Test Suite Optimization for Human-in-the-loop Testing Processes in Industry: Addressing Slow Test Feedback and Risks from Untested Changes

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

by

Roman Haas

Saarbrücken, 2025

UNIVERSITÄT
DES
SAARLANDES

Roman Haas: *Test Suite Optimization for Human-in-the-loop Testing Processes in Industry: Addressing Slow Test Feedback and Risks from Untested Changes,* © September 2025

Abstract

Background Software underpins the digital infrastructure that sustains modern societies. The importance and complexity of software systems necessitate rigorous testing to ensure their quality and, in particular, their correctness. In industrial contexts, humans play a significant role in software testing. They plan and conduct manual tests, make critical decisions about test completion, and monitor for risks, for example, arising from untested code changes. As software systems become more complex, the challenges of testing increase, consuming more time amidst limited resources. At the same time, compiling a test plan and deciding when all risks have been mitigated are particularly demanding tasks for the humans involved.

Objective In this dissertation, we seek to optimize human-in-the-loop testing processes in industrial practice by enhancing their efficiency and effectiveness. We target two optimization levers: (1) adopting automated test optimization techniques to improve manual testing; (2) supporting test management and quality assurance in the labor-intensive task of allocating test effort and assessing test completion. To accomplish the former, we explore optimization opportunities in manual testing, in particular, established optimization techniques from automated testing. We strive to understand the prerequisites and limitations for their transferability to existing manual testing processes, and how effective these techniques can be. To realize the latter, we prioritize untested code changes according to estimated risk.

Methods and Results To achieve our objective, we have conducted a series of empirical studies on human-in-the-loop testing, using methods such as field experiments and sample studies with industry partners. Manual test suites offer great optimization opportunities, since they often suffer from long run times—up to five person-months for our industry partners. Based on historical data and stakeholder interviews with our industry partners, we demonstrate the transferability and effectiveness of optimization techniques from automated to manual testing. Our results show that applying test case selection and prioritization to manual testing captures up to 81% of failures while reducing execution time by 43%.

The second optimization lever addresses the labor-intensive code and test reviews which our industry partners conduct to mitigate the risks of untested code changes. We explore risk factors for code changes and propose a simple risk-based prioritization approach for untested code changes. In our evaluation using historical quality assurance documents from our industry partners, this approach was able to prioritize risky changes significantly higher than less risky changes. Our studies have demonstrated the suitability and effectiveness of the proposed solutions in practice, and after our studies, many subjects have been convinced to adopt our solutions by embedding them in their testing process.

Conclusion We demonstrate a variety of optimization opportunities and levers for human-in-the-loop software testing. Our empirical studies provide evidence of the feasibility and effectiveness of our optimization techniques in industry contexts. This dissertation constitutes a solid foundation for future research on human-in-the-loop testing processes and facilitates adoption by practitioners through detailed optimization guidelines. Both are crucial contributions to the future of software engineering.

Zusammenfassung

Hintergrund Software bildet das Fundament der digitalen Infrastruktur, auf der moderne Gesellschaften aufbauen. Um die Softwarequalität und in erster Linie die Korrektheit dieser bedeutenden und komplexen Softwaresysteme sicherzustellen, sind gründliche Tests unerlässlich. Im industriellen Kontext spielen Menschen im Softwaretesten eine wichtige Rolle. Sie planen und führen manuelle Tests durch, sie treffen kritische Entscheidungen über den Testabschluss und sie überwachen Risiken, die beispielsweise durch ungetestete Codeänderungen entstehen. Mit der zunehmenden Komplexität von Softwaresystemen wachsen auch die Herausforderungen beim Testen. Das Testen wird dadurch zeitaufwendiger, während die verfügbaren Ressourcen begrenzt bleiben. Besonders anspruchsvolle Aufgaben für die beteiligten Personen sind das Erstellen eines Testplans und die Entscheidung, wann das Testen abgeschlossen ist.

