four participants (P1, P3, P5, P12) stressed the importance of following strict code review guidelines, insisting on never skipping them, even in urgent situations. On the other hand, P2 mentioned that they sometimes skip code reviews *"for very small PRs or for very small changes or fixes."*

Discussing contribution guidelines, we learned that while some projects have guidelines in the "contributing.md" file in the repository (P2, P3, P7, P9), others have them on the website of the project (P4), or keep the detailed guidelines only with their internal QA team (P3). There are also ways of communicating guidelines beyond documents, e.g., in meetings (P1, P2, P10), through encouraged discussions on GitHub pull requests or issues (P4, P6, P9), in emails (P2), or in the online chat groups of the development team (P6). Several participants reported to not have publicly documented contribution guidelines (P1, P8, P10, P11, P13), e.g., because a project is closed source (P10), or has only very few contributions (P8). P9 explains that they intentionally avoid being overly strict during code review to maintain a more welcoming experience for first-time contributors [76].

4.4.3 Reflection on Applied QA Practices

When we asked the participants how they evaluated their state of quality assurance practices in their projects, we only got concrete answers to the testing aspect. Six participants use coverage metrics to judge the adequacy of their test suites (P1, P2, P5, P8, P10, P13). Other ways to judge tests were *intuition* and *exploratory manual testing* (P13), the reviewer's opinion (P4, P5, P10), or whether users report bugs (P9).

For automated workflows, code reviews, and contribution guidelines we could not draw out any comments from our participants judging the adequacy of these practices. Possibly, this is connected to the lack of available developer resources.

4.4.4 Motives to Extend QA Practices

While it was difficult to elicit how the engineers reflect on the adequacy of their quality assurance approaches, they more easily shared their motives to extend these practices. This could be seen as an early, pragmatic way of reflection, focused directly on concrete improvements.

The software engineers shared various motives to add new or more tests namely: based on new issues, failures, bugs, or user/customer reports (P3, P4, P5, P6, P7, P8, P11, P13, P14); new features, functionalities, or changes (P2, P3, P4, P7, P10); criticality or complexity of a component (P1, P2, P5, P12); missing coverage (P5, P10); and when higher-level tests failed to identify the root cause of a failure (P3, P5).

Three participants mentioned rarely updating or adding new automatic continuous integration workflows in their projects (P4, P5, P9). Instead, they extend the components executed by the workflows, e.g., by adding new tests to the test suite. When updates to automated workflows occur, they typically stem from new needs, driven by changes in the project's technology (P3, P4, P6, P9, P13). Having new issues or failures may prompt changes in workflows to help in pinpointing the root cause of problems (P3, P6).

Updates to the automated workflows are motivated by a need for specific information (P3) or personal inspiration from other open source projects (P1).

Three participants gave motives for updating their contribution guidelines such as user reports about problems (P6) or outdated information (P1), or when the project changes (P3).

RQ1 Summary

In general participants are positive about QA and know about SA but only a few actively use SA. There is a well-justified variety in the concrete QA practices that projects follow. The concrete usage of QA practices is motivated by the project's goal, context and available resources. A reflected judgment on the adequacy of current QA practices is not common, except testing. Extending QA practices is mainly based on concrete needs.

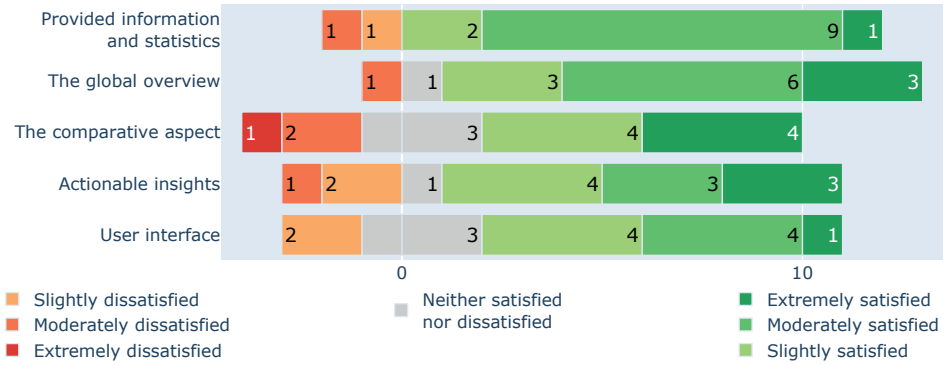

Figure 4.5: Participants' satisfaction levels with RepoInsights.

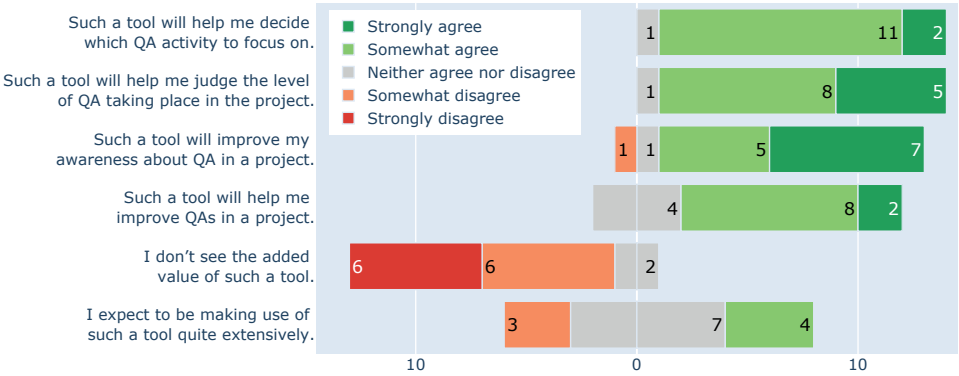

Figure 4.6: Participants' expectation of a quality analytics dashboard.

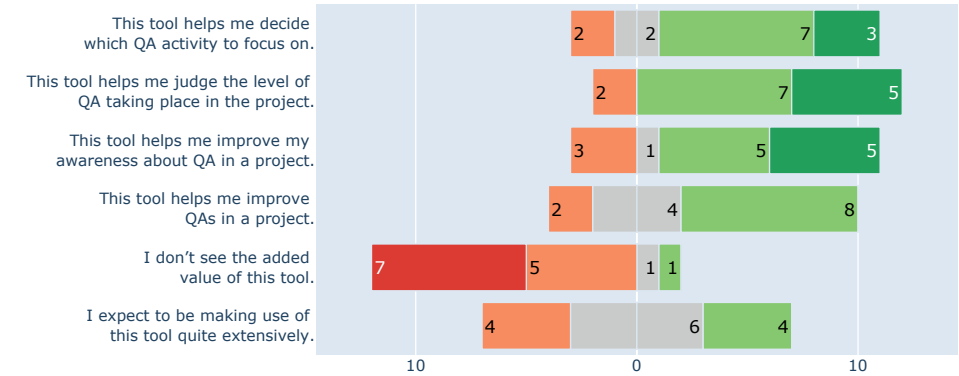

Figure 4.7: Participants' perception of RepoInsights.

4.5 RQ2: What Are Software Engineers' Perspectives on a Quality Assurance Analytics Dashboard Like RepoInsights?

To gauge our participants' perspectives on a quality analytics dashboard, we report their expectations and actual perceptions of RepoInsights recorded in the pretest-posttest design. We also characterize the situations in which participants said they would use a QAA dashboard. In Sections 4.5.2 and 4.5.3 we delve into the qualitative analysis of the interviews to provide a deeper understanding into how RepoInsights enabled software engineers reflect on QA and how it could be improved.

Looking at Figure 4.5, we see that the participants were generally satisfied with the different aspects of RepoInsights. The comparative aspect, received the worst ratings, which aligns with our observation that it was also used the least during the assignment.

Regarding the participants' expectations of a QAA dashboard based on our provided description, we see in Figure 4.6 that they are quite positive about the core intentions of the tool. These encompass helping to decide which QA activities need more focus, judging the level of QA, and improving (the awareness about) QA in a project. The participants see added value in the tool, but are divided on whether they would extensively use it. When we compare this to the perception of the tool after the assignment, we see in Figure 4.7 that the participants' summarized attitudes did not change much. They lean slightly more critical than before the assignment. This matches with our observations that the participants started to reflect very concretely on the dashboard and its value, opening up many concrete wishes for extensions of the originally described idea of a QAA dashboard.

4.5.1 Usage

Overall, all participants were positive about using a dashboard like RepoInsights in the future. In terms of frequency of using a QAA dashboard, three participants indicated that they would use it periodically when they are already reflecting on the health of their project (P4, P9, P11). Others mentioned one-off use cases, such as checking the QA status of their project once (P13, P1). As P2 indicates: *"[the tool] helps as a starting point to look into [QA practices]."* P13 points out that the frequency in which new information is available on the dashboard influences how often they would use it.

The role of the software engineer and their habits also play a role here. P14 indicates they are not a person who *"usually looks at dashboards"* but would use it if they were a QA engineer. P6 indicates that it would need to become a habit first, and P14 expect that the main maintainer of a project would use RepoInsights more. Two participants feel already quite aware of the state of QA in their projects (P7, P9), which limits additional value.

The participants mentioned further purposes they would use the dashboard for, beyond assessing the QA of their project. P13 suggests to use the QAA dashboard to judge the processes of open source projects they consider as dependencies, P4 proposes troubleshooting release issues such as adapting to a major platform change. P2 mentions inspecting other projects to evaluate their responsiveness to contributions, or to get inspired by their QA tools and workflows to use in their own projects.

4.5.2 RQ2.1.) In What Ways Did the QAA Dashboard Help Software Engineers Reflect on QA Practices?

We address this research question by observing how participants interact with the dashboard while completing the assignment. We identify the specific aspects of the dashboard that supported them in reflecting on QA practices.

Displayed metrics

The participants mainly used the dashboard's global overview provided metrics to reflect on QA practices and make judgments about them.

For rating the level of code reviews, seven participants pointed to the average number of discussion comments in pull requests (PRs) (P1, P2, P4, P5, P9, P10, P12). Getting more comments was referred to as positive sign for code reviews. On the other hand, not receiving comments or having few comments was a sign of not having reviews or just a quick code review *"P02: ...no comments in the PR that I would assume the code review was not done or it was not it was just a quick code review..."*. Also, seven participants used the provided review time of PRs, which is the average time it takes to review and merge PRs after their creation (P2, P5, P6, P10, P11, P12, P14). Similarly, six participants referred to the reported average number of participants in PRs (P2, P3, P9, P10, P12, P13), for instance: *"P13: my reasoning is that there are a lot of contributors per single pull request. There's the quality of reviews."* Six participants took the displayed average change size of PRs into account while making their judgments (P1, P3, P5, P7, P10, P13). This change size includes number of added and deleted lines per PR, and number of changed files. Participants expressed judgments such as the difficulty of reviewing PRs with big changes and how the size of changes provides insight into the stage of development in a project, *"P03: we are seeing a lot of additions and deletions at the same time. That means the project is very much in an exploratory stage or it is in active development"*. Moreover, having a high number of additions and deletions was suspected to be fake and misleading, based on previous experience with GitHub's diff, or also suspected to be from automatic changes, e.g., auto-fixing the code style.

Regarding reflection on the automated workflows (AWs), five participants referred to the number of checks (P1, P2, P3, P4, P11). When the average number of checks is high, participants anticipated thorough testing (P11), overloading and confusing contributors (P3), and increasing difficulty in identifying failing checks (P3). Moreover, P3 viewed an excess of checks as a sign of an exploratory project and pointed out that projects should not have multiple checks for the same purpose. Other metrics used in reflecting on AWs were the reported AWs success rate and the number of failures (P8, P9, P12). The software engineers were generally optimistic regarding an average failure rate of 30% in both projects of the assignment.

When reflecting on testing, five participants based their evaluation merely on the provided code coverage percentage (P2, P3, P7, P10, P13). They considered 85% to be a high percentage for the test suites.

Meaningful details

Additional details shown in the dashboard allowed participants to gain a deeper understanding of the context of the projects, enabling them to make more informed judgments.

Regarding judging code reviews, four participants referred to the detailed information on the PR details page (P7, P9, P10, P13). P13 specifically noted the presence of Dependabot-

created PRs in one project, which could skew the data. P7 found the outlier PRs appearing on the PR trend graphs to be another point of interest, which have touched thousands of lines while other PRs have mostly changed a few lines.

When judging automated workflows, five participants noted the names of the top failed checks in AWs overview of the dashboard (P4, P7, P8, P9, P13). This detail helped them in gaining a better understanding of the purpose of AWs in projects, which they could then relate to other information such as average number of checks and their runtime. *"P09: [while judging AWs, based on check names] ... they're clearly testing on three platforms."*. Another example of judging details of AW failures was when P1 compared test failures and build failures, expressing skepticism regarding build failures, while test failures were deemed more acceptable, as contributors might not run all tests locally.

Regarding assessing tests, three participants made specific judgments using the code coverage trend graph (P9, P10, P13). They noted that a decreasing trend in code coverage in an active project could indicate concerns about code quality.

Providing the global overview

When reflecting on various QA practices, participants primarily took into account multiple metrics and details related to the projects' QAs. While this might seem straightforward when focusing on individual practices and metrics, our emphasis lies in considering information about multiple QA practices' metrics. The global overview highlights the complementary nature of various QA practices: *"P05: ... So I think a tool that integrated so many statistics and it really helps."*; *"P02: Having everything in one page, like the code review guidelines, I think that's nice"*; *"P01: I really like the fact that you have an overview of possibly all the different things, I like that it's combined."*

Providing visible links

When reflecting on the guidelines of projects, five participants relied solely on the indication of their existence (P2, P3, P7, P8, P12). However, P7 and P8 expressed interest in seeing the content of the guidelines to make more accurate assessments.

We had included links to projects' available issue and PR templates, as well as their code of conduct and contribution guidelines. Initially, at the start of the assignment, we encouraged participants to base their judgments about projects' QA solely on the information provided on the dashboard. However, despite this guidance, P7, P10, and P13 clicked on the links to inspect the content of these files and evaluate their quality.

For a global overview of the project, consolidating all information in one place, particularly raw, unprocessed data, may not be necessary. Instead, incorporating visible links allows access to relevant information, facilitating a thorough understanding of project details and context without overwhelming users with excessive data (e.g., content of guidelines, configurations details, build logs, etc.).

RQ2.1 Summary

Participants reflected on projects' QA practices using: the displayed metrics regarding projects' QA practices, the provided meaningful details to get a deeper understanding of projects' context, the global overview aspect, and the visible links to plain data.

4.5.3 RQ2.2.) In What Ways Could the QAA Dashboard Be Improved to Help Engineers Reflect on QA Practices?

We collected and qualitatively analyzed comments, suggestions, and feedback about RepoInsights provided by participants during the interviews. In this section we present the result of this analysis as a set of improvement points.

Providing high granularity of information

Eight participants suggested increasing the granularity of information for various QA metrics and statistics (P1, P3, P4, P7, P10, P12, P13, P14). Specifically, they highlighted the necessity of higher granularity to identify areas for improvement and make informed decisions about where to focus efforts. *"P01: ... you need this granularity to really see where you're lacking."*; *"P03: information is very aggregated. I would expect it to be somewhat detailed."*

Regarding tests, five participants highlighted the need for detailed information to properly reflect on testing (P1, P7, P10, P12, P13). They expressed that the provided code coverage and its trend alone were insufficient. They recommended incorporating higher granularity for the provided information, including detailed granular coverage of different submodules or subsystems, highlighting parts with missing coverage, and the tests contributing to the coverage of each module.

When reflecting on code reviews, five participants suggested more details, including: acceptance rate of PRs, fine-tuned change information rather than merely additions and deletions per PR, distinguishing different types of comments in PRs and providing a distribution of where PRs touch the code (P3, P4, P7, P13, P14).

In discussions about automated workflows, P1 suggested providing insights into the use of automated workflows to indicate if checks are mandated during the development process. Moreover, P14 suggested adding the date of the last update of automated checks and providing recommendations for updating them in response to major changes in a project. Additionally, P13 and P14 recommended higher granularity in AWs by providing detailed breakdowns of their purpose.

Regarding guidelines, two participants suggested providing more details: P7 suggested analyzing the quality of guidelines through keyword matching or using AI-based analysis to assess their completeness. Additionally, P13 proposed analyzing the differences between issue or PR templates and what users submit, particularly for open-source projects that receive numerous issues and PRs.

Providing additional data insights

To have a deeper understanding of what is happening in the projects, participants suggested incorporating new data insights and statistics. While these suggestions may not directly

relate to project QA, they are vital for contextualizing projects and facilitating more insightful reflection: **Additional data regarding tests**: providing more data insights such as adding new metrics for test coverage such as mutation score, number of tests, proportion of commits changing project's tests, lines of added code versus tests, and number of executed/failed tests (P4, P8, P10, P13). **Trends analysis**: incorporating trend analysis to visually depict the evolution of metrics over time. While this feature was provided for certain metrics, such as PR activities and code coverage, participants suggested extending it to include other relevant metrics (P7, P10, P13, P14). **Projects' age and goals**: are related to the availability of guidelines (P1, P3, P7, P9, P11). **More charts and metrics**: offering statistical charts and median values for presented metrics to mitigate the skewness caused by outliers (P1, P3, P13). **Number of contributors**: the intensity of QA practices in projects can depend on the number of contributors (P3, P4, P13). **Development processes**: a long review time can be acceptable if multiple sets of time-consuming checks need to be completed, and that having multiple workflows does not necessarily mean that they are used and enforced in the PRs; so depending on the development process reflection on the QA can vary (P1, P3). **Metrics definition**: adding extra information about the definition of all provided metrics to avoid confusion and enable a more accurate reflection (P4, P9). **Project Releases**: incorporating information about project releases, including the usage of automatic releases, to assess project activity levels and indicate robust QA practices (P13). **Project Activity Overview**: offering an overview of recent project activities to show its current level of engagement (P13). **Issues Statistics**: introducing statistics for issues similar to PRs, such as their waiting duration and topic, to provide a comprehensive view of project maintenance (P4).

Providing meaningful comparison

Participants recommended enhancing the comparative aspect by: **Custom comparison**: three participants suggested adding the feature to select specific projects or groups of projects for comparison (P1, P5, P10). **Comparison against similar projects**: two participants (P1, P9) suggested having clusters of similar projects based on their attributes, such as programming language. Another participant suggested the ability to compare against projects within the same GitHub organization.

Providing the context of projects

Six participants expressed concern about their lack of familiarity with the context of projects while working on the assignment (P1, P3, P9, P11, P13, P14). *"P13: But probably the developers of those projects will have a different perspective because they know what's happening in their projects."* This context encompasses various factors such as the type of software project, its requirements and goals. Additionally, it involves considerations like available developer resources. Furthermore, more details about the configuration of automated workflows, and the acceptance of outside contributions are (e.g., OSS) or are not (e.g., companies) important. Understanding the context is also crucial when making comparisons with other projects. It is more logical to compare similar projects to gather insights about improving QA practices.

One explanation for this emphasis of our participants on providing the project context is that they were not familiar with the projects we asked them to judge. Even if users are working on projects they know well, providing sufficient context can enhance their

understanding. Beyond that, users may have varying levels of familiarity with different aspects of their projects—precisely what RepoInsights tries to address regarding QA—and providing contextual information can help bridge gaps in their knowledge.

Regarding reflection on guidelines, four participants provided specific judgments when no guidelines were available (P3, P4, P5, P15). For instance, they deemed it acceptable if it was a single-developer project, interpreting the absence of guidelines as an indication of only one developer. They also inferred that potential contributors would not know how to contribute to the project without guidelines.

Providing actionable insights

Nine participants gave feedback regarding the need to make insights of the dashboard more actionable (P1, P2, P4, P5, P9, P10, P12, P14). More specifically, four participants (P1, P3, P5, P12) pointed to the need of more detailed concrete suggestions to improve actionability of dashboards insights: *"P12: That's what I'm doing is like, you know, looking at these statistics, it might tell you that you need to take some action, but it's not clear what action is required."*. Moreover, they mentioned that insights regarding some QA practices are more actionable, *"P12: I can see the code coverage and that's something that I may need to implement some action on. I can see whether or not there are actual contributor guidelines. That's something that can be actioned on."*, there are other areas that need to be more actionable by suggesting how to improve QAs, for example P12 who mentions that a low number of reviewers during code review is useful to know, but does not provide a solution.

Providing the ability to filter (out) information

Four participants recommended adding filtering capabilities to include relevant information and exclude irrelevant data. (P7, P8, P9, P14). The dashboard includes various QA-related data to provide a global overview, however, depending on the project's context, and on the user's preferences, it would be useful to filter out unwanted information. Examples are: Dependabot or any other bot-created PRs that might skew the data, or unimportant build failures that makes it difficult to see what really matters.

Not all projects are interested in all information equally. While the default dashboard view intentionally shows a variety of QA practices to inspire adopting them, users should be able to hide information they do not find necessary for the future times they consult the dashboard. There exists a wide range of approaches to QA practices across projects. Certain projects prioritize extensive testing of complex software components during development (P1), while others solely rely on end-user tests and reported issues for post-production testing (P14). Additionally, code coverage metrics hold high importance for some projects (P1, P3, P8), while in others, developers lack awareness of this concept (P6). These differences underscore the need for a customizable QAA dashboard to accommodate diverse project goals and perspectives.

RQ2.1 Summary

Based on participants suggestions and feedback, providing the followings can help them in better reflection: high granularity of information, meaningful comparison, context of projects, actionable insights, and ability to filter (out) information.

Concrete feedback and suggestions: We also received a plethora of feedback and suggestions related to concrete enhancements of the dashboard's user interface and user experience, such as explanatory texts on how the metrics are calculated and what data they precisely include. We documented these points to refine our prototype in future iterations. As this information is mainly specific to the implementation of our prototype, it falls outside the scope of this chapter. However, toolmakers and researchers keen on developing similar software analytics tools can access this information through our provided replication package [152].

4.5.4 RQ2. Summary Summarizing Software Engineers' Perspectives on RepoInsights

The participants were generally satisfied with RepoInsights, particularly its global overview, provided information, and statistics. Their perception of RepoInsights closely alined with their positive expectations of a QAA dashboard. Most participants expressed interest in future use. From their comments, we formulated four insights on what was successful in the design of RepoInsights as well as six suggestions to improve it.

4.6 Related Work

Quality assurance practices in software engineering have been studied both individually and in combination [58]. Studies on complementarity of quality assurance practices, such as software testing [8, 16, 77, 113], modern code reviews [15], automated workflows [147], build automation [21, 22, 24], and automated static analysis [18], have shown that QA techniques are mostly *not* used in conjunction [58] and software engineers' **situational awareness** regarding them is lacking [112]. In this study, we aimed to overcome this awareness gap [146] with a tailored software analytics dashboard [145].

For more than a decade [156], **software analytics** has played a significant role in enabling software engineering individuals to make informed, data-driven decisions across various facets of software development. Applications include quality assessment tools such as SonarCube[7], Kiuwan[8], and Bitergia[9], as well as those tailored for DevOps environments like New Relic[10], and Datadog[11]. Moreover, research endeavors have contributed to this domain, yielding open-source tools such as GrimoireLab [157], a comprehensive toolset for software development analytics, SQuAVisiT [158], which offers flexible visual software analytics capabilities, Q-Rapids [159], facilitating continuous quality assessment throughout the software development lifecycle, QaSD [160], a quality-aware dashboard designed to enhance software quality and development processes, and QConnect [161], a quality-aware dashboard focused on developers' activity and productivity.

The highlighted tools above, underscore the multifaceted impact of software analytics on software engineering. However, none of them target QA practices. Our goal is to fill this gap to potentially enhance the situational awareness [133] surrounding QA practices and empower software engineers to make informed decisions regarding QA of their projects.

[7] `https://www.sonarqube.com/`, last visit: April 11, 2024
[8] `https://www.kiuwan.com/`, last visit: April 11, 2024
[9] `https://bitergia.com/`, last visit: April 11, 2024
[10] `https://newrelic.com/`, last visit: April 11, 2024
[11] `https://www.datadoghq.com/`, last visit: April 11, 2024

4.7 Threats to Validity

Reliability: to ensure consistency in our qualitative analysis, two authors coded half of the interviews independently and met repeatedly to discuss emerging codes until reaching a negotiated agreement [155]. However, other researchers' interpretations of the data may differ. We publish our full codebook to enable validation of our findings [152].

Respondent Bias: participants may have provided socially desirable answers when evaluating a tool developed by the interviewer. To mitigate this, we encouraged constructive criticism and maintained a welcoming attitude toward critical feedback. The wide range of improvement suggestions received suggests this threat was reasonably addressed.

Internal Validity: the open-ended interview format threatens reproducibility. Many key insights emerged from only a few interviews, so repeating the study may miss some perspectives.

External Validity: findings may not generalize beyond the specific QA practices, project context and participants we studied. We selected active and popular projects with ample data on PRs and AWs for the dashboard. We only included projects with available test coverage information. Additionally, we varied code review activity and contributor counts to enable comparisons and observe participants' reactions to different quality assurance levels. We openly recruited participants from industry, academia, and OSS, then, directly recruited two more from industry and academia to ensure diversity.

4.8 Implications and Future Work

Implications. While software engineers are generally positive about quality assurance (QA) and software analytics, our study revealed that reflection on the adequacy of QA practices is not common, except for testing activities (RQ1). From our sample of participants, and from prior work of Khatami and Zaidman [12], we know that there is currently no widespread adoption of analytics tools to specifically support reflection on QA practices across projects [112]. This represents an opportunity for toolmakers to increase software engineers' situational awareness surrounding QA practices and allow them to reflect holistically on their projects' QA activities.

Our evaluation of this idea, through a prototype of a quality assurance analytics (QAA) tool RepoInsights and the subsequent analysis of developers' perspectives, highlighted several key implications: **(1)** there is interest and perceived value in centralized quality analytics that provide a global overview of QA metrics and practices across projects (RQ2.1). Participants were generally satisfied with RepoInsights's ability to surface this higher-level perspective. However, **(2)** QAA tools must go beyond simply displaying metrics. Software engineers prefer high granularity, contextualized information tailored to each project's specific practices and needs (RQ2.2). Furthermore, actionable insights derived from data and presented to the user are essential. **(3)** Successful QAA designs should provide meaningful comparisons between projects, the ability to customize viewed information, and rich context about each project's QA activities (RQ2.2).

Overall, we hypothesize based on the insights of our study, that QAA tools presenting contextualized, actionable insights can empower data-driven evolution of QA strategies

[12] Chapter 3 of this dissertation.

as project situations change. Seamless integration into developer workflows is key for increasing adoption and reflection.
Future work. Key areas for future research to maximize the adoption and impact of QAA tools include: adopting and evaluating intuitive visualization techniques to derive tailored, actionable insights from quality data beyond just presenting raw metrics. Investigating integration pathways to embed QAA into developer workflows and toolchains. Evaluating effectiveness and validating QAA tools' by measuring impact on quality and productivity.

4.9 Conclusion

In this study, we set out to raise the situational awareness of software engineers surrounding quality assurance (QA) in their projects. Through a pretest-posttest study combined with interviews, we evaluated our conceptual design of a software quality assurance analytics (QAA) dashboard. We elicited the current approaches in adoption and reflection of QA practices from our 14 participants. We observed how our RepoInsights dashboard supported them, and we identified a set of recommendations for a dashboard like RepoInsights to empower software engineers to better reflect on QA. Our key insights are: reflection is uncommon, except for testing, and extending QA practices is based on concrete needs (RQ1). Participants valued the global overview of metrics/practices and the meaningful details that the QAA dashboard provided (RQ2.1). Desired features included high granularity information, meaningful comparisons, tailored insights, filtering capabilities, and more details to have a deeper understanding of projects' context (RQ2.2).