Ziele In dieser Dissertation zielen wir darauf ab, menschengestützte Testprozesse aus der Industrie zu optimieren, indem wir die Effizienz und Effektivität des Testens steigern. Wir konzentrieren uns auf zwei Hebel für die Optimierung: (1) die Übertragung von Testoptimierungstechniken für automatisierte Tests auf das manuelle Testen; (2) die Unterstützung des Testmanagements und der Qualitätssicherung bei der arbeitsintensiven Steuerung des Testaufwands und der Bewertung, ob das Testen abgeschlossen werden kann. Um Ersteres zu erreichen, untersuchen wir Optimierungsmöglichkeiten für das manuelle Testen, insbesondere etablierte Optimierungstechniken aus dem automatisierten Testen. Wir versuchen zu verstehen, unter welchen Voraussetzungen und mit welchen Einschränkungen sie auf bestehende manuelle Testprozesse übertragen werden und wie effektiv diese Techniken dabei sein können. Für Zweiteres setzen wir auf eine risikobasierte Priorisierung ungetesteter Codeänderungen.

Methodik und Ergebnisse Um unser Ziel zu erreichen, haben wir eine Reihe empirischer Studien zu menschengestützten Testprozessen durchgeführt. Dabei kamen Methoden wie Feldexperimente und Fallstudien mit Industriepartnern zum Einsatz. Manuelle Testsuiten bieten vielfältige Optimierungsmöglichkeiten, denn sie leiden häufig unter langen Laufzeiten – bei unseren Industriepartnern dauert die Ausführung einer manuellen Testsuite bis zu fünf Personenmonate. Auf Grundlage historischer Daten und Stakeholder-Befragungen mit unseren Industriepartnern zeigen wir, dass Optimierungstechniken wirksam vom automatisierten auf das manuelle Testen übertragen werden können. Unsere Ergebnisse demonstrieren, dass die Auswahl und Priorisierung von Testfällen bis zu 81 % der Fehler bei manuellen Tests erfasst und gleichzeitig die Ausführungszeit um 43 % reduziert.

Der zweite Hebel für die Optimierung adressiert arbeitsaufwendige Code- und Testreviews, die unsere Industriepartner durchführen, um das Risiko ungetesteter Codeänderungen zu mindern. Hierfür untersuchen wir Risikofaktoren für Codeänderungen und schlagen auf dieser Grundlage einen einfachen risikobasierten Priorisierungsansatz für nicht getestete Codeänderungen vor. Bei der Evaluierung anhand historischer Qualitätssicherungsdokumente unserer Industriepartner war unser Ansatz in der Lage, risikoreiche Änderungen signifikant höher zu priorisieren als weniger risikoreiche. Unsere Studien haben die Praxistauglichkeit und Wirksamkeit der vorgeschlagenen Lösungen bestätigt. Viele der Beteiligten waren nach unseren Studien so überzeugt von unseren Lösungen, dass sie diese in ihren Testprozess integriert haben.

Schlussfolgerung Wir demonstrieren eine Vielzahl von Optimierungsmöglichkeiten und dort ansetzende Hebel für menschengestützte Testprozesse. Unsere empirischen Studien belegen die Anwendbarkeit und Effektivität unserer Optimierungstechniken im Industriekontext. Diese Dissertation bildet eine solide Grundlage für zukünftige Forschung im Bereich menschengestützter Testprozesse und erleichtert Praktikern die Implementierung durch detaillierte Optimierungsrichtlinien. Beide Aspekte liefern einen entscheidenden Beitrag zur Zukunft des Software Engineerings.

Acknowledgments

I would like to express my deepest gratitude to those who have supported and guided me throughout my PhD journey.

First and foremost, my profound thanks go to my supervisor, Sven Apel. I am deeply grateful for the opportunity to pursue an external dissertation under his guidance. His prompt and precise direction throughout my PhD journey has been invaluable, providing critical feedback on my research ideas and paper drafts. He taught me to recognize opportunities in criticism and rejections, which allowed me to outgrow myself.

I also extend my sincere gratitude to my external reviewers, Daniel Méndez Fernández and Michael Felderer, for their commitment to serving on my dissertation examination board and undertaking this occasionally tedious duty.

I owe a special thanks to my co-authors for their collaboration and support. I thank Daniel Elsner for our hands-on workshops that inspired our fruitful collaboration. I am thankful to Raphael Nömmer for his perseverance when conducting our joint field experiment. I am grateful to Michael Sailer for his ongoing engagement in his original Master's Thesis topic. I thank Mitchell Joblin for sharing his wealth of experience in methodology and statistics.

My colleagues at CQSE have been instrumental in my journey. I am profoundly grateful to Elmar Juergens for instilling in me the ambition to undertake a PhD. I greatly value his innovative ideas for solving real-world problems, and his insights into successful research. Special thanks go to Andreas Göb and Jakob Rott who offered vital support during the ABAP implementation of test impact analysis. I extend my thanks to my colleagues for their constructive feedback, engaging discussions on research topics, and participation in my studies.

I had the pleasure to work on a joint research project with my colleagues from the Technical University of Munich. I am grateful to Fabian Leinen and Roland Würsching for enlightening discussions and extensive feedback on my work. My heartfelt appreciation goes to my colleagues from the University of Saarland who integrated me—an external PhD candidate—into the research group as if I had been an internal.

The close cooperation with industry partners allowed me to understand many problems addressed in this thesis better and to evaluate my solutions on real-word data with experts from various domains. My sincere thanks go to Munich Re, Zeiss, ILP, and LV 1871 for their invaluable support. I am grateful for the PhD program by CQSE and the funding from the German Federal Ministry of Education and Research (BMBF), without which this research endeavor would not have been possible.

Most of all, my deepest gratitude goes to my family which played an essential role in allowing me to focus on the most relevant topics during my working and research hours, enabling progress in my PhD while ensuring family time. My wife, Melanie, has been an unwavering pillar of support throughout the entire journey. As a proud father of two wonderful daughters, I love to see them grow.

Contents

List of Figures

List of Tables

Acronyms

ACF	Added critical findings
ANF	Added normal findings
CEN	Code centrality
CHF	Changed functions
CLI	Changed lines of code
COC	Complexity change
COM	Complexity of reference function
LEN	Length of reference function
RCF	Removed critical findings
RNF	Removed normal findings
UCF	Unresolved critical findings
UNF	Unresolved normal findings

Publications

Parts of this dissertation and its contributions build on previous publications by the author. We delineate the publications presented in this thesis and divide personal and co-authored contributions from multi-authored publications.

Contributions from Multiauthor Publications

The Chapters 3, 4, and 5 are based on the following publications where the author of this dissertation is first author. All of the publications listed below have multiple authors, and in some cases the first authorship is shared between two authors [53, 56]. For the sake of transparency, we separate the personal contributions of the author of this dissertation from the contributions of co-authors, following the Contributor Roles Taxonomy (CRediT)[1].

[53] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel. "How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies." In: *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021, pp. 1281–1291

The 2021 research by Haas et al. [53] was conducted by the author of this dissertation and Daniel Elsner. Both contributed equally to the research, that is, conceptualization, methodology, data curation, investigation, validation, visualization, writing of the original draft, and, in particular, to the content findings. The case study presented in Section 3.3 was conducted separately for the two study subjects, with the author of this thesis being responsible for study subject 1.

[56] R. Haas, R. Nömmer, E. Juergens, and S. Apel. "Optimization of Automated and Manual Software Tests in Industrial Practice: A Survey and Historical Analysis." In: *IEEE Transactions on Software Engineering* 50.8 (2024), pp. 2005–2020

For the 2024 article by Haas et al. [56], the author of this thesis and Raphael Nömmer share first authorship. Both contributed equally to this work, that is, its conceptualization, methodology, data curation, investigation, validation, visualization, writing of the original draft, and, in particular, to its content findings. While the authors collaborated closely on this research effort, especially on the field experiment and the optimization guidelines, the following contributions can be attributed individually: Pareto testing was contributed by Raphael Nömmer. The author of this dissertation contributed the survey and discussion of optimization-relevant differences between automated and manual testing processes.

1 CRediT by NISO, see also https://credit.niso.org/

[57] R. Haas, M. Sailer, M. Joblin, E. Juergens, and S. Apel. “Prioritizing Test Gaps by Risk in Industrial Practice: An Automated Approach and Multimethod Study.” In: *IEEE Transactions on Software Engineering* 51.5 (2025), pp. 1554–1568

The author of this work is first author of the 2025 article by Haas et al. [57] and contributed core parts of the topic conceptualization and methodology as part of his supervising role in Michael Sailer’s Master’s Thesis [139]. Michael Sailer is the second author of the work by Haas et al. [57]; he originally developed the approach presented in Section 5.2 and implemented the tool TestGapRadar. The author of this dissertation conducted the multimethod study outlined in Section 5.3 and wrote the original draft of the article manuscript.