Overall, our results demonstrate interest in QAA tools that go beyond raw data & metrics to deliver highly granular, contextualized, and actionable insights. We hypothesize that such a QAA tool that takes our recommendations into account, can potentially raise situational awareness and facilitate reflection and improvement of QA practices.

5

Catching Smells in the Act: A GitHub Actions Workflow Investigation

GitHub Actions (GHA) are a way to automate CI/CD workflows within the GitHub platform. The deep integration of GHA into GitHub enables to automate a wide range of social and technical activities. In this study, we investigate workflow smells*, i.e., characteristics in the workflow that possibly indicate a deeper problem. Through a mining study, we first expose a list of frequent change patterns in the workflows of 83 GitHub projects. We then manually analyze these frequent change patterns to understand the negative effects that the frequent changes try to remove. To validate the list of 22 potential workflow smells that we thus obtain, we carry out a contribution study with 32 projects on GitHub through pull requests that contain a fix to the candidate smell. By qualitatively analyzing the maintainers' comments in 32 pull requests, we settle on 7 confirmed GHA workflow smells.*

5.1 Introduction

Continuous Integration (CI) is the software engineering practice in which developers not only integrate their work into a shared mainline frequently, but also verify the quality of their contributions continuously [21]. Continuous Deployment (CD) adds frequent delivery through automated deployments [162]. Both practices have become integral to collaborative software development to streamline the software delivery process and increase the quality of the delivered software [20, 24, 47, 58, 83, 163, 164].

To support CI/CD developers rely on dedicated platforms that provide the necessary infrastructure and tooling. Among the available platforms are Travis CI [66], Circle CI, Cirrus CI, Azure Pipelines, GitLab CI, and GitHub Actions (GHA). GHA enables software engineers to automate workflows directly from within GitHub repositories, attaching

This chapter is based on *A. Khatami, C. Willekens, and A. Zaidman. In IEEE 24th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 47-58. IEEE, 2024.* [].

triggers to various events like code pushes, pull requests [76], and releases. Since its introduction in November 2019, GHA emerged as the predominant workflow automation tool on GitHub [106]. GHA's integration with the GitHub ecosystem and open-source library of reusable workflow actions have been key contributing factors to its rapid adoption [164].

The growing adoption of GHA has attracted considerable research interest, with numerous studies empirically investigating its adoption [87, 106, 137, 138, 165, 166]. These studies have highlighted concerns related to security [138, 167–170], dependency management [164, 171], and resource usage [172] in GHA workflows.

Zampetti et al. identified bad smells for continuous integration [173], i.e., factors in continuous integration that create barriers or challenges when it comes to setting up, maintaining, or ensuring the high-quality outcome of a CI workflow. Additionally, we know that GHA workflow configurations may degrade in quality and accrue technical debt [55]. As such, given that five years have passed since the introduction of GHA, and a substantial commit history for GHA workflow configurations has accumulated, this study sets out to investigate whether GHA workflows suffer from *workflow smells*.

While earlier research has identified CI/CD workflow smells [173, 174], the unique proposition of GitHub Actions workflows warrants a deeper investigation into GitHub Actions-specific workflow smells. More specifically, the fact that GitHub Actions are more fine-grained in nature than typical CI/CD workflows, their wider range of use case scenarios (e.g., GHA are also applicable to issue management, or code reviewing), and the reusability and composability of GHA workflows may lead to different smells than those previously identified. Additionally, earlier studies have identified issues related to security [167, 170], and opportunities for optimization [172] in GHA workflows, but our investigation is set up to be holistic, also targeting maintainability and efficiency smells.

Our investigation is steered by four research questions.

RQ1 Are there common patterns of frequent changes in GHA workflow configurations?

To answer our first research question, we analyze the evolution of GHA workflow configurations over 10,012 commits in 83 projects, thus aiming to understand whether there are common patterns of frequent changes in GHA workflow configurations.

Next, we inspect the common change patterns in GHA workflow configurations, to determine whether some of these common change patterns could indicate a fix to a problematic workflow configuration. This leads to our second research question:

RQ2 Are frequent change patterns in workflow configurations indicators of *workflow smells*?

Once we have analytically established a set of smells, we turn our attention to their automatic detection in RQ3:

RQ3 Can we automatically detect GHA configuration smells?

Finally, to validate our list of candidate GHA smells, we conduct a contribution study in 32 open source GitHub projects by opening 40 pull requests that contain a fix to the candidate smell. Based on the developer's decision to accept or reject our contribution, and the feedback that the developer's provide, we aim to answer our fourth research question:

RQ4 To what extent do developers find the proposed fixes for identified GHA configuration smells relevant?

In summary, this paper makes the following contributions:

- A set of 7 frequently found and validated GitHub Actions workflow smells. One of these smells is completely unique to our study.
- A set of scripts designed to automatically detect these workflow smells.
- A contribution study that provides insight into the relevance and importance of the workflow smells.

5.2 Background

In their seminal blog post of 2000, Fowler and Foemmel advocate for Continuous Integration (CI) as a mechanism to ensure that integrating new code contributions does not break the build, and an integrated automated test suite to detects errors [175]. In a series of papers, Elazhary et al. have established that many companies follow CI practices, but with quite some variations in how they use CI [119, 132]. Both Vasilescu et al. [20] and Hilton et al. [24] highlight that CI improves productivity, leading to more PRs being processed, accepted and merged, PRs being accepted faster, and projects releasing a new version more often.

GitHub Actions. (GHA) is a continuous integration and continuous delivery (CI/CD) service that is tightly integrated into the GitHub platform [138] and has rapidly gained in popularity [106]. While GitHub Actions provides CI/CD functionality similar to other CI/CD tools, it is unique in that it enables to run workflows for several events that happen in a repository, e.g., code reviewing, communication with developers, verifying licence agreements, and monitoring and fixing dependencies and security vulnerabilities [176].

GHA Workflows. A *GHA workflow* contains one or more *jobs* and can be triggered based on different events in the repository. A *job* contains a list of *steps* which specify a command, for example a bash script or a *action*, reusable code that implements common CI/CD tasks. For *jobs*, the operating system on which they run can be defined as well as a *strategy* which allows the same job to be run with different inputs [172].

Security. Security is one of the five major challenges when automating workflows [169]. Poorly secured CI platforms can lead to code being stolen or injected, as well as bypassing code reviews resulting in potentially malicious code being added to the code base [138]. In order to eliminate security risks, four security properties have been identified for CI including: admittance control, execution control, code control, and access to secrets [170]. Koishybayev et al. investigated these security properties for GHA and found that none of them always hold, noting that these properties can be fixed through proper workflow

configuration [170]. Benedetti et al. created a tool to automatically identify security issues in GHA workflows [168].

CI/CD Smells. Previous research has investigated CI smells which affect the performance of the development cycle and CI pipelines. Duvall et al. identified 16 general patterns and anti-patterns for CI/CD [177, 178] Zampetti et al. expanded this catalogue by interviewing experts and analysing Stack Overflow[1] discussions [173]. A total of 79 (of which 44 new) smells were identified in 7 categories and evaluated in terms of relevance by professional developers [173].

Gallaba et al. was the first to define, detect, and remove configuration smells in CI. Firstly, a catalogue of four smells was defined for which they created a tool, *hansel* to detect these instances of smells for Travis CI. Through 49 pull requests with removed smells, they got 36 improved CI configurations accepted by the project maintainers [55].

Vassallo et al. investigated automatically detecting CI anti-patterns [179]. Their tool detects the existence of four relevant anti-patterns, and they found 3,823 instances of these smells accross a set of projects [179]. Vassallo et al. created a similar tool – *CD-Linter* – for GitLab [174]; it focuses on smells that are detectable through configuration files. This resulted in four candidate smells: fake success; retry failure; manual execution; and fuzzy version. Using their tool, 145 issues were opened that addressed these smells. They received a response rate of 74% with 53% of maintainers reacting positively to the smells.

5.3 Study Set-Up

Our study follows a bottom-up approach towards the identification of GHA workflow smells. In Section 5.4, we start by identifying common change patterns in workflow configuration files using a mining study (RQ1). Subsequently, in Section 5.5 two authors analyze these common change patterns to understand whether there are issues underlying the change patterns that ultimately led to the change. This analysis leads to a set of candidate workflow smells (RQ2). In Section 5.6 we investigate the feasibility and effectiveness of a set of custom-developed workflow detection scripts for these candidate smells (RQ3). Finally, in Section 5.7 we carry out a contribution study in which we file pull requests with fixes to the candidate workflow smells in GitHub projects. In observing the acceptance, rejection, and pull request comments, we aim to validate the candidate smells as recognized and relevant workflow smells (RQ4). Within Section 5.4 to 5.7, we first describe the methodology that we followed to answer each RQ, after which we describe the results.

5.4 RQ1: What are the common patterns of frequent changes in GHA workflow configurations?

RQ1 methodology: To discover potential smells in GitHub Actions (GHA) workflow configurations, we employed an exploratory approach by analyzing the evolution of the GHA configuration files in open-source software repositories.

We initially selected the 100 most popular repositories on GitHub, based on the number of stars received, using the SEART GitHub search service [88]. We selected the 20 most popular projects for five different programming languages: JavaScript, Java, C#, Python, and

[1] `https://stackoverflow.com`

Table 5.1: Number of changes in GHA workflow configuration files across projects, categorized by programming language.

Language	Number of projects	Number of commits	Number of changes
JavaScript	18	1,825	2,933
Java	15	519	856
C#	15	825	1,399
Python	15	3,105	7,665
Typescript	20	3,738	7,184
Total	83	10,012	20,037

TypeScript. This approach ensured a diverse dataset of changes to workflows that are not specific to any particular programming language. For each selected repository, we collected all commits that modified at least one ".yml" or ".yaml" file in the ".github/workflows" directory, if such a directory existed. Repositories without this directory were excluded, resulting in a final dataset of 83 projects. We treated each modified workflow file within a commit as a single change, enabling a granular analysis of workflow evolution and identification of common modification patterns. In total, we collected 10,012 commits for 83 projects. Each commit could contain multiple changes, resulting in a total of 20,037 changes across all projects (Table 5.1).

To analyze and categorize the changes in GHA workflows, we employed an open card sorting approach [180], allows categories to emerge naturally from the data without imposing pre-defined groups [140]. We treated each change as a single code. The second author performed the initial coding and categorization, which was then reviewed and finalized through a discussion between the first two authors, until reaching a negotiated agreement [181]. To accelerate the process, we developed automated scripts based on the observed patterns of changes, which automatically labeled and grouped changes according to the established categories. An additional round of manual verification was conducted to ensure the accuracy and reliability of the categorization. This approach allowed for efficient and accurate categorization while maintaining the integrity of the open card sorting methodology.

RQ1 answer: In our analysis we have identified 64 distinct types of frequent changes made to the GHA configuration files. We categorized these 64 types of changes into 8 higher-level categories, based on their purpose, displayed in Table 5.2. The most prominent high-level pattern *run step configuration* appears in 36.5% of changes, and can be found in 89.2% of the projects. We now discuss the categories of changes.

Run step configuration is the most prevalent pattern, encompassing changes such as run command updates, action configuration, adding or removing run steps, and updating run steps. We observed that developers frequently adjust their run steps to optimize execution, leverage GHA capabilities, and improve the efficiency of their CI/CD processes.

Dependency versioning primarily involves bumping versions and hash versions, as well as using hashes instead of versions. These changes highlight the importance of keeping dependencies and runtime environments up to date and ensuring the reproducibility of builds.

Table 5.2: Categories of common changes in the evolution of the GHA YAML configuration files. The complete table with more fine-grained categories of changes is available in the provided dataset[182]. Please note that the percentages of sub-categories in the "Occurrences" column are calculated relative to their parent category. However, the percentages of sub-categories in the "Projects" column are calculated based on the total number of projects.

Category	**Occurrences**		**Projects**	
Sub-category	#	%	#	%
Workflow Organization	**2,177**	**9.2%**	**83**	**100%**
Add/Remove workflow	1,907	86.5%	83	100%
Move/Refactor workflow	137	6.2%	33	39.8%
Add/Update workflow name, etc.	161	7.3%	33	39.8%
Run Step Configuration	**8,709**	**36.5%**	**74**	**89.2%**
Run command updates	3,395	39%	59	71.1%
Action configuration	2,741	31.5%	63	76%
Add/Remove run step	2,301	26.4%	60	72.3%
Update run step, etc.	272	3.1%	50	60.2%
Dependency Versioning	**4,508**	**18.9%**	**70**	**84.3%**
Bump version	2,160	47.9%	64	77.1%
Bump hash version	2,151	47.7%	19	22.9%
Use hash instead of version, etc.	197	4.4%	34	41%
Job Configuration	**1,159**	**4.9%**	**63**	**75.9%**
Add/Remove job	659	56.9%	45	54.2%
Matrix configuration	290	25%	44	53%
Update/Add job name, etc.	210	18.1%	37	44.6%
Environment Setup	**1,589**	**6.6%**	**45**	**54.2%**
Update env/env variable	1455	91.6%	43	51.8%
Update runs-on, etc.	134	8.4%	16	19.2%
Trigger Conditions	**1,973**	**8.3%**	**66**	**79.5%**
Update "on"	1,553	78.7%	65	78.3%
Prevent running on forks, etc.	420	21.3%	34	41%
Scheduling	**2,408**	**10.1%**	**28**	**33.7%**
Add timeout	2,254	93.6%	13	15.6%
Add/Update concurrency, etc.	154	6.4%	26	31.1%
Miscellaneous	**1,308**	**5.5%**	**70**	**84.3%**
Add/Update/Remove "if"	362	27.7%	34	41%
Add/Update/Remove comment, etc.	97	7.4%	27	32.5%
Access control configuration	373	28.5%	50	60%
Formatting and Styling	476	36.6%	57	69%
Total	**23,862**		**83**	

Workflow organization includes adding, removing, moving, or refactoring workflows, and updating workflow names. Developers frequently adjust their workflow structure to possibly better suit their project's needs and improve maintainability.

Trigger conditions, job configuration, environment setup, and scheduling play important roles in the evolution of GHA workflows, with developers adjusting triggers, job requirements, environment variables, and scheduling configurations to ensure proper execution and optimize resource usage.

Job configuration and environment setup involves adjusting job requirements and environment variables to ensure proper execution and optimize resource usage. The most common changes in job configuration include adding or removing jobs, matrix configuration, and updating job names. Environment setup primarily involves updating environment variables and runs-on configurations. These changes indicate that developers frequently adapt their workflows to different environments and job requirements.

RQ1 Summary

Our analysis reveals 64 common patterns of frequent changes in GHA workflow configurations, grouped into 8 categories based on their purposes. These common change patterns suggest that developers frequently adjust their GHA configurations to optimize execution, improve maintainability, update dependencies, and ensure proper workflow execution.

5.5 RQ2: What types of smells exist in GHA workflow configurations?

RQ2 methodology: We hypothesize that some of the frequent change patterns identified in RQ1 encompass *bad practices* that software engineers address in their GHA configurations. The updating of versions, refactoring of workflows, and optimization of run steps, for example, can be seen as efforts to improve the quality and maintainability of GHA configurations by eliminating potential smells. In our context, we consider a smell to be a suboptimal pattern or practice that can negatively affect the maintainability, performance, security, or reliability of the workflow. The frequent change patterns motivated us to delve deeper in changes to identify these potential smells.

We started this by critically examining each labeled change from RQ1 and performed the following steps: (1) We excluded changes that directly and immediately modify the workflow's behavior, such as adding or removing jobs, as these changes are not indicative of configuration smells. (2) We retained changes that do not immediately alter workflow behavior but enhance maintainability and reliability, such as using hashes instead of versions to ensure consistent action versions and mitigate the risk of unexpected behavior changes due to action updates. (3) We grouped related changes together based on their context and purpose, such as adding and updating permissions, which both pertain to access control management in GHA workflows. We further categorized changes based on their specific purposes, like adding conditional statements to prevent execution on forks or for other particular reasons.

For the labeled changes from steps 2 and 3, the second author critically grouped them into named smells based on their purpose and previous research on *GHA security*, *GHA performance/optimization*, and *Other CI/CD smells*. To ensure validity, the first and second authors discussed the grouping and naming of these potential smells, reaching agreement

on 35 candidate smells. We then removed smells with only one occurrence and those requiring project contextual knowledge beyond the configuration file (e.g., repository state or available actions). This resulted in 22 potential smells, defined in Table 5.2. The complete list of 35 candidate smells, including removed ones is available in the replication package [182].

RQ2 answer. Our analysis of smells in common patterns of changes in GHA workflows revealed 22 distinct smells, which we categorized into three main groups: security (3 smells), performance/optimization (10 smells), and other CI/CD smells (9 smells). Table 5.3 provides an overview of these smells, including the number of fixes for each smell among the 83 analyzed projects, the number of projects in which the smell was fixed by at least one commit, and the supporting evidence from previous research. To trace the fixes to these smells back to the specific changes in GHA configuration commits of these projects, see our study's replication package [182].

Security smells were identified based on changes addressing potential vulnerabilities. Table 5.3 lists 3 smells in this category, their description, and their frequency of occurrence in projects. These smells can lead to unauthorized access, code injection, and other security breaches. Fixing these smells can reduce the attack surface and ensures more secure workflows. The identified security smells are backed by evidence from previous studies [138, 169, 170].

Performance/optimization smells were identified based on changes addressing resource usage and limitations. In Table 5.3 we observe 10 smells with this type, including their descriptions. These smells can lead to inefficient resource usage, longer build times, and increased costs; addressing them may result in optimizations. A recent study on resource usage and limitations in GHA supports most of the identified performance/optimization smells [172]. While *Smell 10 and 11* were not mentioned in that study, GitHub documentation provides rationale for addressing them to optimize storage usage [183].

Other CI/CD smells were identified based on changes addressing bad practices, configuration smells, and the need for pipeline restructuring. Table 5.3 provides more details on the 9 instances of this type of smell. These smells can lead to maintainability issues, inconsistent build results, and difficulty in understanding and debugging workflows. Fixing these smells can improve readability, maintainability, and reliability, making it easier for team members to understand and modify workflows. The identified CI/CD smells are backed by research on bad practices in CI/CD [173, 174, 177, 178].

RQ2 Summary

Our analysis of common change patterns in GHA workflow configurations revealed 22 distinct smells, which we categorized into three main groups: security smells (3 instances), performance/optimization smells (10 instances), and other CI/CD smells (9 instances). These smells were identified based on changes addressing potential vulnerabilities, resource usage and limitations, and bad practices or configuration issues. The identified smells are backed by evidence from previous research on GHA security, performance, and CI/CD.

Table 5.3: Identified fixed smells in the history of GHA configuration files changes.

Cat	ID	Smell	# Proj.	# Tot.	Ref.
Security	1	**Define permissions for workflows with external actions** Problem: Overly permissive access increases security risks. Solution: Specify minimal permissions.	39	82	[170]
	2	**Use commit hash instead of tags for action versions** Problem: Tags can be modified. Solution: `uses: actions/checkout@<commit-sha>`	8	11	[138]
	3	**Set permissions for GitHub Token** Problem: Default token overly permissive. Solution: Set `permissions` key.	4	8	[170]
Performance/Opt.	4	**Prevent running issue/PR actions on forks** Problem: Actions fail due to lack of permissions. Solution: Check repository owner.	12	20	[172]
	5	**Avoid jobs without timeouts** Problem: Can run indefinitely. Solution: Set timeout for jobs.	11	14	[172]
	6	**Stop workflows when newer commit in PR** Problem: Inefficient resource usage. Solution: Use concurrency to cancel runs.	8	10	[172]
	7	**Stop workflows when newer commit in branch** Problem: Inefficient resource usage. Solution: Use concurrency to cancel runs.	8	10	[172]
	8	**Avoid CI when no source code changed** Problem: Unnecessary resource usage. Solution: Use paths or paths-ignore.	7	15	[172]
	9	**Avoid scheduled workflows on forks** Problem: Inefficient for inactive forks. Solution: Check repository owner.	6	10	[172]
	10	**Avoid uploading artifacts on forks** Problem: Inefficient resource usage. Solution: Check repository owner.	5	6	[183]
	11	**Use 'if' for upload-artifact action** Problem: Unnecessary uploads. Solution: Add condition.	4	5	[183]
	12	**Avoid deploying jobs on forks** Problem: Unnecessary deployments. Solution: Check repository owner.	2	2	[172]
	13	**Avoid new workflow while previous running** Problem: Inefficient resource usage. Solution: Use concurrency groups.	2	2	[172]
Other Smells	14	**Correct indentation** Problem: Reduces readability. Solution: Use YAML linter.	25	57	[177]
	15	**Use fixed version for runs-on** Problem: Environment changes cause issues. Solution: Specify exact version.	12	17	[174]
	16	**Name run steps** Problem: Reduces readability. Solution: Use descriptive names.	11	12	[177]
	17	**Use cache parameter vs. cache option** Problem: Increases complexity. Solution: Update to cache parameter.	9	11	[173]
	18	**Use single-command steps** Problem: Reduces clarity. Solution: Split into simpler steps.	5	5	[177]
	19	**Run tests on multiple OS's** Problem: May miss OS-specific issues. Solution: Use `matrix` strategy.	4	10	[173]
	20	**Specify package versions** Problem: Non-reproducible builds. Solution: Specify exact versions.	3	5	[174]
	21	**Add comments to workflows** Problem: Reduces maintainability. Solution: Add explanatory comments.	3	3	[177]
	22	**Run CI on multiple language versions** Problem: May miss version-specific issues. Solution: Use `matrix` strategy.	2	13	[173]

5.6 RQ3: Can we automatically detect GHA configuration smells?

RQ3 methodology: To facilitate the automated detection of the smells identified in RQ2, we developed a suite of Python scripts. These scripts can detect lines in GHA configuration files that contain smells. We provide our Python scripts in our replication package [182].

To select popular and active projects, we started with the 100 most popular projects for each of the five programming languages used in RQ1 (JavaScript, Java, C#, Python, and TypeScript). We then applied additional criteria: (1) having GHA workflows, and (2) not in our dataset of 83 projects for RQ1. We ordered them by the number of pull requests merged in the last 30 days and selected the top 40 projects, thus ensuring these projects are in active development. To evaluate the effectiveness of our automated smell detection scripts, we curated a new dataset comprising 119 GHA configuration files from these 40 projects.

We first established a ground truth by manually analyzing the configuration files and identifying smells. This step took 1 person 2 days. We then executed our smell detectors on the dataset and recorded false positives and false negatives. Recall, precision, and F1-scores were calculated for each of the 22 smell detectors using the following formulas : $Precision = \frac{TP}{(TP+FP)}$, $Recall = \frac{TP}{(TP+FN)}$, and $F1-score = 2 \times \frac{(Precision \times Recall)}{(Precision+Recall)}$, where TP, FP, and FN represent true positives, false positives, and false negatives.

RQ3 Answer. The results in Table 5.4 demonstrate the overall effectiveness of our automated smell detection scripts. Notably, 8 out of the 22 smell detectors (36.4%) achieved perfect precision, recall, and F1-scores of 100%. Furthermore, the median recall, precision, and F1-score across all smell detectors were 91.8%, 100%, and 0.93, respectively, signalling the overall good performance of our automated detection approach.

The detectors for smells 12, 19, 20 and 22 exhibited relatively low recall values of 19%, 40%, 42.9%, 52.6% respectively. These specific detectors struggle, because more contextual understanding is required to identify all instances of these specific smells. Detecting workflows that deploy the project (*Smell 12*) or locating where test suites are run (*Smell 19* and *22*) proved non-trivial. The detectors, focused on a single workflow, missed instances across multiple workflows. *Smell 20* also received a lower recall score because not all install commands support version specification, which the detector relied upon.

RQ3 Summary

Our evaluation demonstrates the feasibility and effectiveness of automatically detecting GHA configuration smells using custom-developed Python scripts. While the majority of our smell detectors exhibited high precision, recall, and F1-scores, for 4 out of 22 smells our detectors exhibit low recall.

Table 5.4: Evaluation of automated detection of GHA smells.

Smell #	Ground truth	FP	FN	Recall	Precision	F1-score
1	59	0	0	100%	100%	1
2	88	0	1	98.9%	100%	0.99
3	17	0	0	100%	100%	1
4	27	2	6	85.7%	92.6%	0.86
5	101	0	0	100%	100%	1
6	38	3	5	87.5%	92.1%	0.90
7	37	1	1	97.3%	97.3%	0.97
8	33	4	7	80.6	87.9%	0.84
9	16	0	3	84.2%	100%	0.91
10	18	9	2	81.8%	50%	0.62
11	13	0	2	86.7%	100%	0.93
12	20	1	81	19%	95.0%	0.32
13	38	4	1	97.1%	89.5%	0.93
14	54	0	35	60.7%	100%	0.76
15	99	1	0	100%	98.99%	0.99
16	61	0	0	100%	100%	1
17	7	0	0	100%	100%	1
18	59	0	0	100%	100%	1
19	20	0	30	40%	100%	0.57
20	12	2	9	52.6%	83.3%	0.65
21	25	0	1	96.2%	100%	0.98
22	7	1	8	42.9%	85.7%	0.57
Total	849	**Median**		91.8%	100%	0.93

5.7 RQ4: To what extent do developers find the proposed fixes for identified GHA configuration smells relevant?

RQ4 methodology: To externally validate and assess the practical relevance of the identified GHA workflow smells, we conducted an open-source contribution study [184]: we created pull requests (PRs) to popular and actively maintained GitHub repositories, addressing the smells detected in their GHA workflow configurations. By submitting these pull requests, we aimed to gauge the receptiveness of the open-source community to our proposed changes and gather insights into the smells that we observed in RQ2. We acquired approval from the Human Research Ethics Council of our university and followed their guidelines throughout the study.

For our contribution study, we selected 40 projects on GitHub using the same criteria detailed in RQ3. For each of these projects we ran our tool discussed in RQ3 on their GHA workflows to identify their smells. Subsequently, we strategically prepared PRs by addressing multiple smells per PR, to ensure that each smell received at least one reaction across all submissions. The PRs included explanations of the smells they aimed to fix (see Figure 5.1). To ensure transparency, we explicitly mentioned that the contribution was

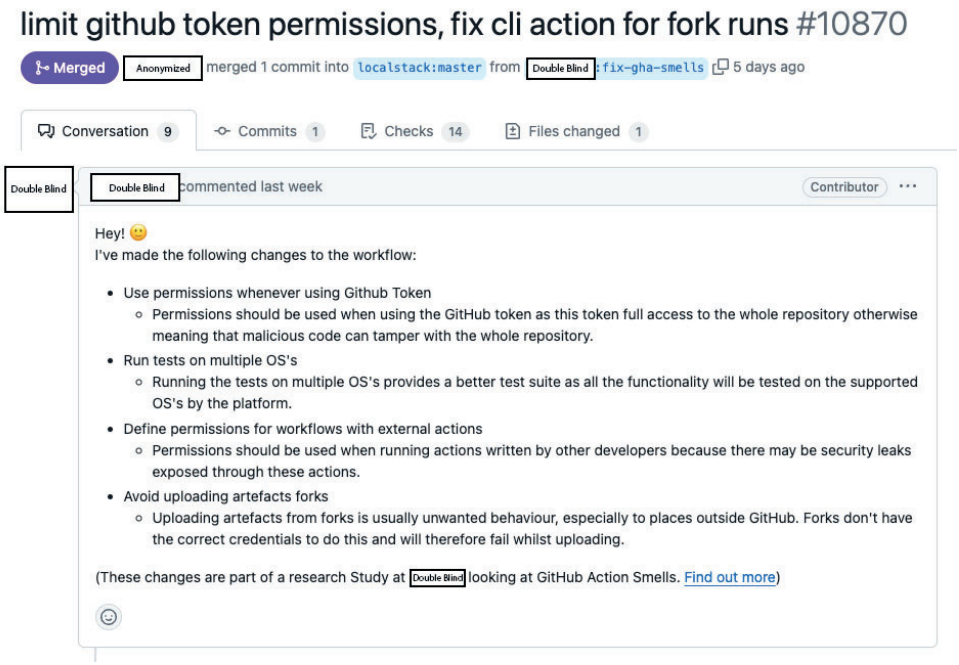

Figure 5.1: An example of the submitted pull request, #35.

part of a research study and provided a link to a separate web page containing further information about the study.