1

Introduction

This chapter shares material with prior publications [53, 56, 57].

Software is nearly ubiquitous in our daily lives, underpinning the very infrastructure that sustains modern societies. From the seamless operation of global communication networks to mobility and advancements in healthcare, software serves as the invisible, yet indispensable force driving innovation and functionality. This infrastructure is run by large software systems for which the requirements evolve over time. Their complexity and change frequency are two of manifold reasons why these systems are prone to errors; making software testing crucial to ensure expected functionality. Testing ensures high software quality, including its correctness, by uncovering potential defects.

As the complexity of software systems increases, the challenge of adequately testing them escalates. In industrial practice, limited resources necessitate optimizing the testing process to maximize efficiency and effectiveness. This involves strategically selecting, prioritizing, and executing an as small as possible set of tests to achieve maximum test coverage, fault detection, and risk mitigation while minimizing time, cost, and effort. The ultimate goal is to ensure a lightweight testing process without compromising software quality, allowing timely identification of defects to maximize user satisfaction on production environments and reduce developer effort.

In this thesis, we focus on two optimization levers of industrial software testing where humans are in the loop: First, testing is sometimes conducted by humans. In fact, manual testing is a de-facto industry standard to test software and is very resource-intensive, offering vast potential for optimization. Still, from the literature on test optimization, it remains mostly unclear how to leverage this potential. While there is considerable research on optimization techniques for automated testing, their applicability in manual testing remains unclear. Having shown to be effective for automated testing, it appears promising to apply their core ideas in manual testing. To address this issue, we investigate the transferability of optimization techniques from automated testing to manual testing.

Second, it is often humans, such as test managers, who allocate test efforts and assess test completion. Our industrial partners focus their testing efforts on untested code changes—also known as test gaps—because they are known to be more likely to contain defects, and their test management completes testing as soon as all relevant changes are tested. Unfortunately, the defect-proneness of changes varies, making it harder for test management to decide which gaps are relevant and need to be closed. Furthermore, potential defects are not the only risk factor that needs to be considered, but their impact is also relevant. That is, potential defects within business critical software components could result in tremendous costs, which is why they need to be tested more thoroughly. To support test management and quality assurance roles in identifying relevant test gaps, we propose a risk-based prioritization of test gaps that considers likelihood and impact of potential defects.

1.1 Optimization of Industrial Manual Software Testing Processes

Software testing can be conducted automatically or manually. It is almost a rule that software test suites grow with their systems under test [169]; consequently, large software systems have large test suites [128]. Regardless of whether testing is automated or manual, these extensive test suites take considerable time to execute. Beyond the execution cost, lengthy run times hinder early and meaningful feedback to developers [32, 67, 168]. For instance, our industry research partners' test suites (see Section 4.2.2) take, on average, a week for automated tests to run, 5 person-months for manual tests; they struggle with late feedback and insufficient resources to complete tests promptly. Manual testing is especially tedious, costly, and involves significant human effort, yet it remains widely used in industry [7].

Despite advanced test automation techniques, manual software testing often complements automated testing in practice [36, 91]. Manual testing strategies can arguably detect faults that automated methods might miss [19]. Depending on the project's context, automation can be prohibitively costly [163], complex [157], or even infeasible [152], making manual testing indispensable in the foreseeable future. The increasing number of test cases and frequency of execution, driven by shorter release cycles, exacerbates the issue of long-running test suites hindering the software development process [67, 168].

Significant research has focused on optimizing automated testing, for example, regression test optimization [29–31, 46, 61, 62, 90, 96, 127], but few efforts have aimed to apply techniques such as regression test case selection [28, 111], prioritization [66, 87], or failure prediction [67] to manual software testing. Several barriers, including the lack of data (e. g., code coverage information [66]), impede this transfer. Unlike automated testing, manual testing processes are not always integrated with version control, continuous integration systems, test frameworks, or build tools. Furthermore, it remains unclear whether and to what extent humans in the loop limit the applicability of test optimization techniques for automated testing in manual contexts. In general, manual tests often slow down development, making their optimization even more crucial [66, 67]. Additionally, varied test processes, environments, and test suite run times dictate how ambitiously optimization techniques must be applied to deliver fast feedback while maintaining high fault detection rates.