Table 5.6 and Table 5.7 displays the details of the submitted PRs, including their IDs, associated GitHub projects, status, and the fixed smells. The order of projects is based on the time of opening their associated PR. We submitted the PRs in April-May 2024, ensuring that maintainers of the projects had at least 3 weeks to provide feedback. In total, out of the 40 submitted PRs, 32 received at least one response at the time of writing this paper, either in the form of a comment, or a decision regarding merging or closing the PR. The responses to the proposed fixes for the smells are as follows: 46 accepted smells '✓', 34 rejected smells 'X', and 37 pending final decisions ' O '. At the time of writing this paper, 15 PRs were merged, 8 closed, and 17 pending. Among the pending PRs, 9 received comments and are included in Table 5.6 and Table 5.7.

The analysis focused on the 165 comments received on our pull requests (PRs). We used an open card sorting approach [180], which allows categories to emerge naturally from the data without imposing pre-defined groups [140]. Initially, the first and second authors independently coded 20% of the comments each. After agreeing on the codes, the second author proceeded to code the remaining comments. The codes were then sorted using sticky notes on a Miro board[2] for each smell, taking into account the state of the PR related to that smell (merged, pending, or closed). The first and second authors collaboratively performed the categorization by manually sorting the codes into groups during a joint

[2]https://miro.com/

Table 5.5: Type of comments received per each smell and the state of the associated PR(s).

PR State	Clarification Question	Suggest Edit	Feedback
Merged	2, 3, 4, 5, 20	4, 6, 7, 9, 10, 11, 12, 18	1, 3, 4, 6, 7, 9, 12
Pending	1, 2, 4, 6, 13	4, 5, 7, 10, 11, 12, 13	2, 13, 15
Closed	2, 4, 11	–	1, 2, 4, 5, 6, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22

session. To ensure reliability and validity, they conducted a discussion meeting to review the codes and categories until reaching a negotiated agreement [181].

RQ4 answer: The qualitative analysis of the comments received on the submitted pull requests adds to quantitative data on accepted and rejected pull requests; it enables us to better understand how maintainers perceive the identified GHA configuration smells, and the proposed fixes.

Our qualitative analysis revealed seven general types of comments: (1) **Clarification Question**: comments asking for further information or explanations regarding the proposed changes; (2) **Suggested Edit**: comments providing suggestions to improve the submitted PR; (3) **Feedback**: comments expressing opinions or concerns about the proposed fixes for the smells, or discussing the judgments of the PR; (4) **Automated messages**: Comments from software bots [185] that automate tasks or report to facilitate contribution; (5) **Decision-related comments**: comments regarding the decision to accept or reject the contribution; (6) **Appreciation**: comments expressing gratitude for the submitted contribution; and (7) **Unrelated comments**: comments not directly related to the proposed changes, e.g., mentioning other maintainers to review the PR.

We focus on the first three types of comments received on PRs related to each smell, as they provide relevant information about how maintainers perceived our identification and fixes of the GHA configuration smells. The details of this qualitative analysis are available in our replication package [182].

Clarification Questions: were related to the purpose and impact of the proposed changes to fix a smell (*Smell 1, 2, 3, 4, 6, 11, 12, 13, 15, 19,* and *20*). Maintainers sought to better understand how fixing the smells will impact their workflow. Figure 5.2 shows an example related to *Smell 1*, where the developer asks questions to better understand how permission configuration of the workflow works.

In 5 cases of a smell fix, our PRs were merged after addressing clarification questions, while in 7 cases, the proposed fixes were rejected (see Table 5.5). The reasons for rejection varied. For *PR 29273*, maintainers stated that they would consider an alternative solution for *Smell 2*, but since the workflow's task was not critical, they decided against making changes. In *PR 28909*, a maintainer expressed concerns about the trade-offs of fixing *Smells 11, 15, and 19*, as modifying the functioning workflow could lead to unintended consequences, potentially compromising its stability. Another maintainer in the same PR argued that addressing *Smells 4 and 12* would introduce a trade-off between maintainability

Table 5.6: List of repositories that received at least a response from the maintainers (Smells 1-11). The complete list of opened PRs is available in our replication package [182]. Accepted smells are shown with a ✓, rejected smells with a X, and smells pending a final decision with a ‘ O ’.

#	Project	# Stars	Status	PR ID	Fixed Smells										
					1	2	3	4	5	6	7	8	9	10	11
1	Jackett/Jackett	11.4k	Merged	15274	–	–	–	–	–	✓	✓	–	✓	–	–
2	jquery/jquery	58.9k	Closed	5480	–	–	–	–	–	–	–	–	–	–	–
3	oracle/graal	19.8k	Pending	8836	–	–	–	–	–	–	O	–	–	O	–
4	prisma/prisma	37.5k	Merged	23965	–	–	–	–	–	–	–	–	–	✓	✓
5	nuxt/nuxt	52.4k	Merged	26937	–	–	–	–	–	–	–	–	✓	✓	✓
6	cypress-io/cypress	46.2k	Merged	29416	–	–	✓	–	–	–	–	✓	✓	–	–
7	dotnet/AspNetCore.Docs	12.4k	Merged	32420	–	–	–	–	–	–	✓	–	✓	–	–
8	ray-project/ray	31.5k	Merged	44990	–	–	–	–	✓	–	–	–	–	–	–
9	parcel-bundler/parcel	43.2k	Pending	9672	–	–	–	–	–	O	–	O	–	O	O
10	halo-dev/halo	32k	Merged	5809	–	–	–	–	–	✓	–	–	–	–	✓
11	doocs/leetcode	29.2k	Merged	2677	–	–	–	–	–	✓	✓	✓	–	–	–
12	spacedriveapp/spacedrive	29.1k	Merged	2412	✓	✓	–	–	–	–	–	–	✓	–	–
13	unoplatform/uno	8.5k	Merged	16508	–	✓	–	–	–	–	✓	–	–	–	–
14	scikit-learn/scikit-learn	58.4k	Closed	28909	–	–	–	X	–	–	–	–	–	–	X
15	microsoft/semantic-kernel	18.6k	Pending	6041	–	O	–	O	O	–	–	–	–	–	–
16	keycloak/keycloak	20.2k	Closed	29164	–	–	–	–	–	–	–	X	–	X	X
17	getsentry/sentry	37.1k	Closed	69915	X	X	–	–	X	–	–	–	–	–	–
18	dbeaver/dbeaver	37.8k	Closed	29273	X	X	–	–	–	X	–	–	–	–	–
19	commaai/openpilot	48.2k	Closed	32326	–	–	–	–	–	–	–	–	–	–	–
20	abpframework/abp	12.3k	Pending	19665	–	O	–	–	O	–	–	–	–	–	–
21	App-vNext/Polly	1.2k	Merged	2097	–	–	–	✓	–	–	–	–	–	✓	✓
22	jenkinsci/jenkins	22.5k	Pending	9236	O	–	–	–	–	O	–	–	–	–	–
23	trpc/trpc	33k	Closed	5702	–	–	–	–	–	–	–	–	–	–	–
24	appwrite/appwrite	41.5k	Pending	8075	–	–	–	O	–	O	–	–	–	–	–
25	gui-cs/Terminal.Gui	9.2k	Pending	3449	–	–	–	–	O	–	–	O	–	–	–
26	gpt-engineer-org/gpt-engineer	50.8k	Merged	1156	✓	–	–	–	–	✓	–	–	–	–	–
27	cheeriojs/cheerio	27.9k	Merged	3826	–	–	–	–	–	–	–	✓	–	–	–
28	remix-run/remix	28.1k	Pending	9478	O	O	–	–	O	–	–	–	–	–	–
29	localstack/localstack	52.5k	Merged	10870	✓	–	✓	–	–	–	–	–	–	–	–
30	netty/netty	32.9k	Closed	14077	–	X	–	–	–	–	–	–	–	–	–
31	openzipkin/zipkin	16.8k	Merged	3770	–	–	–	–	✓	–	–	–	–	–	–
32	google/gson	23k	Pending	2684	–	–	–	–	O	–	–	–	–	–	–
Summary: 40 Opened PRs, 15 Merged, 8 Closed, 17 Pending															
			Total Accepted:		3	2	2	1	2	4	4	3	5	3	4
			Total Rejected:		2	3	0	1	1	1	0	1	0	1	2

and resource optimization. Fixing these smells would require adding more “if” statements to prevent running the workflow on forks, which would optimize resource usage. However, this optimization comes at the cost of reduced maintainability, as the additional conditional statements make the workflow harder to understand and modify. We see indications that the maintainers tend to prioritize maintainability over resource usage optimization of their project’s forks. Lastly, in *PR 23965*, the attempt to fix *Smell 19* by running tests on multiple operating systems using a matrix strategy led to a trade-off between test coverage and developer productivity. The increased number of jobs resulted in a higher occurrence of false positives due to flaky tests, requiring developers to manually verify more test cases.

Table 5.7: List of repositories that received at least a response from the maintainers (Smells 12-22). The complete list of opened PRs is available in our replication package [182]. Accepted smells are shown with a ✓, rejected smells with a X, and smells pending a final decision with a ‘ O ’.

#	Project	# Stars	Status	PR ID	Fixed Smells										
					12	13	14	15	16	17	18	19	20	21	22
1	Jackett/Jackett	11.4k	Merged	15274	–	–	–	–	–	–	–	–	–	–	–
2	jquery/jquery	58.9k	Closed	5480	–	–	–	–	–	X	X	X	–	–	X
3	oracle/graal	19.8k	Pending	8836	–	O	–	–	–	–	–	O	–	–	–
4	prisma/prisma	37.5k	Merged	23965	–	–	–	–	–	–	–	X	–	–	–
5	nuxt/nuxt	52.4k	Merged	26937	–	–	–	–	–	–	–	–	–	–	–
6	cypress-io/cypress	46.2k	Merged	29416	–	–	–	–	–	–	–	–	–	–	–
7	dotnet/AspNetCore.Docs	12.4k	Merged	32420	–	–	–	–	–	–	–	–	–	–	–
8	ray-project/ray	31.5k	Merged	44990	–	–	–	X	–	–	–	–	–	–	–
9	parcel-bundler/parcel	43.2k	Pending	9672	–	–	–	–	–	–	–	–	–	–	–
10	halo-dev/halo	32k	Merged	5809	✓	–	–	–	–	–	–	–	–	–	–
11	doocs/leetcode	29.2k	Merged	2677	✓	–	–	✓	–	–	–	–	–	–	–
12	spacedriveapp/spacedrive	29.1k	Merged	2412	–	–	–	–	✓	–	✓	–	–	–	–
13	unoplatform/uno	8.5k	Merged	16508	–	–	–	–	✓	–	X	–	–	–	–
14	scikit-learn/scikit-learn	58.4k	Closed	28909	X	–	–	X	–	–	–	–	–	X	–
15	microsoft/semantic-kernel	18.6k	Pending	6041	–	–	–	O	O	–	O	–	–	–	–
16	keycloak/keycloak	20.2k	Closed	29164	–	–	X	–	–	–	–	–	–	–	–
17	getsentry/sentry	37.1k	Closed	69915	–	–	–	X	X	–	X	–	–	–	–
18	dbeaver/dbeaver	37.8k	Closed	29273	–	–	–	X	–	–	–	–	–	–	–
19	commaai/openpilot	48.2k	Closed	32326	–	–	X	–	–	–	–	–	–	X	–
20	abpframework/abp	12.3k	Pending	19665	–	–	O	–	O	–	O	–	O	O	–
21	App-vNext/Polly	1.2k	Merged	2097	✓	–	–	–	–	–	–	–	–	–	–
22	jenkinsci/jenkins	22.5k	Pending	9236	O	–	–	–	–	O	–	–	–	–	–
23	trpc/trpc	33k	Closed	5702	X	–	–	–	X	–	–	X	–	–	–
24	appwrite/appwrite	41.5k	Pending	8075	O	–	–	–	–	–	–	–	–	–	–
25	gui-cs/Terminal.Gui	9.2k	Pending	3449	O	–	–	–	–	–	–	–	–	–	–
26	gpt-engineer-org/gpt-engineer	50.8k	Merged	1156	–	✓	–	–	–	–	–	–	✓	–	–
27	cheeriojs/cheerio	27.9k	Merged	3826	✓	–	–	–	–	–	✓	–	–	–	–
28	remix-run/remix	28.1k	Pending	9478	–	–	–	–	–	–	–	–	–	–	–
29	localstack/localstack	52.5k	Merged	10870	✓	–	–	–	–	–	–	X	–	–	–
30	netty/netty	32.9k	Closed	14077	–	–	–	–	–	X	–	–	–	–	–
31	openzipkin/zipkin	16.8k	Merged	3770	–	–	–	–	–	✓	–	–	–	–	–
32	google/gson	23k	Pending	2684	–	–	–	O	–	–	O	–	–	–	–
Summary: 40 Opened PRs, 15 Merged, 8 Closed, 17 Pending															
			Total Accepted:		5	1	0	1	2	1	2	0	1	0	0
			Total Rejected:		2	0	2	4	2	2	3	4	1	1	1

Consequently, the expanded testing effort reduced developer productivity, as they spent more time on testing-related tasks rather than other aspects of development.

Suggested Edits: maintainers suggested edits for 10 of the smells, falling into two categories: **1) alternative smell fixes** and **2) typographical corrections**. In terms of alternative smell fixes (*Smell 4, 9, 10*, and *12*), developers proposed different approaches to address these smells compared to our original fixes, aiming to enhance workflow reusability and maintainability (Figure 5.3). These alternative solutions can provide insights into resolving smells while aligning with project-specific needs. Secondly, maintainers identified and suggested typographical corrections for our submitted PRs, such as incorrect

.github/workflows/upload_release_asset.yml Outdated

```
@@ -13,6 +13,7 @@ jobs:
13  13          FOSSA_API_KEY: ${{secrets.FOSSAAPIKEY}}
14  14          repo-token: ${{ secrets.GITHUB_TOKEN }}
15  15          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    16 +    permissions: {}
```

Anonymized on Apr 26 Member

Double blind We need to specify the permissions we want here right? What happens if this object is empty? https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobsjob_idpermissions

Figure 5.2: Example of a comment asking a clarification question.

capitalization. The suggested edits confirm the maintainers' receptiveness to accepting smell fixes, as confirmed by Table 5.2, which shows that no PRs receiving edit suggestions were closed.

5

Anonymised on Apr 29 Member

I generally use the organization name only, so it's easier to reuse the code in multiple repos. (Yes, I know it's a copy / paste reuse, but still...)

Suggested change

```
-   if: ${{ github.repository == 'dotnet/AspNetCore.Docs' }}
+   if: ${{ github.repository_owner == 'dotnet' }}
```

Commit suggestion

Figure 5.3: Developer providing a different fix to Smell 4.

Feedback: maintainers provided three categories of feedback: **1) positive feedback on the impact of fixing the smell** (*Smell 3* and *7*): maintainers acknowledged that fixing *Smell 3* improves the developer experience, and for *Smell 7*, they confirmed that the workflow functions as expected after the changes. **2) skepticism about the value of the contribution** (*Smell 1, 2, 4, 6, 10, 11, 12, 14, 15, 17, 20, 21*): maintainers expressed doubts about the usefulness of the proposed fixes, finding the purpose of the PRs unclear or questioning the necessity of modifying functioning workflows (Figure 5.4). In *PR 32326*, *Smell 14 and 21* were rejected because the maintainers found the explanations regarding the smells unclear. Moreover, *Smell 4, 10 and 12* received conflicting feedback in two PRs, where maintainers were uncertain about GitHub's policy on running workflows on forks, leading them to question the relevance of the smells. Ultimately, *PR 2097* was merged, while *PR 28909* was closed, with the maintainers' doubts about GitHub's policy about running project's workflows on its forks being the primary reason for their skepticism. Additionally, *PR 28909*, which aimed to fix *Smell 11, 12, 15, and 20*, was closed due to the developers' concerns about potential side effects. However, since at least one other PR addressing these smells was merged, we believe that a more thorough explanation of the smells could have helped alleviate the maintainers' skepticism. Moreover, reporting a

calculation of the impact of smell fixes along with the PR, similar to what Durieux has done in a contribution study of Docker smells [186], can address this skepticism about fixing GHA smells in future studies. Two PRs addressing *Smell 1, 2, 15, 16, and 18* also received feedback indicating that the maintainers preferred discussing changes to workflows before submitting PRs. In these cases, the maintainers' reluctance to accept external contributions that change infrastructure-related aspects of the project, such as GHA workflows, was evident. Overall, the skepticism regarding the value of the proposed smell fixes appears to stem from two main factors: first, maintainers' uncertainty about the relevance and necessity of the fixes, particularly for functioning workflows, highlighting the importance of clear and comprehensive explanations when proposing changes. Second, the culture of open-source projects, which tends to be cautious about accepting external contributions that modify infrastructure-related aspects, such as GHA workflows, without prior discussion [*"appreciate the contribution but please in the future discuss changes to infrastructure..."*, *PR 69915*]. **3) explanations for why the smell is not applicable** (*Smell 2, 4, 9, 10, 12, 15, 17, 19, 22*): maintainers indicated that the workflows in question do not perform critical tasks (*Smell 2, 15, 17, 19, 22*), or expressed deliberately keeping upload artifacts actions on forks because they need it (*Smell 10 and 11*), [*"we want to upload artifacts because we need them later, also forks will or might need them, I see no reason to add a condition for this."*, *PR 29164*].

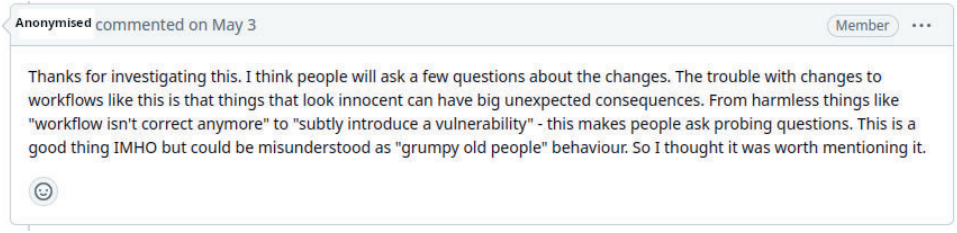

Figure 5.4: An example of maintainers skeptical about changing the workflow.

To answer RQ4, we analyzed the reactions received for each smell (X and ✓ in Table 5.6 and Table 5.7) and identified three groups: 1) *mostly accepted smells*, which received at least 2 acceptances and at most 1 rejection, indicating maintainers' receptiveness to fixing these smells in their workflows; 2) *mostly rejected smells*, which had more rejections than acceptances, with at least two non-pending PRs, suggesting that developers were generally not open to accepting the proposed fixes; and 3) *smells with mixed opinions or insufficient responses*, which received both positive and negative reactions or had only one non-pending PR, indicating no clear consensus regarding the importance or relevance of these smells.

Mostly accepted: 7/22 smells are considered mostly accepted (*Smell 3, 5, 6, 7, 8, 9*, and *10*) and were merged in 14 projects (Table 5.6 and Table 5.7). Out of these smells, 3 received no rejections (*Smell 3, 7*, and *9*), while 4 received 1 rejection each (*Smell 5, 6, 8*, and *10*). *Smell 5* was rejected in *PR 69915* because of project's culture of not accepting external contribution to infrastructure-related aspects of the project. Similarly, *Smell 6* was rejected in *PR 29273* through feedback indicating that the maintainer did not consider the contribution valuable in case of their simple workflow. *Smell 8* and *10* were rejected in *PR 29164* due to the existing workflow logic already handling the skipping of runs based on modified code

areas, and the maintainer mentioned that artifacts are needed by other workflows being run on forks, aligning with the category of "explanations for why the smell is not applicable." While a few rejections occurred, e.g., due to project-specific factors, the overall positive receptiveness of developers regarding these smells highlights their recognition of the value in fixing them.

Mostly rejected: 6/22 smells were mostly rejected: *Smell 2, 14, 15, 17, 18*, and *19*. *Smell 14*, and *19* were rejected in all PRs, with no acceptance. *Smell 15* and *19* had the most rejections (4 each), suggesting developers were not receptive to the proposed changes. For *Smell 14* and *15*, maintainers showed "skepticism about the value of the contribution". *Smell 19* was rejected because maintainers "did not find the smell applicable to their project". These findings highlight that certain smells were rejected due to maintainers' skepticism about the value of the contribution or its relevance to their projects, or project-specific reasons making the smells inapplicable.

Mixed opinions/inconclusive: 9/22 smells received mixed opinions or insufficient responses: *Smell 1, 4, 11, 12, 13, 16, 20, 21*, and *22*. *Smell 4, 16*, and *20* have an equal number of rejections and acceptances, preventing strong conclusions about their perception by developers. *Smell 1, 11*, and *12* have 3, 4, and 5 accepted PRs, respectively, but were also rejected twice due to developers showing "skepticism about the value of the contribution." Despite the higher acceptance rate, the presence of multiple rejections prevents us from confidently stating that developers were receptive about these smells. *Smell 13, 21*, and *22* received only one response each, with the remaining PRs pending, providing insufficient data to draw conclusions about their perception by developers. In summary, the mixed opinions and limited responses for these smells necessitate further investigation.

RQ4 Summary

While there was a consensus among maintainers regarding the relevance and value of fixing 7/22 smells (*Smell 3, 5, 6, 7, 8, 9*, and *10*), as indicated by their acceptance, 6/22 smells (*Smell 2, 14, 15, 17, 18*, and *19*), were mostly rejected, suggesting a lack of consensus on the applicability or importance of the proposed fixes for these smells in their projects. Additionally, there were 9/22 smells (*Smell 1, 4, 11, 12, 13, 16, 20, 21*, and *22*) that received mixed opinions or insufficient responses, preventing a clear consensus about their perception by developers.

5.8 Discussion

This study provides insight into the existence and relevance of **seven GHA workflow smells**, which have been validated by open-source developers through our contribution study. The novelty of this work lies in the bottom-up approach that we have taken to identify these seven workflow smells: from identifying frequent change patterns in GitHub Actions workflows, to analyzing the negative side effects that are alleviated through the change patterns and thus establishing a set of candidate workflow smells, to validating the candidate smells through a contribution study. Through this process, we were also able to identify a new smell, namely: *Smell 10: "Avoid uploading artifacts on forks"*.

The identification of *Smell 10* is a key contribution of this study, as it highlights the

resource usage optimization with avoiding uploading artifacts on forked repositories. The validation of this smell through open-source contributions, demonstrates its relevance and importance in the context of GHA workflow optimization.

The other six smells identified in this study were previously discussed in the literature, with five of them (*Smell 5, 6, 7, 8, and 9*) presented as optimization techniques, and one (*Smell 3*) flagged as a security concern (Table 5.4). However, prior to our study, there was a lack of evidence regarding developers' perceptions and willingness to accept fixes for these smells. Our contribution study fills this gap by providing valuable insights into how developers view these smells and their readiness to accept the PRs fixing them. The acceptance of fixes by developers for all seven smells, including the newly introduced *Smell 10*, suggests that addressing these issues is considered important and beneficial by the maintainers of the projects that we worked with.

We developed automated smell detectors for the GHA configuration smells. The evaluation of these detectors showed promising results, with the majority exhibiting high precision, recall, and F1-scores (0.84 - 1). However, *Smell 10* had a lower F1-score (0.62), as our detector only considered two common artifact upload actions on GHA. Future work should analyze more upload actions to improve the detector's F1-score.

The rejection and mixed opinion of the maintainers of the projects in our contribution study other 15 candidate workflow smells, makes us less confident about their relevance. In future research, we intend to expand the contribution study and set up interviews with maintainers to better understand the relevance of these smells in different project contexts.

5.8.1 Implications & Future Work

The findings of this study have several implications for researchers, tool developers, and practitioners working with GitHub Actions:

Awareness of GHA workflow smells: the identified smells provide a foundation for understanding common issues in GHA configurations. Researchers and practitioners can leverage this knowledge to develop best practices and create educational materials that raise awareness [52, 112] about these smells, ultimately helping developers prevent, identify, and address them in their GHA workflows.

Tool support for smell detection and refactoring: the automated smell detectors developed in this study can be integrated into existing GHA development tools or integrated development environments (IDEs) to provide real-time feedback and suggestions for improving workflow configurations [18, 19, 103].

Empirical studies on workflow smells: for some candidate workflow smells, we received mixed opinions on their relevance, which could be explained by project-specific circumstances. Further study is needed to better understand how context influences the perception and importance of smells.

5.8.2 Threats to Validity

This section discusses potential threats to the validity of our study and the steps taken to mitigate them [187]:

External validity: the identified smells and their validation are based on a sample of 83 and 40 GitHub projects, respectively, which may not be representative of all GHA workflows.

To mitigate this threat, we selected popular, active projects with diverse programming languages, sizes, and domains.

Internal validity: the identification of workflow smells and the development of automated detectors relied on the authors' expertise and interpretation of the frequent change patterns which formed the internal validation of the potential smells. To mitigate potential biases, two authors independently reviewed the change patterns and discussed any disagreements until a consensus was reached. Additionally, we associated the identified potential smells with other studies on GHA and CI/CD to mitigate our own bias.

Conclusion validity: the conclusions drawn from the contribution study (RQ4) are based on a limited number of PRs and subjective opinions, potentially affecting generalizability. Many candidate smells received mixed or insufficient feedback due to the limited number of contributions. Future larger-scale studies, or survey/interview studies with more developers could strengthen the findings.

We followed a rigorous methodology and thoroughly documented the source code, scripts, and procedures to ensure replicability, enhancing transparency and reliability. We provide an replication package with all our materials [182].

5.9 Conclusion

This study provides insights into the existence and relevance of seven GitHub Actions (GHA) workflow smells. Employing a bottom-up approach, by analyzing frequent change patterns in 10,012 commits from 83 projects, we identified 64 change patterns grouped into 8 categories (RQ1).

From these frequent change patterns, we identified and defined a candidate list of of 22 potential GHA workflow smells (RQ2). The external validation confirmed the relevance of six previously discussed smells and identified a new smell, *Smell 10: "Avoid uploading artifacts on forks"*, highlighting optimization of resource usage. Custom smell detectors (RQ3) showed promising results, with F1-scores ranging from 0.84 to 1 for most smells, except for *Smell 10* with a lower recall due to considering limited actions for uploading artifacts in our detector. The contribution study (RQ4), involving 32 pull requests, provided insights into developers' perceptions and willingness to accept fixes, which enabled us to validate 7 out of the 22 candidate GHA workflow smells as actual smells.