1.2 Prioritization of Test Gaps in Industrial Software Testing Processes

As said previously, resources in software development are limited, particularly for large software systems. Therefore, it is critical to allocate test efforts to detect the most serious defects as early as possible. This requires estimating which parts of the software are particularly prone to defects. The area of defect prediction seeks to identify faulty code, for example, through static program analysis, possibly enhanced by heuristic search or machine learning [70, 79, 92, 98, 172]. Despite numerous studies, results often lack generalizability [59], and current approaches frequently perform poorly in real-world applications [121, 122]. So it does not surprise that they are rarely applied in practice [92, 122, 160], with notable exceptions, though [151].

Addressing the notorious issues of defect prediction of our industry partners (in particular, Munich Re and LV 1871), we strive for an approach that is viable in practice and aligns with their needs: *prioritizing test gaps* by their *risk*. A *test gap* is any method, function, or module that has been modified during a specific timeframe (e. g., start of last development phase or iteration), yet has not been executed in its most recent version during testing (e. g., automated unit test or manual acceptance test). Intuitively, defects are introduced by code changes, and defects cannot be detected if they were not tested. In this vein, the literature suggests that modified code tends to be more defect-prone [27, 84, 115, 140]. Eder et al. found that, despite structured testing processes, about half of all changes go untested into production, with untested changes harboring up to five times more defects than others [27]. This underscores the importance of test gap analysis, which test management uses to allocate testing efforts and assess test completion, for example, for a release.

The number of test gaps requiring investigation by test management and quality assurance hinges on numerous factors, especially the number and size of code changes and the testing scope (i. e., the number and diversity of test cases). Typically, dozens, hundreds, or even thousands of test gaps need to be investigated [77]. In Section 2.2.4.1, we show an example for a test gap analysis result from our industrial partner Munich Re yielding thousands of test gaps, each varying in risk. Risky test gaps such as logic modifications or data manipulations may be obscured by less critical changes, for example, refactorings [134]. In the context of this thesis, the risk of a test gap refers to the probability of the test gap being defect-prone and the magnitude of impact of a potential defect. Assessing test gaps and their risks demands considerable effort and may yield subjective results, and it can be challenging in large systems with evolving teams. Clearly, an *automatic prioritization of test gaps* would be most helpful to streamline manual inspections and reduce the risk of overlooking critical test gaps, which is also supported by our industrial partners.

1.3 Thesis Goal

This thesis aims to enhance the feedback provided by human-in-the-loop testing processes in industrial contexts by increasing their efficiency and effectiveness, thereby reducing defects and improving software quality for users. To achieve this, we explore in this thesis two essential levers for optimization of industrial testing where humans are involved: inherently resource-intensive manual testing and assessing test completion without overlooking risky untested changes. Both fields offer open research questions we aim to answer within this thesis, and adopting both measures promises the highest efficiency and effectiveness gains for human-in-the-loop testing processes.

Automated and manual testing are similar in many ways: they pursue the same goal, in that they ensure the quality of a system under test, they involve running test cases to identify defects, and automated and manual test suites both may suffer from too large and long-running test suites, necessitating optimization. Given the many effective optimization techniques available for automated testing, the question arises under what circumstances can these techniques be effectively transferred to manual testing? Existing research on manual testing optimization often targets specific techniques in narrow contexts. Existing results are often not generalizable due to the wide diversity of manual testing processes. We need a better understanding of manual testing processes to derive the benefits and limitations of applying optimization techniques from automated testing. It remains unclear which automated techniques are suitable for existing manual processes: What data can be leveraged, and how? In the same vein, practitioners need clarity on *what* techniques are applicable and *how* to incorporate them into their *existing* processes and infrastructures. Additionally, there is a lack of evidence of the extent to which test optimization techniques for automated testing can be adapted to manual testing in industrial practice, and what limitations need to be accepted. We aim for extensive guidance for manual testing practitioners in choosing optimization techniques, making prerequisites, potential benefits and limitations of specific approaches transparent, and providing evidence on the effectiveness of manual test optimization from practice.