The insights of this study can inform future research and tool development efforts in GHA workflow optimization and maintenance, supporting the growing adoption of GHA in the GitHub software development ecosystem.

6

Beyond the YAML File: Understanding Real-World GitHub Actions Workflow Adoption

Continuous Integration and Continuous Deployment (CI/CD) have become fundamental to modern software development, with GitHub Actions (GHA) emerging as a dominant automation platform. In this study, we analyze real-world execution records of GHA, examining how developers react to workflow failures, how these workflows are utilized by projects, and how these aspects relate to project characteristics. We quantitatively analyze 258,300 workflow run records from 952 repositories and perform an in-depth qualitative analysis of 21 selected, diverse GitHub repositories to understand how maintainers and contributors interact with workflow results. We identify three distinct failure response patterns, observe that higher usage intensity of GHA workflows correlates with lower failure rates, and uncover a configuration-usage gap where the presence of configuration files masks disabled or unused workflows. Moreover, our qualitative analysis of relationships between project characteristics and utilization patterns yields five hypotheses for future validation.

6.1 Introduction

Continuous Integration and Continuous Deployment (CI/CD) have become pillars of modern software development, enabling teams to automate testing, building, and deployment processes [24, 26, 66, 119, 132, 188–190]. GitHub Actions (GHA), introduced in 2019, has rapidly emerged as one of the most widely adopted CI/CD platforms, offering developers a powerful yet accessible framework for workflow automation directly integrated with their

This chapter is based on *A. Khatami, C. Brandt, and A. Zaidman. "Beyond the YAML File: Understanding Real-World GitHub Actions Workflow Adoption." In Proceedings of the 30th International Conference on Evaluation and Assessment in Software Engineering (EASE), 2026.*

source code repositories [138, 191]. While the potential benefits of workflow automation are well-documented [20, 24], realization of these benefits depends on how development teams utilize tools in practice [58].

Existing studies [138, 166, 172] analyze static workflow configuration (YAML) files defining automation steps, taking the presence of configuration files as a proxy for usage. However, studies on static analysis tools revealed a significant gap between configured tools and their actual usage [18, 192]. Many projects maintain configuration files for tools that are not or rarely used, or whose results are consistently ignored by developers [193]. This disconnect suggests that configuration files alone provide an incomplete view of how automation technologies impact software development processes.

In this study, we instead examine *workflow run data*: the execution records of GHA workflows. This includes the events that trigger runs and run outcomes. We qualitatively analyze the responses of developers to failures through subsequent commits and pull request (PR) discussions to develop a richer understanding of GHA's role in modern software development processes. We address the following research questions:

RQ1 What are the observed GitHub Actions utilization patterns?

Following a quantitative approach for RQ1, we examine utilization intensity (number of runs and failure rates) and trigger patterns (events causing workflow executions). This reveals how frequently failures occur and whether intensive workflow adoption correlates with failure rates.

RQ2 How do developers react to GitHub Actions workflow failures?

To understand the effect of GHA adoption on development processes, we first focus on failure points in pull requests and main branches, as they enable us to analyze subsequent developer actions that address these failures.

RQ3 What relationships exist between project characteristics and GitHub Actions utilization patterns?

For RQ3 we examine relationships between project characteristics and the patterns identified in RQ2 and RQ1 through a qualitative analysis of 21 repositories. Given the complexity beyond simple categorical mappings, we propose hypotheses for future validation rather than definitive conclusions.

To answer these questions, we quantitatively analyze run records from 765 repositories based on a dataset of 952 GitHub projects [172]. Then we systematically select 21 repositories for a qualitative analysis based on three key dimensions: popularity, workflow run frequency, and failure rates. We conduct in-depth analysis of each repository, examining workflow run records, developer interactions in PRs, associations between workflow runs and commits (both in PRs and the main branch), and the contextual characteristics of projects. Our previous study[1] established that quality assurance (QA) practices vary considerably across different development contexts [52]. The adoption and effectiveness of these practices are influenced by numerous factors including team size, technology

[1] Chapter 4 of this dissertation

stack, and development methodology. GHA workflows, as a mechanism for automating QA activities, likely exhibit similar contextual dependencies.

By examining how development teams interact with GitHub Actions in practice, our research **contributes a more nuanced understanding of GitHub Actions workflow adoption** than configuration analysis alone could provide. We identify patterns in how developers respond to workflow failures, recognizing common workflow utilization patterns, and examine relationships between these patterns and project characteristics. These findings can be the basis for future studies of GHA practices, grounding them in rich observational data.

Our paper is structured as follows: in Section 6.2 we review GitHub Actions studies. We then describe our methodology for our mixed-methods analysis in Section 6.3. In Sections 6.4 through 6.6 we present our findings regarding workflow utilization patterns across different project contexts (RQ1), developer responses to workflow failures (RQ2), and we explore relationships between these patterns and project characteristics (RQ3). Section 6.7 presents a discussion of the implications of our study. We conclude our paper with Section 6.8.

6.2 Background

GitHub Actions (GHA), introduced in 2019, has rapidly become the dominant Continuous Integration and Continuous Delivery (CI/CD) automation tool within the GitHub ecosystem, largely due to its deep integration and the provision of free resources to open source projects [106, 194, 195]. Quantitative studies of GHA configuration files demonstrated widespread and growing adoption: 22% of popular projects adopted GHA according to early studies [196], increasing to 37% for popular GitHub projects and up to 57% for repositories utilizing the most popular programming languages [165]. Previous research explored various facets of GHA adoption.

6.2.1 Impact on Development Practices

Research on the impact of GHA shows that its adoption generally correlates with faster resolution latency for pull requests (PRs) and issues, and increased commit frequency [196]. However, other studies report a more nuanced picture, noting an increase in rejected PRs and fewer commits in merged PRs [87, 166], with accepted PRs receiving more discussion comments and taking longer to merge after adoption [166].

6.2.2 Ecosystem Structure and Evolution

Studies describe GHA workflows and reusable components (Actions) as a complex, rapidly evolving ecosystem [138, 176]. Workflows, defined by jobs and steps, undergo continuous modification [138] including fixing YAML syntax errors, debugging, and updating instructions, leading researchers develop specialized tools for tracking commit changes of workflow configurations [197, 198]. Although Action reuse is widespread [138, 176], most workflows reference outdated releases, typically lagging at least seven months behind the latest version [176]. Moreover, analyses estimate an average annual cost of $504 for GHA adoption of paid-tier repositories, while optimizations in resource consumption are underutilized [172].

6.2.3 Security and Vulnerability Management

A major focus of studies has been the security risks in the GHA workflow configurations [165, 168, 170, 171, 194]. 99.8% of workflows are over-privileged with default read-write access, and 23.7% are exploitable for arbitrary code execution via PRs [170]. Vulnerable third-party actions are common [170], and security recommendations such as using commit hashes are followed by fewer than 3% of repositories [176]. Security tools like CodeQL remain underused; only 13.5% of top repositories enabling CodeQL actually use it [165]. Developers also cite security and complexity concerns when selecting Actions [199].

Most empirical research statically analyzes workflow configuration (YAML) files found in the *.github/workflows* directory, using their presence as a proxy for adoption and usage [138, 166, 172]. Only Bouzenia and Pradel [172] focus on resource optimization of GHA workflows. The novelty of this paper is to analyze workflow run data, i.e., the execution records of GHA workflow runs including their trigger events and outcomes with qualitative analysis of developers' reactions to them. This approach is necessary to gain a more nuanced understanding of GHA workflow adoption.

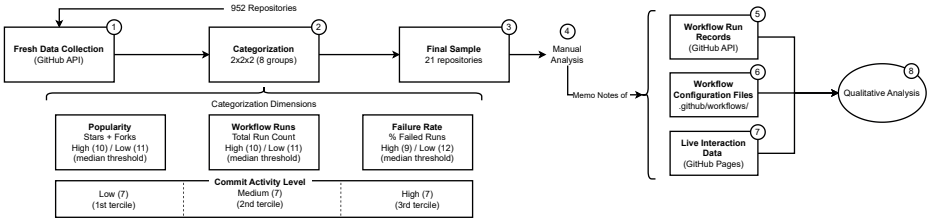

Figure 6.1: Methodology Overview

6.3 Methodology

We conducted an exploratory analysis of GitHub Actions (GHA) workflow adoption through a mixed methods analysis of workflow runs. This section explains our data collection process, repository sampling strategy, and the manual analysis approach used to answer our research questions. An overview is depicted in Figure 6.1.

Our study focuses on analyzing execution data from GHA workflows and related contextual information of repositories. We examine the recorded workflow runs[2] from the GitHub API, whether these runs pass or fail, how they were triggered, their associated pull requests (PRs) or commits, and how developers react to these outcomes. We define *workflow run data* as the execution history of GHA workflows, including timestamps, trigger events[3], GitHub page URLs, status and conclusion, associated commits, PRs, users, and branches. We examine these execution records in combination with observable developer reactions on PR and branch web pages.

[2]"Workflow Run" is the exact terminology used by GitHub, more details of these records are available at `https://docs.github.com/en/rest/actions/workflow-runs?apiVersion=2022-11-28#get-a-workflow-run`

[3]Events that trigger workflows: `https://docs.github.com/en/actions/reference/workflows-and-actions/events-that-trigger-workflows#about-events-that-trigger-workflows`

6.3.1 Data Collection

Our goal is to identify GitHub repositories that actively use GHA and, thus, have workflow run data available for analysis. This presents two challenges: (1) finding a representative sample of repositories using GHA, and (2) ensuring these repositories have accessible execution history. To address these challenges, we begin with an existing dataset while recognizing that GitHub's workflow run retention policy[4] would require us to collect fresh data.

We use the dataset by Bouzenia and Pradel [172], which contains 952 GitHub repositories with workflow run histories. This dataset has a balanced representation of popular repositories (600 with 100+ stars and commits) and less popular ones (352 with fewer than 100 stars). Of the initial 952 repositories, 946 were still publicly available at the time of data collection. We used the GitHub API to collect fresh workflow run data from these 946 repositories, of which 765 contained workflow runs. To collect sufficient data while staying within reasonable API usage limits, we collected up to 1000 run records per repository[5], yielding 258,300 run records collected at the end of 2024 (Figure 6.1 – ①). We filtered 213,640 non-bot-triggered runs across 742 repositories to analyze for RQ1 (Section 6.4).

Our data collection pipelines, available in our replication package [200], queried the GitHub API for recent workflow runs, commit histories, and other repository metadata, saving them into a MongoDB database [200] for subsequent analysis.

6.3.2 Categorization and Sampling Strategy

To capture GHA adoption patterns, we classified the 765 repositories with workflow runs using three attributes: popularity (stars + forks), workflow activity (total run count), and workflow failure rate (percentage of failed runs). We chose these dimensions to represent project scales, CI/CD adoption levels, and failure response contexts essential for our research questions (Figure 6.1 – ②).

Each attribute was split into high/low categories using median thresholds, creating 8 combinations (2×2×2) as shown in Table 6.2. Analysis of the population showed relatively balanced distribution across these groups, ranging from 9.4% to 15.7% of repositories per group, outlining that no single usage pattern dominates. While one repository per group could provide insight into each GHA adoption pattern, we selected multiple repositories from most groups based on observed diversity within groups and the need to capture varied adoption approaches within each categorization groups.

Sampling Approach: we selected 21 repositories using purposive sampling to ensure representation across all 8 classification groups (Table 6.2 and Figure 6.1 – ③). Rather than proportional sampling, we varied the number of repositories per group (1–5 repositories) based on research value, where some patterns (e.g., low-activity projects with low number of runs and high-activity with high number of runs) needed more examples to understand diverse utilization approaches, and practical coverage to ensure every combination was represented without missing important behavioral patterns.

[4] `https://docs.github.com/en/actions/administering-github-actions/usage-limits-billing-and-administration#workflow-run-history-retention-policy`

[5] This threshold reflects both API constraints and our pattern-identification goal. Collecting beyond 1000 runs per repository would require more resource and time while providing limited additional insight, as high-volume repositories represent a small minority with predominantly bot-triggered workflows.

Table 6.1: Sampled Repositories Categorization Attributes

ID	Repository	Pop.	Pop.G	Comm.	Comm.G	Runs	Run.G	Run.Fail%	Run.Fail.G
R1	trusttoken/ contracts-pre22	450	low	18	low	46	low	0.0	low
R2	nextcloud/ocsms	240	low	32	medium	7	low	0.0	low
R3	ngageoint/ geopackage-js	393	low	37	medium	21	low	14.3	low
R4	yandex-cloud/ serverless-plugin	72	low	91	medium	37	low	8.1	low
R5	iouAkira/ someDockerfile	181	low	15	low	15	low	86.7	high
R6	dotMorten/ NmeaParser	351	low	16	low	48	low	47.9	high
R7	nemuTUI/nemu	370	low	20	low	126	low	6.3	low
R8	data-driven-forms/ react-forms	399	low	191	high	64	low	4.7	low
R9	komamitsu/fluency	190	low	75	medium	234	high	16.2	high
R10	aws-cloudformation/ cfn-lint-vscode	416	low	103	medium	1000	high	2.7	low
R11	boutproject/ BOUT-dev	282	low	644	high	1000	high	20.5	high
R12	felix-fly/ v2ray-openwrt	823	high	7	low	2	low	0.0	low
R13	Thinkmill/manypkg	974	high	47	medium	142	high	12.7	low
R14	webextension-toolbox/ webext-toolbox	800	high	27	low	735	high	21.2	high
R15	prontolabs/pronto	2874	high	14	low	29	low	0.0	low
R16	ai/size-limit	8403	high	160	high	147	high	41.5	high
R17	lc-soft/LCUI	4525	high	209	high	135	low	74.8	high
R18	tsl0922/ttyd	9214	high	73	medium	365	high	0.5	low
R19	RailsEventStore/ rails_event_store	1546	high	1151	high	1000	high	0.2	low
R20	BeyondDimension/ SteamTools	22097	high	1466	high	1000	high	25.6	high
R21	libcpr/cpr	7599	high	294	high	1000	high	30.4	high

Pop: Popularity (stars + forks), Pop.G: Popularity group, Comm: Total commits since 2023, Comm.G: Total commits group
Run.G: Run group, Run.Fail%: Failure rate percentage, Run.Fail.G: Failure group

Figure 6.2 demonstrates that our sampled repositories span the distribution across all three classification dimensions (popularity, run counts, and run failure rates), ensuring coverage of diverse values. While commit activity was not part of our classification, our sample spans development activity levels capturing projects with varying development intensities.

Sample Characteristics: our final sample ranges from less popular/active projects (72 stars+forks, 7 commits) to more popular/active ones (22,097 stars+forks, 1,466 commits), with 2-1,000 workflow runs and failure rates from 0% to 86.7% (Table 6.1). This diversity lets us analyze how GHA adoption and developer behaviors vary across different project contexts. We intentionally keep repositories with very few runs, as we are also interested to study project that have workflows configured but rarely use them.

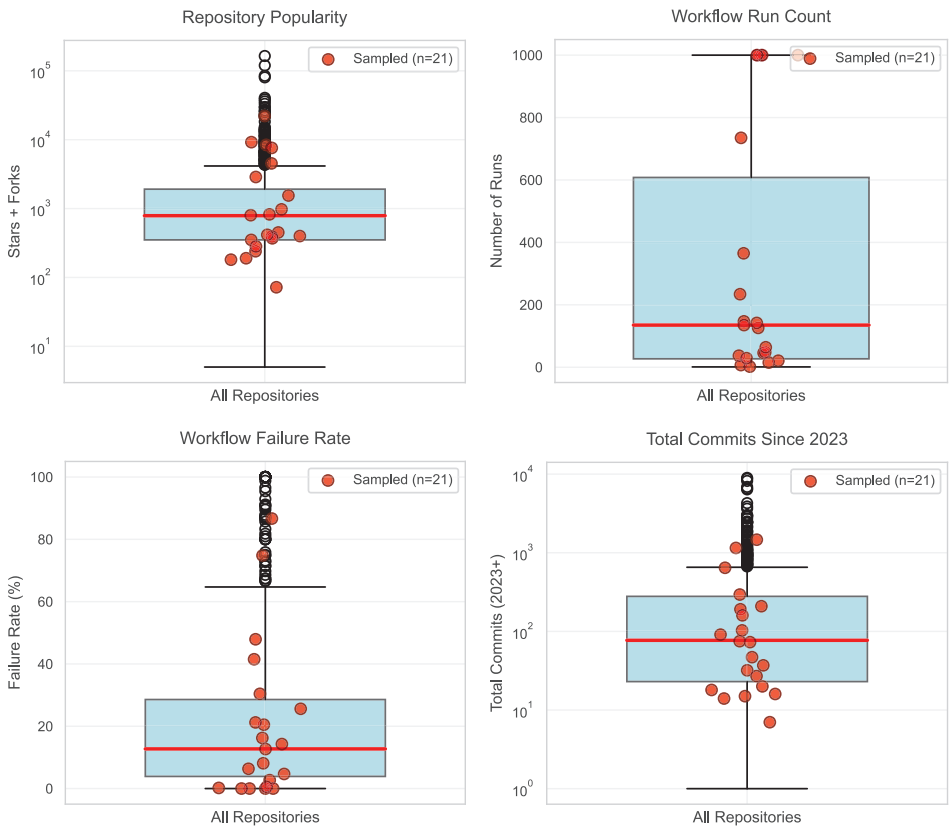

Figure 6.2: Distribution of sampled repositories (red dots, n=21) across the population. Our sample spans the complete range across (a) repository popularity, (b) workflow run count, (c) failure rates, and (d) commit activity since 2023.

We enriched this data with contextual attributes, including projects' programming languages, commit count, workflow configuration commit count, team and project size indicators to facilitate deeper understanding of project contexts during our sampling and manual analysis phases (Figure 6.1 – ③). These additional attributes are detailed in Table 6.3. The complete dataset is available in [200].

6.3.3 Manual Analysis through Qualitative Coding

To answer RQ2 and RQ3, we conduct a manual analysis. For each of the 21 selected repositories, we analyzed workflow run records (Figure 6.1 – ⑤), workflow configuration files (Figure 6.1 – ⑥), and live interaction data[6] (Figure 6.1 – ⑦), both from our API-collected data (available in our dataset [200]) and live data on the repositories' GitHub page.

Our manual analysis of each repository includes:

[6]Any form of interactions observable from the GitHub website UI, e.g., comments in pull requests, commit messages, active/deactivated workflows.

1. **Workflow file configuration:** analyzing the purpose of configuration files (*.github/workflows/*.yml*) and their commit history if and why they were updated.
2. **Workflow run records:** examining workflow run outcomes on main branches and in pull requests, with particular attention to developers' reactions to failures.
3. **Developer interaction:** analyzing commit messages after failures and discussions in PRs to understand if and how workflow failures triggered actions.
4. **Repository contextual information:** analyzing characteristics like programming language, team size, maintainer count, development style [85].

Our qualitative analysis (Figure 6.1 – ⑧) started with open coding and memo-writing by the first two authors for each repository from the mentioned elements above. Both authors analyzed repositories separately before merging the codes and grouping them into higher-level codes. These were then categorized into broader themes through constant comparison [154] with the original memos. The authors discussed the emerging groups until reaching a negotiated agreement [201]. The analysis required substantial manual effort to scavenge data from multiple sources and interpret the collected information through systematic coding.

6.3.4 Answering the research questions

Our systematic qualitative coding process of 21 repositories, together with the quantitative analysis of 765 repositories with run records, enabled us to understand workflow adoption beyond their configuration files by examining the rich contextual data surrounding workflow adoption and developer interactions. We use this mixed-methods analysis to answer our three research questions in Sections 6.4 to 6.6.

Table 6.2: Repository Sampling: 2×2×2 Classification

Group	Pop	%	Sampled Repositories	n
L-L-L	105	13.7	R1, R2, R3, R4, R7	5
L-L-H	112	14.6	R5, R6	2
L-H-L	86	11.2	R8, R10	2
L-H-H	80	10.5	R9, R11	2
H-L-L	72	9.4	R12, R15	2
H-L-H	94	12.3	R17	1
H-H-L	120	15.7	R13, R18, R19	3
H-H-H	96	12.6	R14, R16, R20, R21	4
Total	**765**	**100**		**21**

Pop=Population count; %=Population percentage
Groups: Popularity-Runs-FailureRate (L=Low, H=High)

Table 6.3: Repository Additional Information

ID	Repository	WF.C	Lang	Trigger Event Distribution	Mo.Gap	U.Runs	B.Runs
R1	trusttoken/ contracts-pre22	0	Type-Script	sch: 38, pr: 8	8.3	46	0
R2	nextcloud/ ocsms	0	Java-Script	sch: 7	1.4	7	0
R3	ngageoint/ geopackage-js	4	Type-Script	push: 19, dyn: 2	2.6	19	2
R4	yandex-cloud/ serverless-plugin	0	Type-Script	pr: 18, push: 15, dyn: 4	14.2	20	17
R5	iouAkira/ someDocker-file	6	Python	push: 12, wd: 3	9.3	15	0
R6	dotMorten/ NmeaParser	10	C#	push: 24, pr: 16, wd: 5, dyn: 3	13.0	48	0
R7	nemuTUI/ nemu	2	C	push: 72, pr: 54	13.1	126	0
R8	data-driven-forms/react-forms	0	Java-Script	pr: 62, dyn: 2	15.1	51	13
R9	komamitsu/ fluency	4	Java	pr: 196, push: 32, dyn: 6	13.0	50	184
R10	aws-cloud-formation/ cfn-lint-vscode	17	Java-Script	sch: 878, pr: 78, push: 40, rel: 4	9.6	945	55
R11	boutproject/ BOUT-dev	72	C++	push: 489, pr: 477, dyn: 20, sch: 14	3.3	880	120
R12	felix-fly/ v2ray-openwrt	4	Shell	push: 2	6.9	2	0
R13	Thinkmill/ manypkg	4	Type-Script	pr: 82, push: 60	11.7	136	6
R14	webextension-toolbox/ webext-toolbox	6	Type-Script	pr: 572, sch: 67, push: 49, dyn: 46, wd: 1	15.3	159	576
R15	prontolabs/ pronto	4	Ruby	pr: 22, push: 7	14.8	29	0
R16	ai/size-limit	15	Java-Script	push: 90, pr: 57	12.8	144	3
R17	lc-soft/LCUI	27	C	push: 134, pr: 1	13.9	133	2
R18	tsl0922/ttyd	8	C	push: 220, pr: 145	13.8	135	230
R19	RailsEvent-Store/rails_ event_store	108	Ruby	sch: 960, push: 28, pr: 12	1.9	998	2
R20	BeyondDi-mension/ SteamTools	25	C#	push: 791, pr: 98, create: 69, delete: 42	13.1	901	99
R21	libcpr/cpr	40	C++	pr: 454, push: 426, dyn: 117, rel: 2, wd: 1	7.1	883	117

WF.C: Workflow commits since 2023, Mo.Gap: Months between first and last run record
U/B.Runs: User/Bot runs. Event types: pr: pull_request, sch: schedule, dyn: dynamic, wd: workflow_dispatch, rel: release

6.4 RQ1: What are the observed GitHub Actions utilization patterns?

Beginning our analysis, we investigate the broader patterns of how teams utilize GitHub workflows across different repositories. We conduct quantitative analysis across 765 repositories with existing run records, examining the **utilization intensity spectrum** (run counts and failure rates) and **workflow trigger patterns** (events triggering workflow runs). This reveals how frequently run failures occur, whether intensive workflow adoption correlates with failure rates, and which event types and combinations trigger workflow executions in practice.

The workflow trigger analysis shows us what triggers runs based on the run records, rather than their configuration. In this way, we can distinguish whether workflows are activated as part of maintenance automation, or through development-driven triggers like a push or a pull request.

6.4.1 Utilization Intensity Spectrum

Repositories demonstrate an intensity spectrum from minimal to extensive workflow usage. Correlation analysis reveals a statistically significant negative relationship between run count and failure rate across 765 repositories: as workflow usage increases, failure rates tend to decrease. Pearson's coefficient assumes a linear relationship, so we report it alongside two coefficients that assume only monotonicity [96, 97]: Pearson's $r = -0.230$ ($p < 0.001$), Spearman's $\rho = -0.079$ ($p = 0.028$), and Kendall's $\tau = -0.067$ ($p = 0.007$). All three agree in sign and remain significant. This high usage can result from two distinct patterns: frequent triggering of workflows due to active development (*activity-intensive*), or comprehensive workflow configurations that execute many jobs per trigger (*configuration-intensive*). In both cases, whether 50 workflow runs result from one event triggering 50 jobs or 50 separate events each triggering one job, the total run volume contributes to the observed statistically significant negative correlation in Figure 6.3.

To discuss the different trends in Figure 6.3, we split the resitories into three visually distinct but roughly equally sized buckets: low-usage (<100 runs, 45%), medium-usage (100-499 runs, 26.5%), and high-usage (≥500 runs, 28.5%). These thresholds separate three observable behavioral patterns in Figure 6.3: the left cluster shows sporadic usage with high failure variability, the right cluster shows sustained automation with more stable failure rates, and the middle represents a transitional pattern[7].

High-Usage Pattern (28.5% with ≥500 runs): these repositories show high run volumes through two distinct approaches. `RailsEventStore/rails_event_store` exemplifies the *activity-intensive* approach with 1000 runs and 0.2% failure rate during two months, generating volume through frequent development activity supported by 960 scheduled runs[8] and an explicit culture of keeping all runs passing. `boutproject/BOUT-dev` represents the

[7] **Note on threshold selection:** The numerical thresholds (100 and 500 runs) presented in this section serve as descriptive categories for interpreting patterns within our specific sample, not as generalizable classification boundaries. These values emerged from visual analysis of our data distribution and are specific to our collection period and sampling approach. The primary contribution is identifying the pattern of systematically decreasing failure rates with increased usage intensity; the specific threshold values are artifacts of our sample and should not be applied to other contexts.

[8] The schedule event allows to trigger a workflow at a scheduled time, like a cron job.

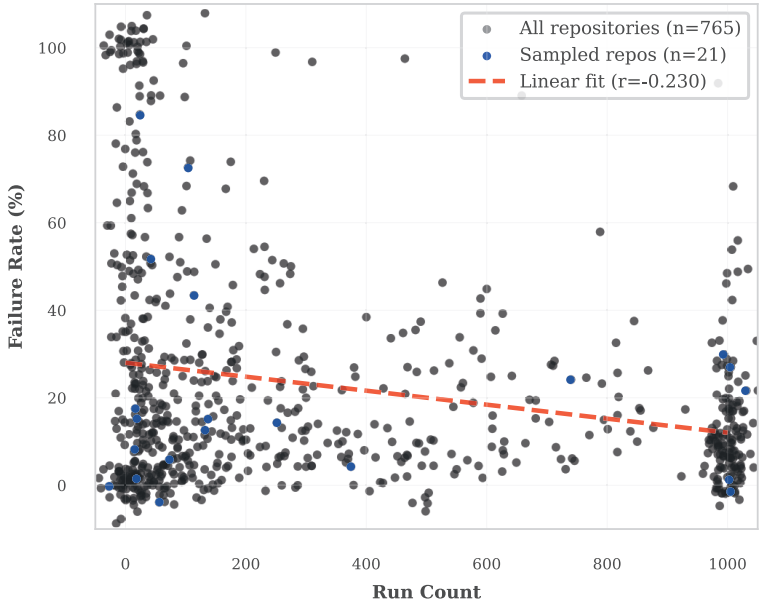

Figure 6.3: Repositories Run Count vs. Run Failure Rate. Points are jittered horizontally to reduce overplotting.

configuration-intensive approach with 1000 runs and 20.5% failure rate across 50+ workflow files during 3.3 months, reaching high run record volume through automation breadth while tolerating higher failure rates (e.g., ignoring platform-specific failures *"LGTM, fedora failure is unrelated"*).