The aim regarding effort allocation and completion assessment in testing is to optimize the process so that risky changes are tested as early as possible. This contributes to our major thesis goal in two ways: First, allocating resources to mitigate the risks from these changes increases testing effectiveness. Second, testing efficiency increases when testing is finished once all relevant test gaps have been addressed. The decision of when to complete testing is much easier when test gaps are sorted by estimated risk. We are seeking an alternative solution to defect prediction because the costs of state-of-the-art approaches outweigh the potential benefits for our industry partners. Moreover, state-of-the-art approaches typically do not consider prior testing efforts that are focussed by test gap analysis [70, 151, 172], which our industry partners rely on for the purposes of test effort allocation and test completion assessment. To support test managers in scanning large numbers of test gaps with varying risk, we aim for an automatic risk-based prioritization of test gaps. As there is only little related work on prioritization or risk estimation of test gaps in industrial practice, it is still unclear what makes a test gap more risky than others. Therefore, we seek to evaluate the feasibility of a simple prioritization approach that fulfills our industry partner's requirements and that is designed to be easily adaptable in other contexts. This dissertation strives to bridge the gap between the challenging task to identify defects from code modifications and the practical limitation of testing resources by an automated risk-based prioritization of test gaps.

1.4 Contributions

In this thesis, we make several contributions to enhance feedback from industrial human-in-the-loop testing processes to developers. Our major contributions include:

1. **Exploration of Optimization Technique Transferability from Automated to Manual Testing:** Our first contribution is an analysis of the transferability of optimization tech-

niques from automated testing to manual contexts. A survey of 38 testing professionals across 16 companies provides insights into their testing processes and identifies optimization opportunities. We identify nine optimization techniques applicable to manual testing and discuss the circumstances under which they can be implemented in practice. Our synthesis includes an annotated model of manual software testing processes and guidelines for practitioners on selecting suitable optimization techniques. This model allows practitioners to pinpoint optimization levers in their specific contexts and the guidelines facilitate the adoption of optimization techniques (Chapters 3 and 4).

2. **Evidence on Optimization Effectiveness for Manual Testing:** Our second contribution provides evidence on the effectiveness of optimization techniques from automated testing when applied in industrial manual testing processes. We conducted two industrial case studies on test suites from Munich Re and IVU Traffic Technologies, demonstrating improvements in fault detection probability, test feedback time and test creation efforts by following our guidelines (Chapter 3). To research on commonalities and differences in optimization, we implemented two optimization techniques for automated and manual testing processes. Our empirical study involved five subjects from, inter alia, Bayerische Versorgungskammer, Dolby, ILP, and Carl Zeiss Microscopy. Our results show that optimized automated test suites detect, on average, 80% of failures while saving 66% of execution time, compared to 81% failure detection and 43% time savings for manual tests. Despite inherent manual testing limitations, we provide evidence for the effectiveness of these techniques in industrial settings (Chapter 4).

3. **Risk-based Prioritization of Test Gaps:** Our third contribution is a risk-based prioritization approach for test gaps, evaluated using a multimethod study. This approach targets test management and quality assurance roles that (1) allocate test efforts or (2) assess test-end criteria. Our prioritization considers the magnitude of impact and defect probability, measured by code criticality (e. g., code centrality) and complexity metrics/static code analysis results. Through a multimethod study involving 31 historical quality assurance reports from eight industrial software systems of Munich Re and LV 1871, and semi-structured interviews with six quality engineers that authored the reports, we were able to validate the risk criteria's transferability. Our automated approach exhibits a ranking performance equivalent to expert assessments. That is, test gaps labelled as risky in historical quality assurance reports are prioritized significantly higher than less risky ones at the 30th percentile, on average. This study shows that a lightweight prioritization method like ours is suitable to efficiently highlight high-risk test gaps while filtering low-risk ones (Chapter 5).

In general, our proposed solutions are designed for high adoptability. For example, we implemented our analyses in a language-agnostic way to maximize impact. Employing empirical research methods, including field and sample studies [150], we investigate optimization potential in extensive industrial human-in-the-loop testing processes across domains such as finance, audio, and optics. Our studies draw on historical static and dynamic testing data, results from static code analysis, quality assurance data, and semi-structured interviews with testers, test managers, and quality assurance experts. This thesis provides a solid foundation to identify optimization potentials in extensive and long-running testing processes in industry, which often conduct manual testing, and need to mitigate risks from untested changes in an as cost-effective way as possible. Our contributions offer help to practitioners to shape lightweight human-in-the-loop testing processes without compromising software quality, focussing on a timely identification of defects.

1.5 Outline

This thesis is structured as follows: Chapter 2 provides necessary background on software testing, interfaces, and optimization terminology. Chapter 3 examines manual testing processes in industrial settings through a developer survey and evaluates the transferability of automated test optimization, presenting optimization guidelines for practitioners. It includes case studies from two industrial software projects, discussing benefits and limitations. Chapter 4 explores the optimization of automated and manual software tests in industrial practice through a multimethod study, revealing insights on their commonalities and differences. It includes a field experiment implementing two test optimization techniques in five industrial study subjects, with a historical analysis. Chapter 5 targets the automated prioritization of test gaps, introducing a method to prioritize them by risk and evaluating it in a multimethod study involving real-world quality assurance data from eight industrial subjects. Chapter 6 summarizes findings and suggests future research directions based on our contributions. Appendix A contains links to supplementary Web sites with additional material, facilitating reproduction of the empirical studies.

2 Background

This chapter shares material with prior publications [53, 56, 57].

This chapter provides an introduction into the foundations that this dissertation is built upon. At the beginning, we give an overview of basic software engineering terminology, including software testing, which is used throughout this work. This is followed by an extensive overview of the research area on software test optimization and its research fields.

2.1 Basic Concepts for Software Testing

We introduce software engineering terminology which we use to discuss the optimization of human-in-the-loop software testing processes in industrial contexts. Initially, we define terms related to software testing processes to establish a common foundation for testing process optimization. Subsequently, we examine concepts from the interface between software testing and development which are crucial for the optimization discussions in this work.

2.1.1 Software Testing Process

Testing terminology can vary based on context. In this thesis, we primarily adhere to the standard terminology of ISO/IEC/IEEE 29119-1 [1]. In this section, we delve into this terminology, enriching our discussion with examples and contextual insights from our research.

2.1.1.1 *Goals of Software Testing*

Software testing aims to uncover potential quality issues within a software system [44]. Following ISO/IEC/IEEE 29119-1, we formally define it as follows:

Definition "Software testing"
Software testing is a verification process that ensures high quality of software systems.

Correctness, a primary quality attribute, is often addressed by identifying faults, commonly referred to as *bugs* or *defects* (see also Sec. 2.1.1.3). Testing also positively influences software's accessibility, maintainability, reliability, scalability, and usability. Throughout this thesis, the term *system under test* refers to the software being evaluated. The term *test* *item* may relate either to an entire software system being tested or a subset of such a system, for example, single software components or classes.

Testing supports two processes: *verification* and *validation*. Verification focuses on the test item's conformance to specifications, such as requirements or change requests. Validation, in contrast, assesses the test item's acceptability in meeting stakeholder needs. Simply put, verification focuses on building the software product right and validation deals with building the right product [10].

In the context of this thesis, testing primarily serves two *purposes*—defect detection and information gathering—as per the standard [1]. Detecting defects allows for their resolution, thereby enhancing software quality, such as correctness. Additionally, the information gathered supports decision-making in the test process, for example, (test) managers can make information-based decisions on when to end testing.

2.1.1.2 *Testing Forms, Testing Levels, and Test Activities*

Testing can be conducted in various ways, at different abstraction levels, and aiming for different objectives. We first distinguish between static and dynamic testing, as well as manual and automated testing. We then present an overview of test levels that implement verification across abstraction layers and discuss several testing objectives pursued by test activities.

ISO/IEC/IEEE 29119-1 differentiates between two forms of testing: *static* and *dynamic*. Static testing refers to activities such as reviews, model verification, and static analysis, which do not require execution. Conversely, dynamic testing involves executing code through test cases. This thesis focuses on dynamic testing.

We use the terms *manual testing* and *automated testing* to differentiate between software verification processes where test cases are either "run manually by a human test executor, or [are] executed by a test automation tool" as per the IEEE standard [1]. In this thesis, manual tests typically occur at higher abstraction levels, such as system tests, while automated tests range from unit to system levels. To collect testing processes for which substantial activities are carried out by humans, we define:

Definition "Human-in-the-loop testing process"
A *human-in-the-loop testing process* contains at least one activity from test planning and execution which is conducted by a human.