Low-Usage Pattern (45% with <100 runs): these repositories show extreme variability in their run outcomes. Successful minimal usage includes `felix-fly/v2ray-openwrt` (2 runs, 0% failure), `prontolabs/pronto` (29 runs, 0% failure), and `trusttoken/contracts-pre22` (46 runs, 0% failure), exemplifying focused minimal automation. High failure rate minimal usage includes `iouAkira/someDockerfile` (15 runs, 86.7% failure rate): this project creates new workflow files per change and keeps pushing changes till they pass (using it as a tool to publish to docker hub). In `dotMorten/NmeaParser` (48 runs, 31.3% failure rate) the maintainer keeps pushing changes to fix failing workflows after making changes, which results in many workflow failure records. These usage patterns with high failure rates show experimental or trial-and-error approaches.

Medium-Usage Pattern (26.5% with 100-499 runs): these repositories show transitions between low and high-usage patterns. `webextension-toolbox/webextension-toolbox` shows 735 runs with a 21.2% failure rate, where 149 of 156 failures were bot-triggered, the PRs were then closed by the bots (e.g., #1071, #1070, #1068, #1067, #1065, etc.) indicating automation complexity without observable influence on the development. `libcpr/cpr` demonstrates 883 runs with 6.9% failure rate while explicitly accepting failures (*"CI failures are expected and I will fix them soon"*).

The key finding is that average failure rates decrease with higher usage intensity.

Table 6.4: Trigger events: distribution and combinations

(a) Top 10 Trigger Event Distribution			(b) Top 10 Trigger Event Comb.		
Event Type	**Runs**	**%**	**Event Combination**	**Repos**	**%**
push	76,822	36.0	PR + push	238	43.3
pull_request	76,400	35.8	push only	85	15.5
schedule	39,637	18.6	PR + push + sched	67	12.2
dynamic	6,387	3.0	dyn + PR + push	40	7.3
pull_request_target	4,399	2.1	PR + push + rel	26	4.7
issue_comment	2,650	1.2	schedule only	25	4.5
workflow_run	2,174	1.0	PR only	21	3.8
issues	1,743	0.8	PR + push + sched + WD	18	3.3
workflow_dispatch	1,654	0.8	PR + push + WD	15	2.7
release	616	0.3	dyn + PR + push + sched	15	2.7

sched = schedule; dyn = dynamic; rel = release; WD = workflow_dispatch

This pattern suggests that sustained utilization of workflow automation (whether through frequent usage or comprehensive configuration) correlates with lower failure rates, while sporadic or experimental usage results in more variable failure rate.

6

6.4.2 Workflow Trigger Patterns

Table 6.4-a summarizes our analysis of 213,640 non-bot-triggered[9] runs in 742 repositories, showing that teams primarily use development triggers, with selective adoption of scheduled triggers.

Development-Centric Trigger Events: we observe that `push` (36.0% of runs, primary in 332 repositories) and `pull_request` (35.8% of runs, primary in 250 repositories) account for 71.8% of all runs. The most common repository strategy combines both triggers: 238 repositories (32% of all repositories) use the `push` + `pull_request` combination, enabling coverage of both direct commits and PRs (Table 6.4-b).

Automation Focused Trigger Events: we see that teams selectively adopt automation-focused trigger events. `schedule` triggers account for 18.6% of runs and serve as the primary trigger in 108 repositories, similar to our observations in high-intensity utilizers like `aws-cloudformation/cfn-lint-visual-studio-code` with 878 scheduled runs (out of 1000) for automated schema updates. `dynamic` triggers (3.0% of runs) represent automated services like Dependabot and security scans, serving as primary triggers in 22 repositories but appearing across more repositories as supplementary automation (Table 6.2-b).

Repository Trigger Diversity Patterns: repositories demonstrate variety in trigger event usage, with an average of 2.5 events used per repository. We also observe that an increased event diversity in a repository correlates with having more run records (Figure 6.4). However, trigger concentration reveals strategic focus: 284 repositories (38.3%) concentrate more than 80% of their runs in a single trigger event. At the other extreme,

[9] Based on what we see in RQ2, we decided to exclude bot-triggered runs because they showed minimal meaningful reactions compared to user-triggered runs and could skew our results.

repositories like `BeyondDimension/SteamTools` utilize more trigger event diversity (791 `push` + 98 `pull_request` + 69 `create` + 42 `delete`).

The trigger pattern analysis reinforces our intensity findings: higher-usage repositories (≥100 runs) employ diverse trigger combinations, averaging 3.04 unique events, while low-usage repositories (<100 runs) average 1.90 trigger event types. Due to the non-linear relationship visible in Figure 6.4, we used Spearman's rank correlation [97], which reveals a strong positive correlation between trigger diversity and run count ($\rho = 0.589$, $p < 0.001$). This pattern is consistent: 33.7% of low-usage repositories rely on single triggers compared to 6.3% of higher-usage repositories.

RQ1 Summary

GitHub Actions utilization follows distinct patterns: **High-usage repositories** have low failure rates (2.7–20.5%) with more diverse trigger combinations, while **low-usage repositories** exhibit extreme variability (0–86.7% failure rates) with predominantly single-trigger strategies (33.7% use one trigger event). **Development-centric events triggers** dominate: `push` (36.0%) and `pull_request` (35.8%) account for 71.8% of runs. The negative correlation between run count and failure rates combined with strong correlation between trigger diversity and usage intensity indicates that repositories with sustained workflow usage develop more stable (less failure rate) and comprehensive workflow adoption (diverse trigger events).

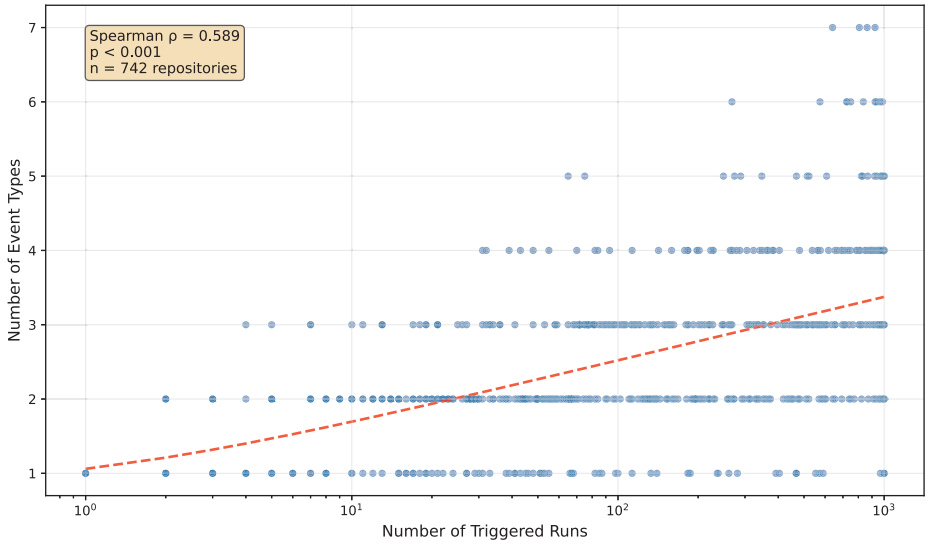

Figure 6.4: Trigger event diversity increasing as # of runs increase.

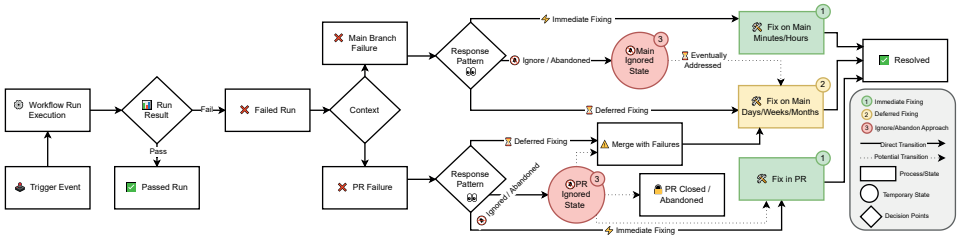

Figure 6.5: Developer Response Patterns to Workflow Failures in PR and Main Branch Contexts

6.5 RQ2: How do developers react to GitHub Actions workflow failures?

After understanding the broader workflow utilization of our large sample of 765 repositories, we investigate in more depth how developers react to workflow failures. As this focuses on understanding the context and unique situations of the project, we opt for a qualitative manual analysis.

Our analysis of workflow failure responses across our 21 sampled repositories (Section 6.3) reveals distinct patterns in how development teams handle GitHub Actions failures. The patterns are not mutually exclusive; projects can exhibit multiple response behaviors. Four repositories in our sampled dataset experienced no failures, so no reactions were observed for them.

We distinguished workflow failures (1) in pull requests (analyzed through PR comments and commits) and (2) on the main branch (tracked through subsequent pushes and timing), as these contexts present different response opportunities and constraints. Figure 6.5 illustrates how our observed failure reaction patterns occur. We now discuss each pattern.

6.5.1 Fix-Oriented Response Patterns

Immediate Fixing (13 repositories)

When developers address workflow failures directly within pull requests before merging, or by pushing commits directly to the main branch *immediately* within minutes or hours after (less than 24 hours).

Examples: `webextension-toolbox/webextension-toolbox` shows collaborative fixing: when a contributor's PR fails lint checks, the maintainer (reviewer of the PR) fixes it in the same PR (#866) or guides the contributor to fix it before merging (PR #890). In `trusttoken/contracts-pre22` the contributor makes multiple commits to resolve failed run issues before merging (PR #1264). In `data-driven-forms/react-forms` the contributor gets feedback from the PR reviewer and keeps pushing fix changes until all check runs pass before a merge (PR #1492). Developers also show immediate fixing behavior when directly pushing to the main branch of a repository. For example, in `ai/size-limit` when there is a commit with failed runs on the main branch, we see cases of immediate fixing in the subsequent commit(s) happening *shortly* after, e.g., commit 058b1f1 workflow runs pass 3 minutes after a commit with a failed run. In `BeyondDimension/SteamTools` we observe instances of fixing on the same day (commit 078d462, and commit 3633915).

`iouAkira/someDockerfile` shows the same behavior but with a different goal: the user keeps pushing changes until the run for Docker image builds passes, effectively treating GitHub Actions as a deployment service to push images to the Docker Hub. `RailsEventStore/rails_event_store` demonstrates an explicit culture of keeping all runs passing where failed commits are followed by immediate fixing commits (we did not observe a deferred fix approach for this repository). `aws-cloudformation/cfn-lint-visual-studio-code` shows a culture of ensuring workflows run successfully. The few run failures that we observed were fixed during the related PR.

Summary: immediate fixing, the most prevalent pattern observed in 76% of repositories with run failures, represents a proactive approach where teams resolve workflow failures within minutes to hours. This pattern manifests both in the pull-based development model [76, 85, 202] and in projects that directly commit to the main branch (Figure 6.5 – ①).

Deferred Fixing (6 repositories)

Occurs when teams accept temporary failures in their main branch and address them in subsequent commits. We distinguish between immediate and deferred fixes based on timing. If the failure occurs on the main branch and the commit that resolves the workflow failure happens days (≥ 24 hours) later, rather than within minutes or hours, we consider it a deferred fix.

Examples: in `libcpr/cpr` a maintainer states *"The CI failures are expected and I will fix them soon on master,"* showing a deferred fixing approach to address failures post-merge rather than blocking development velocity (#1173). `komamitsu/fluency` exhibits deferred fixing where a Dependabot version bump caused CI failures (commit c870a74), but the maintainer addressed the issue almost 2 months later through a separate PR (#915).

In `BeyondDimension/SteamTools` we observed multiple deferred fixing examples: PR #3469 merged with failed runs and was fixed 2 days later (commit 7b9ba5e), and a prolonged CI failure period of two months (between commits 150f93e and c087a02) caused by an UI framework upgrade, was eventually resolved through workflow configuration updates.

The deferred fixing approach spans varying timeframes, from days to months, reflecting different tolerances for temporary failures. `yandex-cloud/serverless-plugin` exhibits extended delays with NodeJS version-related failures taking almost a month after the first failure to be addressed through dependency upgrades, showing tolerance for dependency-related issues. `libcpr/cpr` demonstrates a deferred fixing with a 4-month gap between failed and successful workflow runs on master (between commit 372b54c and commit a9466f7), indicating acceptance of non-critical deployment failures until the next release cycle.

Summary: the deferred fixing approach reflects tolerance for temporary failures, with resolution timeframes spanning days to months. Teams employing this pattern appear to distinguish critical path failures requiring immediate attention and issues that can be addressed during planned maintenance or major updates. This approach prioritizes development velocity over passing workflow runs, accepting temporary failed runs in exchange for unblocked progress (Figure 6.5 – ②).

Ignore/Abandon Approach (4 repositories)

Describes workflow failures that receive no addressing action, representing a temporary state where teams either deliberately accept certain failures, face resource constraints preventing attention, or ultimately remove (or disable) failed workflows. In PRs, failures are ignored if they receive no reaction comments or fixing commits, or if the PR is closed/abandoned without resolution. On the main branch, failures remain ignored if they persist through subsequent commits without addressing action (Figure 6.5 – ③). Our study window is a snapshot: ignored failures may later transition to the deferred fixing pattern.

Examples: several repositories demonstrate explicit decisions to ignore certain workflow failures with clear justifications. `boutproject/BOUT-dev` demonstrates platform-specific ignoring where the reviewer explicitly states *"LGTM, fedora failure is unrelated"* when merging PRs with Fedora platform CI failures (#3029), and also states *"I am happy to remove the coverage run. I have not looked at the results in years..."* showing ignoring test coverage failures (#3067). This indicates an approach of dismissing environment-specific issues deemed non-critical. Similarly, in `data-driven-forms/react-forms` PR #1449 was merged while CodeCov coverage check failed, indicating tolerance for specific quality assurance metric failures. Another example is `ngageoint/geopackage-js`, where sometimes the last commit remains failing for a while (e.g., commit 3981bf0), potentially representing a conscious trade-off between CI/CD hygiene and development velocity.

Complete abandonment of workflow failures occurs when teams determine to remove or disable the workflow. `BeyondDimension/SteamTools` provides a clear example with marking `dotnet.yml` and `xamarin.yml` as *disabled* after persistent failures. The repository also shows workflows abandonment: the *publish* workflow never ran successfully after commit b40dac0, with the team stopping attempts to fix (till the time of our study). `trusttoken/contracts-pre22` demonstrates migration from GHA to CircleCI (abandoning GHA) and then again abandoning CircleCI by removing the workflow; also disabling their Coveralls test workflow. `iouAkira/someDockerfile` has 13 different workflow configurations but only 2 with run records, suggesting that most workflows were effectively disabled after initial setup.

Summary: the ignore/abandon approach reflects three distinct scenarios: deliberate acceptance of failed runs, resource constraints that prevent immediate attention to failed runs, or removal of workflows causing failures. Teams may ultimately resolve ignored failures through three pathways: transitioning to immediate or deferred fixing, continuing to leave them unaddressed, or removing failing workflows entirely. The temporary nature of this classification highlights that ignored failures represent a point-in-time observation rather than a permanent team strategy, with resolution approaches potentially evolving as project priorities and resources change.

RQ2 Summary

Our analysis of 21 repositories reveals three distinct response patterns: immediate fixing (13 of repositories) where teams resolve failures within minutes to hours, reflecting a commitment to passing continuous integration; deferred fixing (6 repositories) where teams strategically tolerate temporary failures, distinguishing between critical issues and those addressable in the future; and ignore/abandon (4 repositories) where failures receive no immediate action due to deliberate acceptance, potential resource constraints, or eventual workflow removal. These non-mutually exclusive patterns demonstrate that teams make nuanced prioritization decisions potentially based on failure context, project constraints, and strategic considerations.

6.6 RQ3: What relationships exist between project characteristics and GitHub Actions utilization patterns?

Understanding how project characteristics influence GitHub Actions utilization is fundamental for developing context-aware CI/CD research and practice [52]. In this section, we combine the insights from our quantitative analysis (Section 6.4) and in-depth qualitative analysis (Section 6.5). We examine relationships between project characteristics (popularity, development style, team size, workflow evolution, and activity level) relate to utilization patterns (RQ1) and failure reactions (RQ2). Given our limited sample size from our qualitative analysis, we deliberately emphasize these as hypotheses that represent a research agenda for future quantitative validation in large-scale mining studies.

6.6.1 Project Characteristics and Workflow Utilization Patterns

Table 6.5 presents workflow utilization metrics grouped by project characteristics. These observed patterns inform our hypotheses (H1-H3) for future validation.

Activity Level: high-activity projects show the most intensive workflow usage with 620.9 average runs and the highest event diversity (3.1), but also experience higher failure rates (28.2%). Medium-activity projects achieve the lowest failure rates (7.8%) despite moderate usage (258 runs). Low-activity projects show limited workflow adoption (143 runs) with surprisingly higher failure rates (23.2%), potentially due to experimental or one-off task usage (e.g. `iouAkira/someDockerfile`).

Workflow Evolution: projects with high workflow evolution demonstrate the most intensive usage patterns (827 runs) and highest event diversity (3.6), but also highest failure rates (30.3%). Projects with no workflow evolution show minimal usage (38.5 runs) with the lowest failure rates (3.2%). Moderate evolution balances usage (459 runs) with failure rates (22.8%) closer to the overall average (19.7%).

Team Size: multi-maintainer projects have the highest run counts (413.8), yet lowest failure rates (10.8%). Solo projects demonstrate moderate usage (339 runs) but high failure rates (29.8%), while single-maintainer projects show the lowest usage (282.6 runs) with

Table 6.5: Workflow utilization metrics (from RQ2).

Characteristic	Category	N	Runs	Fail %	Div.
Activity	High	7	620.9	28.2	3.1
	Medium	7	258.0	7.8	2.4
	Low	7	143.0	23.2	2.6
Workflow Ev.	High	5	827.0	30.3	3.6
	Moderate	5	459.0	22.8	3.4
	Minimal	7	81.3	19.5	2.0
	No	4	38.5	3.2	2.0
Team Size	Solo	3	339.0	29.8	2.3
	1 maint.	10	282.6	23.9	2.8
	Multi maint.	8	413.8	10.8	2.8
Dev. Style	Direct push	2	8.5	43.3	1.5
	Mixed	11	343.9	22.7	2.7
	Pull-based	8	419.1	9.7	3.0
Popularity	High	10	455.5	20.7	2.8
	Low	11	236.2	18.9	2.6

Runs = average workflow runs; Fail % = average failure rate; Div. = average trigger event diversity

intermediate failure rates (23.9%). Event diversity is consistent across team size (2.3-2.8), suggesting trigger complexity is independent of team structure.

Development Style: pull-based development has the highest workflow usage (419.1 runs) and event diversity (3) with the lowest failure rates (9.7%). Direct push approaches show minimal workflow adoption (8.5 runs) with high failure rates (43.3%). Mixed approaches balance moderate usage (343.9 runs) with moderate failure rates (22.7%).

Popularity: high-popularity projects show nearly double the workflow usage of low-popularity projects (455.5 vs. 236.2 runs) with comparable failure rates (20.7% vs. 18.9%), suggesting that project popularity may drive workflow adoption intensity. Event diversity shows minimal variation (2.8 vs. 2.6), indicating that trigger patterns are consistent regardless of popularity.

Hypothesis H1

Pull-based development is linked to lower workflow failure rates compared to direct push approaches.

Hypothesis H2

Projects with higher workflow evolution intensity experience higher failure rates, suggesting that frequent configuration changes are linked to increased workflow failures.

Table 6.6: Workflow failure reaction patterns (from RQ1)

Characteristic Categories	Total	Immediate Fixing (%)	Deferred Fixing (%)	Ignore /Abandon (%)
Activity Level				
High	7	5 (71.4)	4 (57.1)	4 (57.1)
Medium	7	4 (57.1)	2 (28.6)	0 (0.0)
Low	7	4 (57.1)	0 (0.0)	0 (0.0)
Workflow Evolution				
High	5	3 (60.0)	4 (80.0)	4 (80.0)
Moderate	5	4 (80.0)	0 (0.0)	0 (0.0)
Minimal	7	4 (57.1)	1 (14.3)	0 (0.0)
No	4	2 (50.0)	1 (25.0)	0 (0.0)
Team Size				
Solo project	3	2 (66.7)	0 (0.0)	0 (0.0)
1 maintainer	10	5 (50.0)	3 (30.0)	2 (20.0)
Multiple maintainers	8	6 (75.0)	3 (37.5)	2 (25.0)
Development Style				
Direct push	2	1 (50.0)	0 (0.0)	0 (0.0)
Mixed	11	8 (72.7)	2 (18.2)	2 (18.2)
Pull-based	8	4 (50.0)	4 (50.0)	2 (25.0)
Popularity				
High	10	6 (60.0)	3 (30.0)	3 (30.0)
Low	11	7 (63.6)	3 (27.3)	1 (9.1)

Note: Four repositories had no failures. Workflow failure reaction patterns are not mutually exclusive, so percentages do not add up to 100%.

Hypothesis H3

Team size is negatively linked to workflow failure rates, with projects having multiple maintainers showing lower failure rates than solo projects, and single-maintainer projects falling between these extremes.

6.6.2 Project Characteristics and Workflow Failure Reaction Patterns

Table 6.6 summarizes how projects with different characteristics react to workflow failures. These observed patterns inform our hypotheses (H4 and H5) for future validation.

Activity Level: projects with high activity levels demonstrate the most diverse failure reaction strategies, with 71.4% employing immediate fixes, and 57.1% using both deferred fixes and ignore approaches. This suggests that actively developed projects have more flexibility in their failure response strategies. In contrast, projects with low activity exclusively use immediate fixes when they address failures, never showing deferred or ignore fixing among our studied projects, potentially reflecting their more focused scope and simpler

workflows.

Workflow Evolution: a pattern emerges where projects with frequent workflow configuration updates show the highest rates of both deferred fixing (80%) and ignoring failures (80%), while projects with moderate evolution demonstrate the opposite behavior: 80% immediate fixing with no deferred or ignore strategies. This counterintuitive finding suggests that frequent workflow changes may create technical debt that teams manage through selective attention to failures.

Team Size: projects with multiple maintainers show a higher rate of immediate fixes (75%), compared to single-maintainer projects (50%). Solo projects never employ deferred or ignore strategies. This suggests that projects with multiple maintainers are more likely to have diverse failure response strategies, potentially reflecting prioritization, distributed responsibilities, and project complexity.

Development Style: development approaches combining PRs with direct pushes (mixed) are more likely to do immediate fixes (72.7%), while pull-based development shows equal distribution between immediate (50.0%) and deferred fixes (50.0%).

Popularity: repository popularity shows little to no relation with failure reaction patterns, with both high and low popularity projects demonstrating similar distributions across approaches. This indicates that reactions to workflow failures are independent of repository popularity.

Hypothesis H4

Projects with high workflow evolution rates are more likely to employ selective failure fixing strategies (deferred and ignore approaches), potentially indicating that rapid configuration changes create technical debt that teams should prioritize selectively.

Hypothesis H5

Projects with multiple maintainers are more likely to employ immediate fixing strategies than single-maintainer projects.

6.7 Discussion

6.7.1 Collaborative vs. Direct Fixing in Workflow Failures

Our observations reveal that while contributors may resolve workflow failures independently, maintainer involvement creates a duality in fixing approaches: maintainers either (1) guide external contributors to fix issues themselves, or (2) directly resolve failures within the PR. This pattern can reflect decisions about knowledge transfer when maintainers choose to intervene in case of workflow run failure in a pull request. When maintainers provide guidance, failures become learning opportunities for contributors to understand project context [203]. When maintainers fix directly, development velocity is preserved but the knowledge sharing value is lost, potentially affecting contributor retention.

6.7.2 Local Optimization vs. Collective Costs in Deferred Fixing

The deferred fixing approach that we observe in nearly 1/3 of our sampled repositories, reveals an asymmetry: while teams may make locally rational decisions to defer fixes, this can impose hidden costs on the development community.

We saw teams may make calculated decisions about failure priority, treating certain workflow issues as manageable technical debt, e.g., deferred fixing. Maintainers may reasonably defer fixes for infrastructure issues or dependency problems that do not block core development workflows. However, this local optimization can create problematic downstream effects. When subsequent contributors encounter failing builds, they may assume their changes caused the failure, leading to wasted debugging effort. This *failure contamination* represents an asymmetric cost distribution where one team's deferred fix may generate time waste elsewhere.

Additionally, normalization of failing workflows, may diminish contributor confidence in GHA workflows and reduce sensitivity to genuine new issues.

6.7.3 The Configuration-Usage Gap

Our approach of examining workflow run data rather than static configuration files, reveals methodological considerations for GHA workflows research. While the majority of previous studies analyzing GHA adoption focused on workflow configurations (Section 6.2), our study uncovered cases where configurations did not reflect actual adoption: disabled workflows (`nextcloud/ocsms`, with CodeQL scans disabled and no runs in years), unused configurations (`iouAkira/someDockerfile` with 13 configurations but only 2 active), and abandoned workflows. This suggests that configuration-based analyses may overestimate actual workflow adoption and may miss the nuanced reality.

6.7.4 Opportunities for Enhanced GitHub Actions Platform

Our findings illuminate areas where the current GHA platform can better support distributed development coordination:

Intelligent Failure Context: providing automated failure attribution and historical context to help contributors distinguish between inherited issues and newly-introduced problems, reducing debugging overhead.

Adaptive Mentoring Support: enhanced tooling could automatically generate project-specific guidance for common workflow failures, reducing maintainer mentoring burden while preserving contributor learning opportunities [15].

Usage-Aware Analytics: moving beyond configuration to usage-based insights would help teams understand automation adoption patterns and identify underutilized workflows.

6.7.5 Threats to Validity

Internal validity: Subjective interpretation of developer behavior was mitigated through dual coding and negotiated agreement between two authors. **External validity:** Our 21 repositories, while diverse across key dimensions, may not represent all GHA usage patterns. We frame RQ3 findings as hypotheses requiring future large-scale validation. **Construct validity:** Our observation of reactions may not fully reflect underlying team

intentions. The 1000-run collection limit may shorten data for highly active projects. The thresholds used in RQ1 are subjective and for interpretation only.

6.8 Conclusion and Future Work

We examined GitHub Actions adoption through workflow execution data and developer behavior, analyzing 21 repositories to understand how teams utilize automation in practice. Analyzing 765 repositories (RQ1) revealed that higher usage intensity correlates with lower failure rates: repositories with more runs have lower failure rates and use diverse triggers, while repositories with fewer runs show high variability with predominantly single-trigger strategies. Building on these findings, we identified three distinct failure response patterns (RQ2): *immediate fixing*, resolving failures within hours, *deferred fixing*, tolerating temporary failures for days to months, and *ignore/abandon* approaches, where failures receive no immediate action or lead to workflow removal. These patterns indicate teams make strategic prioritization decisions based on context. Finally, our mixed-methods analysis, combining quantitative metrics from 21 repositories with qualitative examination of their contexts, yielded five hypotheses for future validation (RQ3): workflows with more changes show more selective fixing strategies; multiple maintainers enable higher immediate fixing rates; pull-based development shows lower workflow failure rates; workflows with more changes have higher failure rates; and larger teams experience fewer workflow failures.

In terms of future work, we propose to (1) perform a *large-scale validation* across diverse repositories of 5 aforementioned hypotheses. Statistical analysis should establish the generalizability of relationships between project characteristics and workflow utilization patterns identified in our study. Additionally, (2) we need to better understand the cost-benefit relationship of different workflow strategies, e.g., the costs of a deferred fixing pattern. Finally, (3) we intend to investigate AI-assisted workflow guidance systems, and intelligent failure attribution to reduce the cost of fixing workflow failures.

Data Availability

All the data, data collection pipelines, and analyses are available in our replication package [200].

7 Conclusion

In this chapter, we summarize findings from the five studies in this dissertation, connecting them to the research questions introduced in Section 1.4. We then discuss implications for software engineering research and practice, and outline future research directions.

We begin by revisiting the overarching research goal:

> *To understand the complex software quality assurance (SQA) decision space by establishing a holistic understanding of the SQA ecosystem on GitHub.*

This dissertation achieved this goal by following a thread of interconnected discoveries. We began by mapping the SQA landscape empirically: Chapter 2 revealed variation in the adoption and combined use of SQA practices, raising a deeper question about whether developers actually understand the mechanisms already present in their projects. Chapter 3 confirmed this concern, uncovering an *awareness gap*. This gap motivated Chapter 4, which explored whether analytics tooling could bridge it, only to reveal that metrics alone are insufficient without richer contextual information. That finding pointed us toward GitHub Actions itself as the platform where such context naturally resides, leading to Chapters 5 and 6, which examined both the static quality of workflow configurations and the dynamic reality of how teams actually engage with them at runtime.

Together, these studies deepen our understanding of how SQA practices are adopted, perceived, and utilized in the studied GitHub projects. This chapter revisits each research question, synthesizes the findings into a proposed conceptual framework for navigating SQA design decisions, and discusses implications and future research directions.

7.1 Revisiting the Research Questions

RQ I: What is the current state of practice regarding SQA adoption in OSS projects, and how are different SQA practices used in conjunction?

In Chapter 2, we established an initial understanding of the SQA landscape by analyzing 1,454 popular Java projects. While individual adoption of practices varies, with approximately 63% of projects using Continuous Integration (CI) and 38% using Automated

Static Analysis Tools (ASATs), the critical finding is that projects typically do not employ all available SQA practices together with high intensity.

Our analysis revealed only weak correlations between the adoption of different SQA practices. For example, we observed a weak positive correlation between code coverage levels and code review intensity, and between ASAT usage and CI. This pattern suggests that SQA practices are often treated in isolation rather than as components of a cohesive quality assurance system; projects investing heavily in one area (e.g., testing) do not necessarily invest equally in others (e.g., code review). However, when we zoomed in on "mature" projects (characterized by high activity and popularity), we observed that they tend to invest more intensely across *all* SQA practices, suggesting that maturity may drive a more holistic adoption strategy.

Key insight (RQ I)

SQA adoption and usage intensity vary across the studied projects. Investment in one practice is not a reliable indicator of investment in another, highlighting the importance of examining multiple SQA dimensions together.

RQ II: To what extent are OSS developers aware of the SQA practices and mechanisms implemented in their projects?

The adoption patterns identified in RQ I raised a fundamental question: do developers understand the SQA mechanisms already present in their projects? In Chapter 3, we investigated this cognitive dimension through a survey of 471 open-source maintainers and contributors.

We discovered an *awareness gap* with two characteristics. First, developers possess strong high-level awareness: 79% knew CI was configured, 87% knew tests existed, but fewer respondents knew specific details about these mechanisms. Only 47% of developers were aware of their project's test coverage levels; awareness of specific bots, GitHub Actions configurations, and code review benefits proved even lower.

Second, awareness correlates with role: maintainers demonstrated higher awareness than contributors across most dimensions, though contributors showed greater awareness of contribution guidelines and reviewer availability. This suggests that awareness distribution within teams is neither uniform nor comprehensive.

Key insight (RQ II)

Knowing that a practice exists is not the same as understanding it. OSS developers are broadly aware of *what* SQA mechanisms are in place, but lack the deeper situational awareness needed to engage with them meaningfully.

RQ III: How can software analytics tools enhance developers' understanding of and reflection on SQA practices?

Given the awareness gap identified in RQ II, Chapter 4 explored whether aggregated analytics could bridge this gap. We designed RepoInsights, a dashboard consolidating data from testing, code review, and automation workflows, and evaluated it with 14 software engineers.

The dashboard provided value to the participants, offering a global overview and comparative metrics across SQA dimensions. Participants found reflection on SQA adequacy uncommon in their daily practice (except for testing), making the consolidated view novel and useful. However, the study revealed critical limitations in analytics-driven awareness.

Participants requested higher data granularity and richer contextual information. Metrics alone (e.g., "coverage is 60%" or "5 workflows configured") were insufficient. Engineers needed more context: *why* did a workflow fail? *how* does our coverage compare to *similar* projects? *what* should we prioritize?

Key insight (RQ III)

Analytics make SQA information visible and support reflection, but transforming awareness into actionable understanding requires deep contextual knowledge that aggregated metrics alone cannot provide.

RQ IV: How is GitHub Actions utilized in practice for software quality assurance?

The need for contextual depth identified in RQ III directed our attention to GitHub Actions, the automation platform orchestrating multiple SQA practices within a single, observable ecosystem. We examined GHA through two complementary lenses: static configuration quality and dynamic runtime behavior.

RQ IV.a: What quality issues emerge in GitHub Actions workflow configurations that may impact SQA effectiveness?

Chapter 5 investigated whether automation configurations suffer from quality issues analogous to code smells. By mining change history across 83 projects (10,012 commits), we identified 22 potential *workflow smells*, patterns that developers repeatedly modified, suggesting suboptimal initial configurations.

Through a contribution study involving 32 pull requests, we validated 7 workflow smells out of an identified list of 22 candidates. These workflow smells point to configuration issues such as: setting permissions for GitHub tokens, stopping workflow runs when there is a newer commit in a PR or in a branch, avoiding jobs without timeouts, avoiding running CI actions when no source code has changed, avoiding executing scheduled workflows on forks, and avoiding uploading artifacts on forks.

Key insight (RQ IV.a)

Automation configurations, like source code, accumulate quality issues over time and require active maintenance. Practitioners' acceptance of proposed fixes supports the practical relevance of the corresponding workflow smells in the participating projects.

RQ IV.b: How do GitHub Actions workflows behave in practice, and how do development teams respond to workflow failures?

Chapter 6 moved beyond static YAML files to examine what actually happens when workflows execute, combining a quantitative analysis of 700+ repositories with a qualitative

investigation of 21 repositories. Qualitative analysis revealed a *configuration-usage gap*: the presence of a GHA workflow configuration file does not guarantee active utilization.

We found workflows that were disabled, never triggered, or consistently ignored when failing. Through this analysis, we identified three distinct response patterns when workflows fail; (1) *Immediate Fixing*: Teams resolve failures within hours, treating workflow failures as blocking issues. (2) *Deferred Fixing:* Teams tolerate failures for days or months, eventually addressing them but not prioritizing immediate resolution. (3) *Ignore/Abandon:* Teams take no action, merging pull requests despite failing checks or abandoning workflows entirely.

Critically, in our quantitative analysis of 258,300 workflow runs , we observed that usage intensity correlates with less workflow failure rates: repositories with higher workflow activity exhibited lower failure rates and more sophisticated trigger configurations with diverse trigger events. Active engagement with the automation platform appears to drive more stable SQA processes. However, these associations do not establish whether greater activity improves reliability, whether more reliable workflows encourage greater use, or whether other project characteristics explain both.

Key insight (RQ IV.b)

Configuration is necessary but insufficient. Effective SQA automation requires ongoing engagement: teams should not only define workflows but actively reflect on their signals. The gap between static configuration and dynamic utilization represents a fundamental challenge in automation-driven quality assurance.

7.2 Navigating the Complex Decision Space of SQA Design

The introduction to this dissertation framed the central challenge facing software teams: a proliferation of SQA practices and tools has created a complex decision space requiring systematic knowledge to navigate effectively. Our five studies examined SQA practices from multiple perspectives, adoption patterns, developer awareness, analytics interventions, and automation platforms, to understand how teams can navigate the complex decision space of SQA. The availability of an advanced SQA tool does not ensure its optimal utilization, and the implementation of automation does not negate the complexities of human collaboration and infrastructure maintenance.

Synthesizing gaps and patterns observed across our studies, we propose a conceptual framework of six information elements that we hypothesize are relevant for teams navigating the SQA decision space. These elements are proposed as directions for future investigation rather than validated conclusions: our studies indicate their relevance, but do not empirically show their effectiveness in supporting navigation.

Figure 7.1 introduces the six elements of this proposed framework, each framed as a core question a team might ask when designing or improving their SQA system. To convey how the elements may relate to one another, the figure arranges them along three conceptual stages: three elements support *observing* the current state of a project's SQA (what is implemented, whether the team understands it, and how the team engages with it),

two elements support *interpreting* that state (against comparable projects and against the project's own context), and the prescriptive element synthesizes the others into concrete *actions*. The resulting decisions, in turn, reshape the project's SQA state, restarting the cycle. We emphasize that this arrangement illustrates one plausible interplay between the elements rather than an empirically validated sequence; in practice, teams may consult these elements in different orders or concurrently. The remainder of this section elaborates on each element, connecting it to our studies.

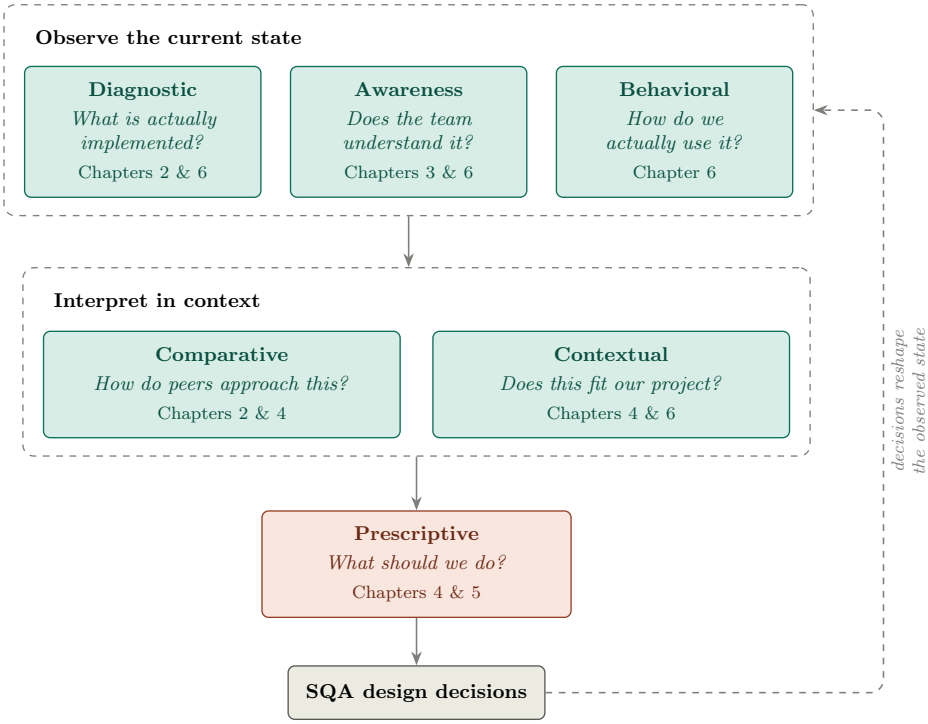

Figure 7.1: Proposed conceptual framework for navigating SQA design decisions. Teal denotes information supporting observation and interpretation, orange denotes prescriptive information, and grey denotes resulting design decisions. Arrows illustrate a possible feedback cycle rather than a validated sequence.

7.2.1 Hypothesis Related to Diagnostic Information

What SQA mechanisms are configured, and do they execute as intended?

Diagnostic information concerns which SQA mechanisms are configured and whether they execute as intended. Chapter 6 showed that the presence of a workflow configuration does not guarantee execution; Beller *et al.* report the same gap for ASATs [18]. Chapter 2 also encountered buildability obstacles when examining projects; such obstacles limit what researchers can verify, but do not establish that developers themselves never run the tests. These distinctions matter when assessing whether a mechanism is absent, inactive, or operational. We propose diagnostic information as a basis for assessing the current

technical state of SQA. Whether providing this information improves subsequent decisions remains to be empirically investigated.

7.2.2 Hypothesis Related to Awareness Information

Does our team understand the SQA mechanisms we have?

Beyond technical functionality lies a cognitive dimension: do team members understand what SQA mechanisms exist, how they operate, and what their signals mean? Our research identified systematic awareness gaps where tools functioned yet team members lacked knowledge of their configuration, triggers, or outputs (Chapters 3 and 6). Our survey of 471 maintainers and contributors revealed that while high-level awareness is common, deep situational awareness is lacking: only 47% of developers were aware of their project's test coverage levels, and awareness of specific bots, GitHub Actions configurations, and workflow details proved even lower.

This gap extends beyond individual knowledge to team-level distribution (Chapter 3). Maintainers, who typically author automation workflows and define SQA parameters, may have a a different mental model of the ecosystem than contributors who interact with the repository sporadically. Resolving this potential mismatch in mental models could enable contributors to move from trial-and-error interactions with the CI/CD pipeline towards a more proactive compliance. We propose awareness information as an element worth investigating for its role in reducing this cognitive friction, though whether closing the awareness gap translates directly into more effective SQA remains an open question.

7.2.3 Hypothesis Related to Behavioral Information

How does our team interact with SQA mechanisms?

Where diagnostic information asks whether SQA mechanisms function, behavioral information concerns whether the team engages with their outputs. A technically sound pipeline that is routinely ignored is, from a quality assurance perspective, perhaps as little effective as the one that never runs. A team may configure comprehensive CI checks, yet routinely merge despite failures (Chapter 6). Developers may receive ASAT warnings, yet ignore them [204]. Test suites may execute, yet produce results without them being examined.

The gap between functioning mechanisms and actual team response is what ultimately determines SQA effectiveness in practice. Understanding response patterns, such as, when teams fix failures immediately versus defer them, and which signals they prioritize versus ignore (Chapter 6), could enable the design of systems aligned with team capacity rather than sophisticated but neglected processes. Therefore, we propose behavioral information as a means of examining how teams prioritize, investigate, defer, or dismiss SQA signals, and whether those responses align with project goals and capacity. Its contribution to better SQA decisions requires further study.

7.2.4 Hypothesis Related to Comparative Information

How do comparable projects structure their SQA?

Comparative information situates a project's SQA practices within the broader ecosystem. In Chapter 2, we found that mature projects utilize SQA practices with higher intensity than the general population. Additionally, participants evaluating our RepoInsights

dashboard (Chapter 4) explicitly valued the comparative aspect, noting that metrics become meaningful when placed in the context of similar projects. Without such reference points, absolute metrics may offer limited actionable guidance (Chapter 4): knowing that coverage is 60% means something very different depending on the domain, team size, and project maturity.

We hypothesize that comparative information serves dual functions: (1) showing benchmarks at the metric level, where only metrics are displayed briefly and (2) revealing related strategies at the mechanism level by providing more details regarding how mechanisms are implemented in various projects. This may help teams move from intuition-based toward more evidence-based adoption. Whether access to such comparisons translates into improved SQA decisions is a direction that we consider worth investigating.

7.2.5 Hypothesis Related to Contextual Information

Which practices fit our specific situation?

Generic SQA guidance may fail because projects differ fundamentally in their contexts. Team size, development model, domain, contributor experience, project maturity, and organizational constraints all modulate what constitutes appropriate SQA investment (Chapter 4). Practices effective for mature open-source libraries may prove inappropriate for early-stage projects; strategies suited to large teams may overwhelm small ones. Our participants in Chapter 4 explicitly raised context as a factor when interpreting metrics, and our observations in Chapter 6 suggest that the development model influences how teams engage with GHA automation.

We propose contextual information as an element that filters general recommendations through project-specific characteristics, distinguishing universally applicable guidance from context-dependent advice. Its precise role in supporting navigation, however, warrants further investigation.

7.2.6 Hypothesis Related to Prescriptive Information

What specific actions should we take?

Our studies suggest that teams require not just awareness of problems but concrete guidance toward solutions. Chapter 5 demonstrated the potential of prescriptive information through our workflow smells catalog: by identifying specific, fixable patterns and providing concrete remediation strategies, we achieved a 70% acceptance rate in our contribution study. In a similar vein, Chapter 4 revealed the limitations of purely descriptive analytics. Participants valued the overview provided by RepoInsights but consistently requested actionable guidance, not just that coverage is low, but which modules to prioritize, and not just that a workflow failed, but why it failed and what to do about it.

Tools that go beyond reporting problems and suggest specific, contextually appropriate solutions could reduce the cognitive burden on teams and increase the likelihood that insights translate into action. We propose prescriptive information as the element that closes the loop from understanding to action, synthesizing the other five elements into concrete recommendations. Its effectiveness as part of a broader framework, however, remains to be empirically validated.

7.2.7 An Interconnected Framework

These six information elements (Figure 7.1) are proposed as interdependent rather than independent. Diagnostic information without comparative context may not establish adequacy. Comparative benchmarks without contextual filtering may suggest inappropriate practices. Behavioral insights without prescriptive guidance may identify problems but not solutions. A team that has full diagnostic clarity about its pipeline but lacks awareness of what those signals mean might still struggle to act. Conversely, a team with strong prescriptive guidance but no behavioral insight may follow recommendations that conflict with their actual capacity and workflow reality. Moreover, as Figure 7.1 suggests, the framework is cyclic rather than linear: acting on prescriptive guidance changes the project's SQA state, which renews the need for diagnostic, awareness, and behavioral information. Navigating the SQA decision space is therefore best understood as a continuous activity rather than a one-time assessment, which aligns with our earlier observation that SQA systems require ongoing engagement rather than one-off configuration (Chapter 6).

We hypothesize that effective SQA navigation benefits from addressing these elements holistically rather than in isolation. Current tool ecosystems typically address fragments of these elements: build systems provide diagnostic information, coverage tools report test metrics, CI platforms show workflow status. Yet these tools rarely connect diagnostics to comparisons, behavioral patterns to prescriptive actions, or awareness to context. This fragmentation may help explain why tool abundance has not eliminated navigation challenges, and points to integrated information infrastructure as a direction for both research and tool development. The interactions between these proposed information elements, their relative importance across different project contexts, and the conditions under which they most effectively support teams should be further uncovered in future research. Finally, it is important to note that this framework is likely incomplete, as additional information elements may emerge as SQA practices evolve and should be incorporated accordingly.

7.3 Implications

This section translates the findings of this dissertation into concrete implications for three audiences: software engineering practitioners designing SQA systems, tool builders creating platforms and analytics, and researchers advancing understanding of software quality assurance practices.

7.3.1 Implications: Software Practitioners

Treat SQA as an Active System Requiring Ongoing Attention

The gap between configuration and reality challenges the approach of configuring SQA mechanisms once and assuming they continue functioning. Teams should recognize that SQA systems require active maintenance (Chapter 5, 6).

Practitioners can implement periodic SQA audits that verify operational status. These audits should answer whether workflows execute when expected (Chapter 6), whether team members understand check results (Chapter 3), and whether failures are addressed or ignored (Chapter 6). Beyond technical verification, teams would benefit from explicit norms (e.g., through a contributing.md file) for responding to automation signals [132, 205]. Making it clear which failures block merging, which require timely investigation, and which

are merely informational, prevents the ambiguity that may lead to inconsistent responses. Finally, teams should prevent alerts from losing their impact. If they routinely merge despite failing checks (Chapter 6), the SQA system may have lost its purpose regardless of its technical sophistication.

Maintain Automation Infrastructure Alongside Code Artifacts

The fact that workflow configurations suffer from maintainability and security issues comparable to production code suggests that automation scripts may benefit from engineering practices (Chapter 5). Workflow configuration changes could undergo code review just as application code does [174], given that GHA YAML modifications can introduce security vulnerabilities or performance degradation similar to changes in production systems [55].

Extending static analysis to automation configurations is one concrete step in this direction. Linters and smell detectors for workflow configuration files are increasingly available and may help teams identify issues before they accumulate [174]. Similarly, budgeting time for automation refactoring and treating it as maintenance work rather than low priority cleanup may help prevent the gradual degradation of CI/CD configurations over time, in the same way that managing technical debt supports the long term sustainability of production code [55].

Design SQA as an Interconnected System

The weak correlations between practice adoption and the observed holistic strategies in mature projects suggest that isolated adoption may underutilize potential synergies between complementary mechanisms (Chapter 2). A starting point could be mapping current SQA practices explicitly by documenting which mechanisms exist, what they verify, and how they relate, as this may reveal gaps and redundancies that are invisible when practices are considered individually.

With a clearer understanding of existing mechanisms, teams may be better positioned to exploit complementarity intentionally. For instance, configuring ASATs to catch style issues before code review could allow reviewers to focus on logic and design, and using CI to enforce coverage thresholds may help ensure tests remain maintained as code evolves [31]. When adding new practices, considering their relationship to existing mechanisms, whether a new tool fills a gap, strengthens an existing check, or creates redundancy, could help prevent the accumulation of disconnected tools and practices that burden rather than support development workflows.

7.3.2 Implications: Tool Builders and Platform Providers

Integrate Context with Actionable Recommendations

Current tools tend to excel at providing diagnostic information but may struggle with comparative, contextual, and prescriptive information (Chapter 4): what this means in a specific project's case and what to do about it. This gap can leave the interpretation burden entirely on developers. Embedding comparative context in metric displays could help address this: rather than showing that coverage is 60%, tools might indicate how this compares to similar projects and whether it represents improvement or decline over time (Chapter 4).

Providing failure attribution and prioritization support is another direction worth exploring. When workflows fail, automatic classification as an infrastructure issue, flaky

test, or regression could help teams prioritize responses more effectively [206]. Similarly, tools that highlight actionable suggestions, e.g., showing untested modules, may reduce the cognitive effort required to act on analytical findings (Chapter 4).

However, recommendations of this kind should be treated as decision support rather than directives. For instance, whether a specific module warrants more testing may depend on project-specific factors that automated tools alone cannot fully capture. The value of actionable recommendations therefore lies not in replacing developer judgment, but in providing richer context that makes that judgment better informed.

Make the SQA Stack Visible and Understandable in OSS

The awareness gap between tool configuration and team understanding may stem partly from how platforms present, or obscure, automation infrastructure (Chapters 3 and 4). Making active SQA mechanisms more visible, for instance, by indicating which checks are enforced, which bots are active, and what quality gates protect the main branch, could help both existing team members and newcomers develop a clearer mental model of the project's SQA culture.

Providing explicit onboarding for contributors is one concrete direction [112, 205]. When new developers encounter a repository, guidance that explains not just contribution standards but the SQA mechanisms they will interact with, which checks must pass, what bot comments mean [43], how to interpret CI results, may reduce the friction observed in our study in Chapter 6. Contributing.md files already serve a related purpose in many repositories [112, 205], and extending their scope to include SQA-specific context could be a low-cost step toward closing the awareness gap, without requiring changes to the underlying platform infrastructure.

Support Behavioral Analysis and Reflection

Tools currently capture rich data about SQA behavior: execution logs, failure patterns, response times, but rarely highlight this information in ways that support team reflection (Chapter 4). Providing visualizations of behavioral patterns over time could help teams develop awareness of their own engagement with SQA mechanisms: how quickly they typically respond to different failure types, or which workflows tend to be consistently ignored. Making these implicit behavioral norms explicit may create opportunities for teams to discuss and intentionally adjust their practices.

Proactive flagging of deteriorating patterns is another direction worth exploring. When failure rates increase, response times lengthen, or ignore patterns emerge, timely signals to teams could help them intervene before SQA systems lose effectiveness (Chapter 6). Enabling comparative behavioral analysis, allowing teams to situate their engagement patterns relative to similar projects, could further support this reflection, though whether such comparisons translate into meaningful behavioral change remains an open question.

7.3.3 Implications: Software Engineering Researchers

Methodological: Verify Configuration Through Execution

The configuration-usage gap identified in our GHA investigation (Chapter 6) has broad methodological implications for mining software repositories research. Studies that infer tool adoption from configuration file presence alone (chapter 6) risk systematic overestimation of actual usage. Where feasible, combining static analysis with dynamic evidence,

for instance, verifying that configured workflows actually execute and that teams respond to their outputs, may produce more accurate characterizations of adoption in practice [18]. Distinguishing between adoption states may further strengthen such studies [207, 208]. A tool that is configured but unused, configured and sporadically used, or configured and actively maintained represents fundamentally different levels of adoption with different implications for understanding practice effectiveness. Where dynamic evidence is not feasible at scale and static analysis remains the only practical approach, explicitly acknowledging that reported adoption rates likely represent upper bounds rather than actual usage would help prevent misinterpretation of findings.

Theoretical: Develop Models of SQA System Design

Current understanding of SQA consists largely of knowledge about individual practices, and their effect on each other [58], and theoretical frameworks for understanding SQA as a system design challenge remain limited. One direction worth pursuing is the identification of SQA archetypes by clustering projects according to their SQA configurations, to discover whether common patterns emerge. Whether mature projects in similar domains converge on recognizable archetypes, and whether such patterns could guide teams toward more effective configurations, remains an open empirical question (Chapter 4).

Modeling interaction effects between practices could advance understanding beyond individual practice effectiveness. Developing and testing theories about how practices influence each other may reveal synergies and trade-offs that are invisible in isolated practice studies (Chapter 2). Building on this, decision support frameworks that suggest next practices to adopt based on current configuration, project context, and team capacity could help translate theoretical insights into practical guidance. Validating such frameworks through controlled studies or longitudinal observation remains an important research challenge.

7.4 Future Research Directions

Beyond the specific implications above, our work suggests several directions for advancing understanding of SQA ecosystems.

7.4.1 Large-Scale Study of GitHub Actions Workflow Behavior

Our investigation of GitHub Actions response patterns examined workflow executions across 21 repositories through detailed qualitative analysis (Chapter 6). Future work should scale this approach to hundreds or thousands of repositories to assess how widely the identified failure response patterns apply across diverse project types, including repositories that make use of reusable GHA workflow components [209] and code clones [210], to determine whether workflows with similar configurations or components exhibit similar failure response behaviors.

Expanding to larger samples would enable investigation of questions our current study could not address: Does deferred fixing correlate with increased bug resolution time? Does it affect contributor retention? Large scale analysis would reveal rare but significant patterns invisible in smaller samples. Moreover, quantitative analysis at scale could assess

how the prevalence of Immediate Fixing, Deferred Fixing, and Ignore/Abandon varies across project contexts, and whether additional response patterns emerge.

7.4.2 Investigating the Human and Cultural Dimensions

Our findings on awareness gaps and failure responses highlight human and organizational dimensions of SQA (Chapters 3 and 6). Future research should investigate how CI notifications affect developer experience and under what conditions they become overwhelming or receive less attention. Studying how teams negotiate quality standards could clarify when failed checks are treated as blocking, when exceptions are accepted, and how these decisions differ between open-source and industrial settings.

Research into learning and expertise development would illuminate how developers build mental models of complex SQA systems and what interventions accelerate the transition from novice to expert understanding of a project's quality assurance infrastructure. Finally, examining cultural variation in SQA practices, whether different organizational or domain cultures approach quality assurance fundamentally differently, could reveal alternative approaches and deepen understanding of contextual factors in SQA.

7.4.3 AI-Augmented SQA Infrastructure

The rise of capable large language models (LLMs) creates opportunities to address several information needs identified in our framework through AI assistance. Future research could evaluate whether automated failure diagnosis systems can analyze logs and historical patterns to distinguish infrastructure issues, flaky tests, and regressions. Such systems would support diagnostic information needs; their accuracy and effects on developers' workload would need to be evaluated.

Intelligent smell detection and remediation could extend our workflow smell catalog by training models to identify antipatterns and generate contextually appropriate fixes automatically (Chapter 5). Natural language SQA narratives could transform complex analytics into summaries that explain not just what metrics show but what they mean and what teams should consider doing, addressing prescriptive information needs. Personalized SQA recommendations that learn from project context and team behavior could suggest practice adoptions, configuration changes, or process modifications tailored to specific situations rather than generic best practices (Chapter 4).

7.4.4 Quality Assurance for AI-Generated Code

A very recent shift adds urgency to SQA. Autonomous coding agents now plan and carry out multi-step changes on their own, and *vibe coding*, i.e., describing what you want in plain language and refining the model's output instead of writing the logic yourself [211], is producing more code, faster, and with less human review, which widens the range of quality problems that SQA must catch. The most direct evidence so far is about the agents' own contributions. In the wild, agent-authored code tends to be changed more after it is merged, and is less likely to remain in the codebase, than human-authored code [1]; it is also checked unevenly, as some developers merge it without running any continuous integration [2]. These problems are not entirely new. Earlier AI coding tools were already found to write duplicated code [212], code and security smells [213], and insecure code in

security-sensitive situations [214], raising questions about how these risks change when generation is embedded in autonomous development workflows.

Future work should measure how common these problems are in AI- and agent-written code, test whether current SQA practices can catch them in practice, and adapt those practices where they fall short. AI is therefore both a tool for quality assurance and a new source of pressure on it.

7.4.5 Longitudinal Studies of SQA Evolution

Our work provides snapshots of SQA practices. Longitudinal research following projects over extended periods would illuminate how SQA systems evolve as projects mature and identify critical junctures where teams successfully adopt new practices or abandon failing ones. Understanding long term effects of different SQA strategies on project health metrics, e.g., contributor retention, defect rates, community growth, etc. would validate or challenge assumptions about practice effectiveness.

Tracking patterns in SQA technical debt accumulation and remediation would reveal when and why automation configurations degrade and what interventions prove effective for restoration. Such longitudinal understanding would enable evidence based guidance about when to invest in different practices, how to sequence adoption, and what warning signs indicate SQA system degradation requiring intervention.

7.5 Concluding Remarks

This dissertation began with a challenge: the proliferation of SQA practices and tools has created a complex decision space that teams struggle to navigate effectively. Through five research studies examining SQA adoption, awareness, and tool utilization in the open source software development, we have established a deeper understanding to facilitate this navigation challenge.

Software engineers should be aware that teams possess tools but might lack diagnostic clarity about whether these tools function. They configure mechanisms but might lack awareness of their operation. They adopt practices but might lack comparative context for evaluating adequacy. They define automation but might lack behavioral insight into actual usage. They receive metrics but might lack contextual interpretation and prescriptive guidance.

Addressing these information elements holistically through improved tools, enhanced platforms, better research methods, and deeper understanding of SQA as system design, represents the path toward more effective quality assurance in software development. The complex decision space can be navigated, but navigation requires comprehensive information infrastructure that current SQA mechanisms may only partially provide.

This dissertation contributes empirical evidence and a conceptual basis for understanding the connections between SQA tools, the people who use them, and the project contexts in which their signals inform decisions. In doing so, it supports software engineers in designing SQA systems that are both technically sound and meaningful in their specific contexts.

Bibliography

References

[1] Razvan Mihai Popescu, David Gros, Andrei Botocan, Rahul Pandita, Prem Devanbu, and Maliheh Izadi. Investigating autonomous agent contributions in the wild: Activity patterns and code change over time. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, MSR '26, pages 236–248, Rio de Janeiro, Brazil, 2026. ACM.

[2] Shamse Tasnim Cynthia, Joy Krishan Das, and Banani Roy. Are we all using agents the same way? an empirical study of core and peripheral developers' use of coding agents. In *Proceedings of the 23rd International Conference on Mining Software Repositories*, MSR '26, pages 520–532, Rio de Janeiro, Brazil, 2026. ACM.

[3] GitHub. About github. `https://github.com/about`, 2025. Last visited November 29th, 2025.

[4] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. The promises and perils of mining github. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 92–101. ACM, 2014.

[5] Jeetu Patel. Software is still eating the world. `https://techcrunch.com/2016/06/07/software-is-eating-the-world-5-years-later/`, 2016. Last visited November 29th, 2025.

[6] Mehdi Jazayeri. The education of a software engineer. In *Proc. International Conference on Automated Software Engineering (ASE)*, USA, 2004. IEEE.

[7] Amy J. Ko, Bryan Dosono, and Neeraja Duriseti. Thirty years of software problems in the news. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 32–39. ACM, 2014.

[8] Maurício Aniche. *Effective Software Testing: A Developer's Guide.* Manning Publications, 2022.

[9] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 195–204. IEEE Computer Society, 2014.

[10] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M. Germán, and Jesús M. González-Barahona. How bugs are born: a model to identify how bugs are introduced in software components. *Empir. Softw. Eng.*, 25(2):1294–1340, 2020.

[11] Gema Rodríguez-Pérez, Andy Zaidman, Alexander Serebrenik, Gregorio Robles, and Jesús M. González-Barahona. What if a bug has a different origin?: making sense of bugs without an explicit bug introducing change. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, pages 52:1–52:4. ACM, 2018.

[12] Khalid El Haji, Carolin Brandt, and Andy Zaidman. Using github copilot for test generation in python: An empirical study. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 45–55. ACM, 2025.

[13] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. Leveraging large language models for enhancing the understandability of generated unit tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1449–1461. IEEE, 2025.

[14] Baris Ardic, Mitchell Olsthoorn, and Andy Zaidman. Uncovering agents.md: What testing guidance open source projects provide to coding agents. In *Proceedings of the 26th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2026.

[15] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE Computer Society, 2013.

[16] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans. Software Eng.*, 39(6):757–773, 2013.

[17] Abir Bouraffa, Carolin Brandt, Andy Zaidman, and Walid Maalej. Not one to rule them all: Mining meaningful code review orders from github. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 349–359. ACM, 2025.

[18] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 470–481. IEEE, 2016.

[19] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empir. Softw. Eng.*, 25(2):1419–1457, 2020.

[20] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 805–816. ACM, 2015.

[21] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 356–367. IEEE, 2017.

[22] Akond Rahman, Asif Partho, David Meder, and Laurie Williams. Which factors influence practitioners' usage of build automation tools? In *International Workshop on Rapid Continuous Software Engineering (RCoSE)*, pages 20–26, 2017.

[23] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C. Gall. Continuous code quality: are we (really) doing that? In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 790–795. ACM, 2018.

[24] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437. ACM, 2016.

[25] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proceedings International Conference on Mining Software Repositories (MSR)*, pages 345–355. IEEE, 2017.

[26] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Jürgens. Modern code reviews in open-source projects: which problems do they fix? In *11th Working Conference on Mining Software Repositories (MSR)*, pages 202–211. ACM, 2014.

[27] Peter C. Rigby, Daniel M. Germán, and Margaret-Anne D. Storey. Open source software peer review practices: a case study of the apache server. In *International Conference on Software Engineering (ICSE)*, pages 541–550. ACM, 2008.

[28] Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. The silent helper: The impact of continuous integration on code reviews. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 423–434. IEEE, 2020.

[29] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 60–71. IEEE, 2017.

[30] Fiorella Zampetti, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. A study on the interplay between pull request review and continuous integration builds. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–48. IEEE, 2019.

[31] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 161–170. IEEE, 2015.

[32] Gustavo Sizílio Nery, Daniel Alencar da Costa, and Uirá Kulesza. An empirical study of the relationship between continuous integration and test code evolution. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 426–436. IEEE, 2019.

[33] Mika V. Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[34] Carolin E. Brandt and Andy Zaidman. Developer-centric test amplification. *Empir. Softw. Eng.*, 27(4):96, 2022.

[35] Mark Swillus and Andy Zaidman. Sentiment overflow in the testing stack: Analyzing software testing posts on stack overflow. *Journal of Systems and Software*, 205:111804, 2023.

[36] Mark Swillus, Rashina Hoda, and Andy Zaidman. On the emergence of testing strategies: A socio-technical grounded theory. *Empir. Softw. Eng.*, 31(5):147, 2026.

[37] Baris Ardiç and Andy Zaidman. Hey teachers, teach those kids some software testing. In *5th IEEE/ACM International Workshop on Software Engineering Education for the Next Generation (SEENG@ICSE)*, pages 9–16. IEEE, 2023.

[38] Andy Zaidman. An inconvenient truth in software engineering? the environmental impact of testing open source java projects. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, pages 214–218. ACM, 2024.

[39] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *36th International Conference on Software Engineering (ICSE)*, pages 72–82. ACM, 2014.

[40] Michael Hilton, Jonathan Bell, and Darko Marinov. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 53–63. ACM, 2018.

[41] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Trans. Reliab.*, 66(4):1213–1228, 2017.

[42] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 30–39. IEEE Computer Society, 2007.

[43] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A. Gerosa. Quality gatekeepers: Investigating the effects of code review bots on pull request activities. *Empirical Software Engineering*, 27(5):108, 2022.

[44] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Review participation in modern code review: An empirical study of the Android, Qt, and OpenStack projects (journal-first abstract). In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, page 475. IEEE Computer Society, 2018.

[45] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *36th International Conference on Software Engineering (ICSE)*, pages 345–355. ACM, 2014.

[46] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: how developers see it. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1028–1038. ACM, 2016.

[47] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[48] Mohammad Masudur Rahman and Chanchal K. Roy. Impact of continuous integration on code reviews. In *Proc. International Conference on Mining Software Repositories (MSR)*, pages 499–502. IEEE, 2017.

[49] Gemma Catolino, Fabio Palomba, Francesca Arcelli Fontana, Andrea De Lucia, Andy Zaidman, and Filomena Ferrucci. Improving change prediction models with code smell-related information. *Empir. Softw. Eng.*, 25(1):49–95, 2020.

[50] Alexander Sterk, Mairieli Wessel, Eli Hooten, and Andy Zaidman. Running a red light: An investigation into why software engineers (occasionally) ignore coverage checks. In *Proceedings of the International Conference on Automation of Software Test (AST)*, pages 12–22. ACM, 2024.

[51] Mica R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors*, 37(1):32–64, 1995.

[52] Ali Khatami, Carolin Brandt, and Andy Zaidman. Software quality assurance analytics: Enabling software engineers to reflect on QA practices. In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 189–200. IEEE, 2024.

[53] Pooya Rostami Mazrae, Alexandre Decan, Tom Mens, and Mairieli Wessel. A preliminary study of github actions workflow changes. In Andrea De Lucia, Dario Di Nucci, Valeria Pontillo, and Gilberto Recupito, editors, *Proceedings of the 15th Seminar on Advanced Techniques & Tools for Software Evolution, University of Salerno, Computer Science Department - Fisciano (Salerno, Italy), June 12 to 14, 2023*, volume 3483 of *CEUR Workshop Proceedings*, pages 78–88. CEUR-WS.org, 2023.

[54] GitHub. GitHub Actions | GitHub Features. `https://github.com/features/actions`, 2025. Accessed: 18 March 2026.

[55] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis CI. *IEEE Trans. Software Eng.*, 46(1):33–50, 2020.

[56] Ali Khatami, Cédric Willekens, and Andy Zaidman. Catching smells in the act: A github actions workflow investigation. In *IEEE International Conference on Source Code Analysis and Manipulation, SCAM 2024, Flagstaff, AZ, USA, October 7-8, 2024*, pages 47–58. IEEE, 2024.

[57] Paul Ralph, Sebastian Baltes, Domenico Bianculli, Yvonne Dittrich, Michael Felderer, Robert Feldt, Antonio Filieri, Carlo Alberto Furia, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara A. Kitchenham, Romain Robbes, Daniel Méndez, Jefferson Seide Mollëri, Diomidis Spinellis, Miroslaw Staron, Klaas-Jan Stol, Damian A. Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, and Sira Vegas. ACM SIGSOFT empirical standards. *CoRR*, abs/2010.03525, 2020.

[58] Ali Khatami and Andy Zaidman. State-of-the-practice in quality assurance in java-based open source software development. *Softw. Pract. Exp.*, 54(8):1408–1446, 2024.

[59] Abir Bouraffa, Carolin E. Brandt, Andy Zaidman, and Walid Maalej. Not one to rule them all: Mining meaningful code review orders from github. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 349–359. ACM, 2025.

[60] Hudson Borges and Marco Tulio Valente. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

[61] Ali Khatami and Andy Zaidman. "state-of-the-practice in quality assurance in java-based open source software development" replication package (version 1.0). `https://doi.org/10.5281/zenodo.7404903`, December 2022.

[62] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *J. Softw. Evol. Process.*, 29(4), 2017.

[63] Michel Maes-Bermejo, Micael Gallego, Francisco Gortázar, Gregorio Robles, and Jesús M. González-Barahona. Revisiting the building of past snapshots - a replication and reproduction study. *Empir. Softw. Eng.*, 27(3):65, 2022.

[64] Foyzul Hassan, Shaikh Mostafa, Edmund S. L. Lam, and Xiaoyin Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 38–47. IEEE, 2017.

[65] Foyzul Hassan and Xiaoyin Wang. Change-aware build prediction model for stall avoidance in continuous integration. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 157–162. IEEE, 2017.

[66] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: synthesizing travis CI and github for full-stack research on continuous integration. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 447–450. IEEE, 2017.

[67] Xianhao Jin and Francisco Servant. A cost-efficient approach to building in continuous integration. In *Proc. International Conference on Software Engineering (ICSE)*, pages 13–25. ACM, 2020.

[68] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. Understanding build issue resolution in practice: symptoms and fix patterns. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 617–628. ACM, 2020.

[69] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, pages 192–201. ACM, 2014.

[70] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 15(3):182–211, 1976.

[71] Mark Swillus, Carolin E. Brandt, and Andy Zaidman. An instrument to measure factors that constitute the socio-technical context of testing experience. *CoRR*, abs/2505.01171, 2025.

[72] Zeeger Lubsen, Andy Zaidman, and Martin Pinzger. Using association rules to study the co-evolution of production & test code. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR)*, pages 151–154. IEEE Computer Society, 2009.

[73] Sebastian G. Elbaum, David Gable, and Gregg Rothermel. The impact of software evolution on code coverage information. In *International Conference on Software Maintenance (ICSM)*, pages 170–179. IEEE, 2001.

[74] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empir. Softw. Eng.*, 16(3):325–364, 2011.

[75] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *First International Conference on Software Testing, Verification, and Validation (ICST)*, pages 220–229. IEEE Computer Society, 2008.

[76] Georgios Gousios, Andy Zaidman, Margaret-Anne D. Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 358–368. IEEE, 2015.

[77] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the IDE: patterns, beliefs, and behavior. *IEEE Trans. Software Eng.*, 45(3):261–284, 2019.

[78] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 179–190. ACM, 2015.

[79] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 559–562. IEEE Computer Society, 2015.

[80] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Trans. Software Eng.*, 40(11):1100–1125, 2014.

[81] Sonarqube. `https://www.sonarqube.org`. Last visited May 20th, 2022.

[82] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 197–207. ACM, 2017.

[83] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of CI build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193. IEEE Computer Society, 2017.

[84] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 495–498. IEEE Computer Society, 2017.

[85] Georgios Gousios and Andy Zaidman. A dataset for pull-based development research. In *Working Conf. on Mining Software Repositories (MSR)*, MSR 2014, pages 368–371. ACM, 2014.

[86] Xunhui Zhang, Ayushi Rastogi, and Yue Yu. On the shoulders of giants: A new dataset for pull-based development research. In *MSR '20: 17th International Conference on Mining Software Repositories*, pages 543–547. ACM, 2020.

[87] Timothy Kinsman, Mairieli Santos Wessel, Marco Aurélio Gerosa, and Christoph Treude. How do software developers use GitHub actions to automate their workflows? In *International Conference on Mining Software Repositories (MSR)*, pages 420–431. IEEE, IEEE, 2021.

[88] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[89] Matús Sulír and Jaroslav Porubän. A quantitative study of Java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, pages 17—-25. ACM, 2016.

[90] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. Lessons from eight years of operational data from a continuous integration service. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2022.

[91] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergõ Balogh, and Tibor Gyimóthy. Code coverage differences of java bytecode and source code instrumentation tools. *Softw. Qual. J.*, 27(1):79–123, 2019.

[92] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. A large-scale empirical study of the relationship between build technology and build maintenance. *Empir. Softw. Eng.*, 20(6):1587–1633, 2015.

[93] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979.

[94] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE, 2010.

[95] Hong Zhu, Patrick A. V. Hall, and John H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366––427, 1997.

[96] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. *A Modern Introduction to Probability and Statistics: Understanding why and how*, volume 488. Springer, 2005.

[97] Jacob Cohen, Patricia Cohen, Stephen G. West, and Leona S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Routledge, 3rd edition, 2013.

[98] Matús Sulír, Michaela Bacíková, Matej Madeja, Sergej Chodarev, and Ján Juhár. Large-scale dataset of local java software build results. *Data*, 5(3):86, 2020.

[99] Foyzul Hassan. Tackling build failures in continuous integration. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1242–1245. IEEE, 2019.

[100] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically repairing dependency-related build breakage. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 106–117. IEEE Computer Society, 2018.

[101] Amine Barrak, Ellis E. Eghan, Bram Adams, and Foutse Khomh. Why do builds fail? - A conceptual replication study. *J. Syst. Softw.*, 177:110939, 2021.

[102] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 111–120. IEEE Computer Society, 2015.

[103] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49. IEEE, 2018.

[104] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344. IEEE Computer Society, 2017.

[105] Richard Amankwah, Jinfu Chen, Heping Song, and Patrick Kwaku Kudjo. Bug detection in java code: An extensive evaluation of static analysis tools using Juliet test suites. *Softw. Pract. Exp.*, 53(5):1125–1143, 2023.

[106] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. On the rise and fall of CI services in GitHub. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 662–672. IEEE, IEEE, 2022.

[107] João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. Studying the impact of adopting continuous integration on the delivery time of pull requests. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 131–141. ACM, 2018.

[108] Akond Rahman, Amritanshu Agrawal, Rahul Krishna, and Alexander Sobran. Characterizing the influence of continuous integration: empirical results from 250+ open source and proprietary projects. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics, SWAN@ESEC/SIGSOFT FSE*, pages 8–14. ACM, 2018.

[109] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at Microsoft. In *12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, pages 146–156. IEEE, 2015.

[110] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. Why do software developers use static analysis tools? A user-centered study of developer needs and motivations. *IEEE Trans. Software Eng.*, 48(3):835–847, 2022.

[111] Yuqing Wang, Mika V. Mäntylä, Zihao Liu, and Jouni Markkula. Test automation maturity improves product quality - quantitative study of open source projects using continuous integration. *J. Syst. Softw.*, 188:111259, 2022.

[112] Ali Khatami and Andy Zaidman. Quality assurance awareness in open source software projects on github. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 174–185. IEEE, 2023.

[113] Maurício Aniche, Christoph Treude, and Andy Zaidman. How developers engineer test cases: An observational study. *IEEE Trans. Software Eng.*, 48(12):4925–4946, 2022.

[114] Gergõ Balogh, Tamás Gergely, Árpád Beszédes, and Tibor Gyimóthy. Are my unit tests in the right package? In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 137–146, 2016.

[115] Marco di Biase, Magiel Bruntink, and Alberto Bacchelli. A security perspective on code review: The case of chromium. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 21–30, 2016.

[116] DongGyun Han, Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, and Giovanni Rosa. Does code review really remove coding convention violations? In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 43–53, 2020.

[117] Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, and Andy Zaidman. UAV: warnings from multiple automated static analysis tools at a glance. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 472–476. IEEE, 2017.

[118] Fiorella Zampetti, Saghan Mudbhari, Venera Arnaoudova, Massimiliano Di Penta, Sebastiano Panichella, and Giuliano Antoniol. Using code reviews to automatically configure static analysis tools. *Empir. Softw. Eng.*, 27(1):28, 2022.

[119] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, and M.-A. Storey. Uncovering the benefits and challenges of continuous integration practices. *IEEE Transactions on Software Engineering*, 48(7):2570–2583, 2022.

[120] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. Un-break my build: assisting developers with build repair hints. In *Proceedings of the 26th Conference on Program Comprehension (ICPC)*, pages 41–51. ACM, 2018.

[121] Henri Bomström, Markus Kelanti, Elina Annanperä, Kari Liukkunen, Terhi Kilamo, Outi Sievi-Korte, and Kari Systä. Information needs and presentation in agile software development. *Information and Software Technology*, page 107265, 2023.

[122] Raymond P. L. Buse and Thomas Zimmermann. Information needs for software development analytics. In *34th International Conference on Software Engineering (ICSE)*, pages 987–996. IEEE Computer Society, 2012.

[123] Amy J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE)*, pages 344–353. IEEE Computer Society, 2007.

[124] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information needs in contemporary code review. *Proc. ACM Hum. Comput. Interact.*, 2(CSCW):135:1–135:27, 2018.

[125] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. Software professionals' information needs in continuous integration and delivery. In *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing*, pages 1513–1520. ACM, 2021.

[126] David Kavaler, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. Tool choice matters: Javascript quality assurance tools and usage outcomes in github projects. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 476–487, 2019.

[127] Mairieli Wessel, Andy Zaidman, Marco Aurélio Gerosa, and Igor Steinmacher. Guidelines for developing bots for github. *IEEE Softw.*, 40(3):72–79, 2023.

[128] Georgios Gousios, Margaret-Anne D. Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: the contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 285–296. ACM, 2016.

[129] M. Wessel, T. Mens, A. Decan, and P. Rostami Mazrae. The github development workflow automation ecosystems. In Tom Mens, Coen De Roover, and Anthony Cleve, editors, *Software Ecosystems: Tooling and Analytics*. Springer Nature, 2023.

[130] Omar Elazhary. Investigating the interplay between developers and automation. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*, pages 153–155. IEEE, 2021.

[131] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. A conceptual replication of continuous integration pain points in the context of travis CI. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 647–658. ACM, 2019.

[132] Omar Elazhary, Margaret-Anne D. Storey, Neil A. Ernst, and Andy Zaidman. Do as I do, not as I say: Do contribution guidelines match the github contribution process? In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 286–290. IEEE, 2019.

[133] Olga Baysal, Reid Holmes, and Michael W. Godfrey. Situational awareness: Personalizing issue tracking systems. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1185–1188. IEEE Computer Society, 2013.

[134] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. Data visualisation in continuous integration and delivery: Information needs, challenges, and recommendations. *IET Softw.*, 16(3):331–349, 2022.

[135] Pavneet Singh Kochhar, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *2013 13th International Conference on Quality Software (QSIC)*, pages 103–112. IEEE, 2013.

[136] Mairieli Santos Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco Aurélio Gerosa. What to expect from code review bots on github?: A survey with OSS maintainers. In *34th Brazilian Symposium on Software Engineering (SBES)*, pages 457–462. ACM, 2020.

[137] Tingting Chen, Yang Zhang, Shu Chen, Tao Wang, and Yiwen Wu. Let's supercharge the workflows: An empirical study of github actions. In *21st IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 1–10. IEEE, 2021.

[138] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of GitHub Actions in software development repositories. In *IEEE International*

Conference on Software Maintenance and Evolution, (ICSME), pages 235–245. IEEE, 2022.

[139] T. Zimmermann. Card-sorting: From text to themes. In Tim Menzies, Laurie Williams, and Thomas Zimmermann, editors, *Perspectives on Data Science for Software Engineering*, pages 137–141. Morgan Kaufmann, 2016.

[140] Juliet M Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology*, 13(1):3–21, 1990.

[141] Wenxin Xiao, Hao He, Weiwei Xu, Xin Tan, Jinhao Dong, and Minghui Zhou. Recommending good first issues in github oss projects. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 1830–1842. ACM, 2022.

[142] Jan Willem David Alderliesten and Andy Zaidman. An initial exploration of the "good first issue" label for newcomer developers. In *14th IEEE/ACM International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 117–118. IEEE, 2021.

[143] Casper Boone, Carolin E. Brandt, and Andy Zaidman. Fixing continuous integration tests from within the IDE with contextual information. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*, pages 287–297. ACM, 2022.

[144] Ali Khatami and Andy Zaidman. Quality assurance awareness in open source software projects on github analysis dataset (version 1). `https://doi.org/10.5281/zenodo.8139381`, July 2023.

[145] Tim Menzies and Thomas Zimmermann. Software analytics: So what? *IEEE Software*, 30(4):31–37, 2013.

[146] Olga Baysal, Reid Holmes, and Michael W. Godfrey. Developer dashboards: The need for qualitative analytics. *IEEE Softw.*, 30(4):46–52, 2013.

[147] Mairieli Wessel, Tom Mens, Alexandre Decan, and Pooya Rostami Mazrae. The github development workflow automation ecosystems. *CoRR*, abs/2305.04772, 2023.

[148] Christopher D. Wickens. The tradeoff of design for routine and unexpected performance: Implications of situation awareness. *Situation awareness analysis and measurement*, pages 211–225, 2000. In Endsley and Garland (Eds.), Situation Awareness Analysis and Measurement.

[149] MongoDB. Mern stack explained: https://www.mongodb.com/mern-stack, 2024.

[150] Donald T. Campbell and Julian C. Stanley. *Experimental and Quasi-Experimental Designs for Research.* Rand McNally, Chicago, 1963.

[151] Earl R. Babbie. *The Practice of Social Research.* Wadsworth, 11th edition, 2007.

[152] Ali Khatami, Carolin E. Brandt, and Andy Zaidman. Replication package for "software quality assurance analytics: Enabling software engineers to reflect on QA practices" paper (SCAM 2024) (version 1). `https://doi.org/10.5281/zenodo.10961021`, April 2024.

[153] Anselm L. Strauss and Juliet M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage Publications, 2nd edition, 1998.

[154] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine, 1967. Reprinted 2017 by Routledge.

[155] D Randy Garrison, Martha Cleveland-Innes, Marguerite Koole, and James Kappelman. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The internet and higher education*, 9(1):1–8, 2006.

[156] Tamer Mohamed Abdellatif, Luiz Fernando Capretz, and Danny Ho. Software analytics to software practice: A systematic literature review. In *1st IEEE/ACM International Workshop on Big Data Software Engineering (BIGDSE)*, pages 30–36. IEEE Computer Society, 2015.

[157] Santiago Dueñas, Valerio Cosentino, Jesus M. Gonzalez-Barahona, Alvaro del Castillo San Felix, Daniel Izquierdo-Cortazar, Luis Cañas-Díaz, and Alfonso Panadero Garza. GrimoireLab: A toolset for software development analytics. *PeerJ Computer Science*, 7:e601, 2021.

[158] Mark van den Brand, Serguei A. Roubtsov, and Alexander Serebrenik. Squavisit: A flexible tool for visual software analytics. In *13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 331–332. IEEE Computer Society, 2009.

[159] Silverio Martínez-Fernández, Anna Maria Vollmer, Andreas Jedlitschka, Xavier Franch, Lidia López, Prabhat Ram, Pilar Rodríguez, Sanja Aaramaa, Alessandra Bagnato, Michal Choras, and Jari Partanen. Continuously assessing and improving software quality with software analytics tools: A case study. *IEEE Access*, 7:68219–68239, 2019.

[160] Lidia López, Martí Manzano, Cristina Gómez, Marc Oriol, Carles Farré, Xavier Franch, Silverio Martínez-Fernández, and Anna Maria Vollmer. Qasd: A quality-aware strategic dashboard for supporting decision makers in agile software development. *Sci. Comput. Program.*, 202:102568, 2021.

[161] Harsh Mukeshkumar Shah, Qurram Zaheer Syed, Bharatwaaj Shankaranarayanan, Indranil Palit, Arshdeep Singh, Kavya Raval, Kishan Savaliya, and Tushar Sharma. Mining and fusing productivity metrics with code quality information at scale. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 563–567. IEEE, 2023.

[162] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015.

[163] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[164] Hassan Onsori Delicheh, Alexandre Decan, and Tom Mens. A preliminary study of GitHub Actions dependencies. In *Proceedings of the 15th Seminar on Advanced Techniques & Tools for Software Evolution*, volume 3483 of *CEUR Workshop Proceedings*, pages 66–77. CEUR-WS.org, 2023.

[165] Jessy Ayala and Joshua Garcia. An empirical study on workflows and security policies in popular GitHub repositories. In *Proceedings of the 1st International Workshop on Software Vulnerability Management (SVM)*, pages 6–9. IEEE, IEEE/ACM, 2023.

[166] Mairieli Wessel, Joseph Vargovich, Marco Aurélio Gerosa, and Christoph Treude. GitHub Actions: The impact on the pull request process. *Empir. Softw. Eng.*, 28(6):131, 2023.

[167] Alexandre Decan, Tom Mens, and Hassan Onsori Delicheh. Quantifying security issues in reusable JavaScript actions in GitHub workflows. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*, pages 692–703. ACM, 2024.

[168] Giacomo Benedetti, Luca Verderame, and Alessio Merlo. Automatic security assessment of GitHub Actions workflows. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 37–45. ACM, 2022.

[169] Sk Golam Saroar and Maleknaz Nayebi. Developers' perception of github actions: A survey analysis. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 121–130. ACM, 2023.

[170] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. Characterizing the security of GitHub CI workflows. In *Proceedings of the 31st USENIX Security Symposium*, pages 2747–2763. USENIX Association, 2022.

[171] Hassan Onsori Delicheh and Tom Mens. Mitigating security issues in GitHub actions. In *Proceedings of the 2nd International Workshop on Software Vulnerability (SVM) co-located with ICSE 2024*, pages 6–11. ACM, 2024.

[172] Islem Bouzenia and Michael Pradel. Resource usage and optimization opportunities in workflows of GitHub Actions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 25:1–25:12. ACM, 2024.

[173] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald C. Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25(2):1095–1135, 2020.

[174] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: a linter and

a six-month study on gitlab. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 327–337. ACM, 2020.

[175] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.

[176] Alexandre Decan, Tom Mens, and Hassan Onsori Delicheh. On the outdatedness of workflows in the github actions ecosystem. *J. Syst. Softw.*, 206:111827, 2023.

[177] Paul M. Duvall. Continuous integration: Patterns and antipatterns in the software lifecycle. `https://dzone.com/refcardz/continuous-integration`, 2010.

[178] Paul M Duvall and Michael Olson. Continuous delivery: Patterns and antipatterns in the software life cycle. *DZone refcard*, 145:64, 2011.

[179] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 105–115. IEEE / ACM, 2019.

[180] Thomas Zimmermann. Card-sorting: From text to themes. In *Perspectives on data science for software engineering*, pages 137–141. Elsevier, 2016.

[181] D.R. Garrison, M. Cleveland-Innes, Marguerite Koole, and James Kappelman. Revisiting methodological issues in transcript analysis: Negotiated coding and reliability. *The Internet and Higher Education*, 9(1):1–8, 2006.

[182] Ali Khatami, Cédric Willekens, and Andy Zaidman. Replication package for "catching smells in the act: A github actions workflow investigation" (SCAM 2024) (version 1). `https://doi.org/10.5281/zenodo.12207164`, June 2024.

[183] About billing for GitHub Actions: https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions, 2024. Accessed: 18-03-2026.

[184] Carolin E. Brandt, Ali Khatami, Mairieli Wessel, and Andy Zaidman. Shaken, not stirred: How developers like their amplified tests. *IEEE Trans. Software Eng.*, 50(5):1264–1280, 2024.

[185] Emad Shihab, Stefan Wagner, Marco A. Gerosa, Mairieli Wessel, and Jordi Cabot. The present and future of bots in software engineering. *IEEE Software*, 39(5):28–31, 2022.

[186] Thomas Durieux. An empirical study of GitHub actions workflow testing. In *Proceedings of the 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023.

[187] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering.* Springer, 2012.

[188] M. Fowler and M. Foemmel. Continuous integration: https://tinyurl.com/ycbl2uhj, 2006. [Online; accessed 18-March-2026].

[189] Eriks Klotins, Tony Gorschek, Katarina Sundelin, and Erik Falk. Towards cost-benefit evaluation for continuous software engineering activities. *Empirical Software Engineering*, 27(6):157, 2022.

[190] Jadson Santos, Daniel Alencar da Costa, Shane McIntosh, and Uirá Kulesza. On the need to monitor continuous integration practices. *Empirical Software Engineering*, 30(5), June 2025.

[191] Yang Zhang, Yiwen Wu, Tingting Chen, Tao Wang, Hui Liu, and Huaimin Wang. How do developers talk about GitHub Actions? Evidence from online software development community. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. ACM, 2024.

[192] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[193] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Le Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[194] Hassan Onsori Delicheh, Alexandre Decan, and Tom Mens. A preliminary study of GitHub actions dependencies. In *Proceedings of the Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*, volume 3483 of *CEUR Workshop Proceedings*, pages 70–77, 2023.

[195] Pooya Rostami Mazrae, Tom Mens, Mehdi Golzadeh, and Alexandre Decan. On the usage, co-usage and migration of CI/CD tools: A qualitative analysis. *Empirical Software Engineering*, 28(2):52, 2023.

[196] Tingting Chen, Yang Zhang, Shu Chen, Tao Wang, and Yiwen Wu. Let's supercharge the workflows: An empirical study of GitHub Actions. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 01–10. IEEE, 2021.

[197] Pooya Rostami Mazrae, Alexandre Decan, and Tom Mens. gawd: A differencing tool for GitHub actions workflows. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. ACM, 2024.

[198] Pablo Valenzuela-Toledo and Alexandre Bergel. Evolution of github action workflows. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 123–127. IEEE, IEEE, 2022.

[199] Sk Golam Saroar and Maleknaz Nayebi. Developers' perception of GitHub actions: A survey analysis. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, page 10. ACM, 2023.

[200] Ali Khatami, Carolin Brandt, and Andy Zaidman. Replication package for "Beyond the YAML file: Understanding real-world github actions workflow adoption", 2026. https://doi.org/10.5281/zenodo.18258226.

[201] D. Randy Garrison, Terry Anderson, and Walter Archer. Critical thinking, cognitive presence, and computer conferencing in distance education. *American Journal of Distance Education*, 15(1):7–23, 2001. Republished/cited as 2006.

[202] Erik van der Veen, Georgios Gousios, and Andy Zaidman. Automatically prioritizing pull requests. In *12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, pages 357–361. IEEE, 2015.

[203] Zhixing Li, Yue Yu, Tao Wang, Shanshan Li, and Huaimin Wang. Opportunities and challenges in repeated revisions to pull-requests: An empirical study. *Proc. ACM Hum.-Comput. Interact.*, 6(CSCW2), November 2022.

[204] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. Mitigating false positive static analysis warnings: Progress, challenges, and opportunities. *IEEE Trans. Software Eng.*, 49(12):5154–5188, 2023.

[205] GitHub. Setting guidelines for repository contributors. `https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/setting-guidelines-for-repository-contributors#about-contributing-guidelines`, 2026. Accessed: 2026-03-19.

[206] George Vegelien, Carolin Brandt, Bas Graaf, and Arie van Deursen. Addressing test flakiness: Practical approaches in a database-reliant industrial system. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '26, page 11, Rio de Janeiro, Brazil, 2026. ACM.

[207] Shujun Huang and Sebastian Proksch. A taxonomy of contextual factors in continuous integration processes. *IEEE Trans. Software Eng.*, 51(7):2067–2087, 2025.

[208] Shujun Huang, Andy Zaidman, and Sebastian Proksch. Does one ci-ze fit all? how continuous integration performs in different contexts. In *Proceedings of the 33rd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 491–502. IEEE, 2026.

[209] Hassan Onsori Delicheh, Guillaume Cardoen, Alexandre Decan, and Tom Mens. Automation and reuse practices in github actions workflows: A practitioner's perspective. *CoRR*, abs/2601.11299, 2026.

[210] Guillaume Cardoen, Tom Mens, and Alexandre Decan. An empirical analysis of code clones in github actions workflows. In *The 33rd IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2026)*. IEEE, 2026.

[211] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *CoRR*, abs/2505.19443, 2025.

[212] Weibin Wu, Haoxuan Hu, Zhaoji Fan, Yitong Qiao, Yizhan Huang, Yichen Li, Zibin Zheng, and Michael Lyu. An empirical study of code clones from commercial ai code generators. *Proceedings of the ACM on Software Engineering*, 2(FSE):2874–2896, 2025.

[213] Mohammed Latif Siddiq, Shafayat H. Majumder, Maisha R. Mim, Sourov Jajodia, and Joanna C. S. Santos. An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 71–82. IEEE, 2022.

[214] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

Glossary

API Application Programming Interface — a set of definitions and protocols that allow different software systems to communicate with each other; used extensively in this dissertation to collect workflow run records and repository metadata from GitHub.

ASAT Automated Static Analysis Tool — a program that examines source code without executing it, identifying potential bugs, code smells, style violations, and security vulnerabilities through pattern matching and abstract interpretation (studied in Chapter 2).

Awareness Gap Awareness Gap — an empirical finding of Chapter 3: while most open-source developers have high-level awareness of SQA mechanisms (e.g., that tests or CI exist), their situational awareness of how those mechanisms operate is substantially shallower; this gap motivates the analytics intervention of Chapter 4 and the GitHub Actions studies of Chapters 5 and 6.

Branch Coverage Branch Coverage — a test coverage metric measuring the percentage of decision branches (e.g., if-else conditions) that have been exercised during testing; a stricter criterion than line coverage.

Build Automation Build Automation — the automated process of compiling source code, resolving dependencies, running tests, and packaging software into a deployable artifact, typically orchestrated by tools such as Maven, Gradle, or Ant and verified on every commit in a CI/CD pipeline.

CCQ Continuous Code Quality — the practice of continuously monitoring software quality at every CI build by running automated tests and code inspections, popularized by the SonarQube platform.

CD Continuous Deployment — a software engineering practice that extends Continuous Integration by automatically deploying every change that passes the CI pipeline to production environments, thereby accelerating the release cycle.

CI Continuous Integration — a software engineering practice in which developers frequently integrate code changes into a shared repository; each integration is automatically verified by building the project and running automated tests so that problems are detected early.

CI/CD Continuous Integration and Continuous Deployment — the combined practice of automatically integrating, testing, and deploying software; a central SQA mechanism studied throughout this dissertation.

Code Review Code Review — the systematic examination of source code by developers other than the original author; in this dissertation the term specifically refers to Modern Code Review as practiced on GitHub through the pull request mechanism.

Code Smell Code Smell — a surface indicator in source code that suggests a deeper design or implementation problem without necessarily being a bug; the concept is extended to workflow configurations through the notion of *workflow smells* in Chapter 5.

Configuration–Usage Gap Configuration–Usage Gap — the disconnect observed in Chapter 6 between the mere presence of a GHA workflow configuration file and the actual, active utilization of that workflow at runtime; configuration files alone are therefore an unreliable proxy for real adoption.

Design Science Design Science — a research methodology used in Chapter 4 that builds and evaluates purposeful artifacts (here the RepoInsights prototype) to solve identified organizational or technical problems.

End-to-End Test End-to-End Test — a software test that validates a complete user scenario by simulating real user interactions with the deployed application, verifying that entire workflows function correctly from the user's perspective.

Failure Response Pattern Failure Response Pattern — one of three behavioral patterns identified in Chapter 6 that characterize how development teams react when a GHA workflow run fails: *Immediate Fixing* (the failure is addressed promptly in the same pull request or the next commit), *Deferred Fixing* (the fix is postponed to a later point), or *Ignore / Abandon* (the failure is left unresolved, sometimes leading to the abandonment of the pull request).

GHA GitHub Actions — a CI/CD and workflow automation platform deeply integrated into the GitHub ecosystem, introduced in November 2019; GHA allows developers to define automated workflows triggered by repository events such as pushes, pull requests, and releases using YAML-based configuration files stored in the `.github/workflows/` directory; studied in Chapters 5 and 6.

IDE Integrated Development Environment — a software application that combines a source-code editor, build tools, and a debugger into a single interface (e.g., Eclipse, IntelliJ, Visual Studio).

JaCoCo Java Code Coverage Library — a free code coverage library for Java that instruments bytecode at runtime; injected into Maven/Gradle build scripts in Chapter 2 to measure test coverage.

Line Coverage Line Coverage (Statement Coverage) — a test coverage metric measuring the percentage of source code lines executed during a test run; together with branch coverage it is used in this dissertation as a proxy for testing intensity.

MCR Modern Code Review — the lightweight, tool-supported form of code inspection practice on platforms such as GitHub, as coined by Bacchelli and Bird [15]; distinct from heavyweight formal inspections, MCR is conducted asynchronously through pull request discussions and serves multiple purposes including defect detection, knowledge transfer, and enforcement of coding standards.

MSR Mining Software Repositories — an empirical research methodology that extracts and analyses data from version control systems, issue trackers, CI logs, and other software artifacts to derive insights about software development practices; the primary method used in Chapters 2, 5 and 6.

Open Card Sorting Open Card Sorting — a qualitative categorization method in which participants group items without pre-defined categories, allowing themes to emerge naturally from the data; used in Chapter 3 to classify developer responses and in Chapter 5 to categorize workflow change patterns.

OSS Open Source Software — software whose source code is publicly accessible, typically hosted on collaborative platforms such as GitHub; the primary research context of this dissertation.

PR Pull Request — a mechanism in the pull-based development model through which a contributor proposes a set of code changes for integration into the main codebase; maintainers and reviewers examine, discuss, and ultimately accept or reject the proposed changes before merging.

Pull-Based Development Pull-Based Development — a collaborative software development model in which contributors submit changes as pull requests that are reviewed and merged by project maintainers; the dominant contribution model on GitHub and a core concept underpinning the code review studies in this dissertation.

QA Quality Assurance — see *Software Quality Assurance* (SQA).

RepoInsights RepoInsights — the Software Quality Assurance Analytics dashboard prototype developed and evaluated in Chapter 4; it aggregates testing, automated workflow, pull request, and guideline data into a global overview and provides a comparative aspect by benchmarking projects against a curated set of popular open-source repositories.

SA Software Analytics — a research and practice area that enables software practitioners to explore data to obtain insightful and actionable information for data-driven decision-making.

Situational Awareness Situational Awareness — borrowed from psychology (Endsley, 1995), the concept refers to the perception of environmental elements and the projection of their status in the near future; in this dissertation it is applied to the SQA domain to describe developers' understanding of the quality assurance mechanisms active in their own projects.

SQA Software Quality Assurance — the set of systematic activities and mechanisms that software engineers employ to ensure software meets its specified requirements and maintains reliability; the practices studied in this dissertation include software testing, modern code review, automated static analysis, and CI/CD automation.

SQA Awareness Software Quality Assurance Awareness — defined in this dissertation as the knowledge software engineers possess regarding the specific QA practices active in their projects, including what tools are configured, how they function, when they execute, and how to interpret their signals; investigated empirically in Chapter 3.

SQAA Software Quality Assurance Analytics — the application of software analytics to SQA data, aggregating information from testing frameworks, code review discussions, and automation logs into a holistic view to enhance situational awareness and support data-driven decision-making; the concept explored through the REPOINSIGHTS prototype in Chapter 4.

Test Coverage Test Coverage — a measure of the extent to which source code is exercised by a test suite; commonly reported as line/statement coverage or branch coverage; used in Chapter 2 as a proxy for testing intensity.

Workflow Run Workflow Run — an individual execution record of a GHA workflow, including metadata such as timestamp, trigger event, status (success, failure, canceled), the associated commit or pull request, and the branch on which it ran; in Chapter 6.

Workflow Smell Workflow Smell — a suboptimal pattern or practice in a GHA workflow configuration that can negatively affect the maintainability, performance, security, or reliability of the workflow, analogous to code smells in source code; seven confirmed workflow smells are identified and validated in Chapter 5, spanning security, performance/optimization, and general CI/CD categories.

YAML YAML Ain't Markup Language — a human-readable data-serialization format used by GitHub Actions for workflow configuration files (stored as `.yml` or `.yaml` files in the `.github/workflows/` directory); the static quality and evolution of these files are analyzed in Chapter 5.

Curriculum Vitæ

SeyedAli Khatami

1996/06/01	Born in Tehran, Iran

Education

2021 – Present	**Ph.D. in Software Engineering** Delft University of Technology, the Netherlands
2018 – 2021	**M.Sc. in Software Engineering** Sharif University of Technology, Iran
2014 – 2017	**B.Sc. in Software Engineering** Islamic Azad University, South Tehran Branch, Iran

Experience

2025 – Present	**Data Engineer / Researcher** Data Refinery Lab, TU Delft, the Netherlands
2021 – 2025	**Ph.D. Candidate** Delft University of Technology, the Netherlands
2018 – 2021	**Software Engineer** Intelligent Information Solutions Center (IIS), Iran
2014 – 2017	**Junior Software Engineer** ITDPRose, Iran

Services

2024	**MSc Thesis Supervision**, TU Delft Supervised Cédric Willekens on his thesis "Catching Smells in the Act: A GitHub Actions Workflow Investigation"
2022	**Co-Founder & Board Member**, Delft Iranian Students Association (D.I.S.A.) 'Alborz', Delft, the Netherlands
2022	**Student Volunteer**, 15th IEEE International Conference on Software Testing, Verification and Validation (ICST 2022)
2020	**Head of Teaching Assistants**, Fundamentals of Programming in Python, Sharif University of Technology

List of Publications

1. **Ali Khatami** and Andy Zaidman: State-of-the-Practice in Quality Assurance in Java-Based Open Source Software Development. *Software: Practice and Experience*, 2024.

2. **Ali Khatami** and Andy Zaidman: Quality Assurance Awareness in Open Source Software Projects on GitHub. In *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2023.

3. **Ali Khatami**, Carolin Brandt, and Andy Zaidman: Software Quality Assurance Analytics: Enabling Software Engineers to Reflect on QA Practices. In *Proceedings of the 24th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2024.

4. **Ali Khatami**, Cédric Willekens, and Andy Zaidman: Catching Smells in the Act: A GitHub Actions Workflow Investigation. In *Proceedings of the 24th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2024.

5. **Ali Khatami**, Carolin Brandt, and Andy Zaidman: Beyond the YAML File: Understanding Real-World GitHub Actions Workflow Adoption. In Proceedings of the 30th International Conference on Evaluation and Assessment in Software Engineering (EASE), 2026.

6. Baris Ardic, Carolin Brandt, **Ali Khatami**, Mark Swillus, and Andy Zaidman: The Qualitative Factor in Software Testing: A Systematic Mapping Study of Qualitative Methods. *Journal of Systems and Software*, vol. 227, 2025.

7. Carolin Brandt, **Ali Khatami**, Mairieli Wessel, and Andy Zaidman: Shaken, Not Stirred: How Developers Like Their Amplified Tests. *IEEE Transactions on Software Engineering*, vol. 50, no. 5, pp. 1264–1280, 2024.

Included in this thesis.

Titles in the IPA Dissertation Series since 2023

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis.* Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond.* Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic.* Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06

J. Jacobs. *Guarantees by construction.* Faculty of Science, Mathematics and Computer Science, RU. 2024-07

O. Bunte. *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software.* Faculty of Mathematics and Computer Science, TU/e. 2024-08

R.J.A. Erkens. *Automaton-based Techniques for Optimized Term Rewriting.* Faculty of Mathematics and Computer Science, TU/e. 2024-09

J.J.M. Martens. *The Complexity of Bisimilarity by Partition Refinement.* Faculty of Mathematics and Computer Science, TU/e. 2024-10

L.J. Edixhoven. *Expressive Specification and Verification of Choreographies.* Faculty of Science, OU. 2024-11

J.W.N. Paulus. *On the Expressivity of Typed Concurrent Calculi.* Faculty of Science and Engineering, RUG. 2024-12

J. Denkers. *Domain-Specific Languages for Digital Printing Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13

L.H. Applis. *Tool-Driven Quality Assurance for Functional Programming and Machine Learning.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-14

P. Karkhanis. *Driving the Future: Facilitating C-ITS Service Deployment for Connected and Smart Roadways.* Faculty of Mathematics and Computer Science, TU/e. 2024-15

N.W. Cassee. *Sentiment in Software Engineering.* Faculty of Mathematics and Computer Science, TU/e. 2024-16

H. van Antwerpen. *Declarative Name Binding for Type System Specifications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-01

I.N. Mulder. *Proof Automation for Fine-Grained Concurrent Separation Logic.* Faculty of Science, Mathematics and Computer Science, RU. 2025-02

T.S. Badings. *Robust Verification of Stochastic Systems: Guarantees in the Presence of Uncertainty.* Faculty of Science, Mathematics and Computer Science, RU. 2025-03

A.M. Mir. *Machine Learning-assisted Software Analysis.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-04

L.T. Vinkhuijzen. *Data Structures for Quantum Circuit Verification and How To Compare Them.* Faculty of Mathematics and Natural Sciences, UL. 2025-05

D. van der Wal. *What is the Point? Single-Input-Change Testing a EULYNX Controller.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-06

A. Rosset. *Uniform Monad Presentations and Graph Quasitoposes.* Faculty of Sciences, Department of Computer Science, VU. 2025-07

L. Guo. *Higher-Order Termination with Logical Constraints.* Faculty of Science, Mathematics and Computer Science, RU. 2025-08

D.A. Manrique Negrin. *A Model Orchestra in Digital Twins: A Model-Driven Approach to Integration and Orchestration.* Faculty of Mathematics and Computer Science, TU/e. 2025-09

C.A. Esterhuyse. *Specification-Centric Multi-Agent Systems.* Faculty of Science, UvA. 2025-10

H.M. Muctadir. *Consistency Matters: Building Consistent Digital Twin Virtual Entities.* Faculty of Mathematics and Computer Science, TU/e. 2025-11

A. Stramaglia. *Model Checking Machine-Control Applications.* Faculty of Mathematics and Computer Science, TU/e. 2025-12

M. Saeedi Nikoo. *Supporting business process management: clone detection and recommendation techniques.* Faculty of Mathematics and Computer Science, TU/e. 2025-13

R.B. Rubbens. *Bridging the Implementation Gap: Advances in Model-Based Concurrent Program Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-14

F.I. van der Berg. *DMC Model Checker: Delta-Driven Variable-Length Next-State Generation via Recursive Compression.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2025-15

D.G. Sprokholt. *Correct Translation between Weak Memory Model Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2025-16

B. Kohlen. *Verified & Fast Probabilistic Verification: Stepwise Refinement of Probabilistic Model Checking into an IEEE 754 Floating-Point Implementation.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-01

D. Frolich. *Exploratory Language Development.* Faculty of Science, UvA. 2026-02

L.B. van den Haak. *Verifying Optimised Parallel Code.* Faculty of Mathematics and Computer Science, TU/e. 2026-03

A. Papotti. *Human Acceptance of Machine Generated Code Security Analysis.* Faculty of Sciences, Department of Computer Science, VU. 2026-04

N. Doorn. *O bug, where art thou? Improving software testing education.* Faculty of Science, OU. 2026-05

C.R. van der Rest. *Reusable Programming Language Components.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-06

T.T.P. Franken. *A Parallel Programming Perspective – Designing AuDaLa and AuDaLogic.* Faculty of Mathematics and Computer Science, TU/e. 2026-07

T.J. Smeding. *Reverse-mode Automatic Differentiation for Functional Array Languages.* Faculty of Science, UU. 2026-08

S.A. Rukmono. *Towards Automated Software Architecture Explanation: An AI-enabled method and industrial evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2026-09

R.T.C. Turkenburg. *Comparing and Constructing (Co)Inductive Predicates.* Faculty of Science, Mathematics and Computer Science, RU. 2026-10

P.B.H. Tasche. *Deductive Verification Techniques for Embedded Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2026-11

W.B. Mbaka. *An Empirical Investigation Into Human Judgments of Security Threats.* Faculty of Sciences, Department of Computer Science, VU. 2026-12

S. Srinivasan. *Taming Uncertainty, Enabling Predictability: Systematic Timing Analysis of Real-Time Systems Across Computation and Communication.* Faculty of Mathematics and Computer Science, TU/e. 2026-13

S.O. Brand. *Symbolic Algorithms for the Analysis of Quantitative Systems.* Faculty of Mathematics and Natural Sciences, UL. 2026-14

D. Engel. *Understanding Binaries, Correctly: Decidability and Formal Verification in Binary Analysis.* Faculty of Science, OU. 2026-15

K. Hagens. *Rewriting Induction for Higher-Order Constrained Rewriting.* Faculty of Science, Mathematics and Computer Science, RU. 2026-16

S. Khatami. *Understanding Software Quality Assurance in Open-Source Communities: Awareness, Adoption, and Tool Uti-*

lization. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2026-17

TODAY'S SOFTWARE DEVELOPMENT ECOSYSTEM OFFERS AN UNPRECEDENTED ARRAY OF QUALITY ASSURANCE MECHANISMS. CONTINUOUS INTEGRATION (CI) PLATFORMS, AUTOMATED TESTING FRAMEWORKS, AUTOMATED STATIC ANALYSIS TOOLS, AND CODE REVIEW SYSTEMS ARE NOW READILY AVAILABLE TO DEVELOPERS, PARTICULARLY IN OPEN-SOURCE SOFTWARE (OSS) PROJECTS HOSTED ON PLATFORMS LIKE GITHUB. YET THIS ABUNDANCE DOES NOT AUTOMATICALLY TRANSLATE INTO EFFECTIVE ADOPTION. INSTEAD, IT INTRODUCES A NEW CHALLENGE: THE RAPID EXPANSION OF SOFTWARE QUALITY ASSURANCE (SQA) PRACTICES AND TOOLS HAS CREATED A COMPLEX DECISION SPACE WHERE TEAMS MUST DETERMINE WHICH PRACTICES TO ADOPT, HOW TO CONFIGURE THEM EFFECTIVELY, HOW TO INTEGRATE THEM INTO DEVELOPMENT WORKFLOWS, AND HOW TO DESIGN PROCESSES THAT ARE RELEVANT FOR THEIR SPECIFIC PROJECT CONTEXT. THIS DISSERTATION AIMS TO CONSTRUCT A HOLISTIC UNDERSTANDING OF SOFTWARE QUALITY ASSURANCE BY BRINGING TOGETHER DIFFERENT ASPECTS OF OPEN-SOURCE SQA PRACTICES THAT CAN INFORM FUTURE DECISION-MAKING.

ISBN 978-94-6518-299-5

9 789465 182995