Examples for human-in-the-loop testing processes include manual testing or those which involve humans for activities, such as, the allocation of testing efforts or the assessment of test end criteria. Automated testing involves humans in designing test cases, maintaining the test suite, and debugging, that is the identification of root-cause(s) of a test failure (see also Sec. 2.1.1.3), but are not considered human-in-the-loop testing processes in the context of this thesis.

Exhaustive testing involves dynamic testing to prove that a specific system under test meets all requirements under all given circumstances, that is, all possible input values in all possible states. The standard points out that exhaustive testing is "impractical" and in practice "not possible". Instead, *sampling* is employed to derive test suites from the vast set of possible input values and states. The standard notes, "[c]hoosing the subset of possible tests that are most likely to uncover issues of interest is one of the most demanding tasks of a tester". [1] This thesis focuses on comprehensive test processes, often involving large test suites, which are too large to provide quick and actionable feedback. To address this, we outline optimization levers in testing processes, test execution, and test suite maintenance (see also Chapter 3).

In general, software testing can be conducted on different abstraction layers, or *test levels*, see also Table 2.1. The most specific level, *unit testing*, verifies a single or a set of related software units. Depending on the context of the system under test, a unit typically refers to a method, class, or package. *Integration testing* refers to testing a set of units, especially to verify their interfaces. The verification of the entire system under test is called *system testing*, or end-to-end testing. Since software systems are often integrated into ecosystems of systems or are required to run on various devices, networks and surrounding systems, *interoperability testing* aims to uncover risks on this high level of abstraction.

Table 2.1: Test levels and their objectives with typical stage in development process

Test Level	**Objective**	**Stage in Development Process**
Unit testing	Verify that individual test items, e. g., single classes or methods, function correctly.	Development
Integration testing	Ensure that combined components of the system under test interact correctly and interface properly.	After unit testing
System testing	Verify the complete integrated system under test against specified requirements.	After integration testing
Interoperability testing	Ensure that the system under test can interact and functions correctly with other systems.	After system testing

The ISO/IEC/IEEE standard requires a *test plan* which "describes the objectives of the testing, and the activities to be performed to achieve those objectives" [1]. The selection of *activities* (also called test type in ISO/IEC/IEEE 29119-1) is based on the risks associated with the objectives, which are largely concerned with meeting product and project requirements. Various test activities exist, and their suitability depends on specific objectives and stages of the development process, see also Table 2.2. *Regression testing*, aimed at revealing failures in unmodified system parts, is typically conducted early in development. *Smoke testing* helps to control build resources in a continuous integration environment since resource- and time-intense build steps are only run when a small test set did not reveal any failures. *Performance*, *Security*, and *Usability testing* refer to different quality attributes and are usually run after system testing. *Exploratory testing* refers to intuitive and undocumented testing by developers and testers, and can already be done during development but is often conducted separately after system testing. To ensure that a large set of changes can be released (or deployed), *user acceptance tests* can be defined in advance and completed before building the release of the system under test. In Chapters 3 and 4, we explore which test activities are employed in industrial automated and manual testing processes.

Table 2.2: Test activities and their objectives with typical stage in development process

Test Activity	**Objective**	**Stage in Development Process**
Regression testing	Verify the absence of failures in unmodified parts in the system under test.	Development
Smoke testing	Perform preliminary tests to ensure the basic functionalities of a system under test are working correctly.	Early in testing phases
Performance testing	Assess the system under test's responsiveness, stability, scalability, and robustness under load and stress.	After system testing
Security testing	Identify security defects (vulnerabilities) in the system under test to ensure data and resource protection.	After system testing
Usability testing	Manually assess the system under test's user interface and user experience for intuitiveness and ease of use.	After system testing
Exploratory testing	Identify defects through unscripted, intuitive human testing.	Throughout testing phases
Acceptance testing	Validate that the software meets business requirements and is ready for release through documented human testing.	End of development; pre-deployment

2.1.1.3 *Test Cases, Test Cycles, and Defects*

Throughout this thesis, we describe optimization techniques for heterogenous testing processes that target various optimization levers. This section introduces key testing artifact terminology crucial for detailed discussion. These techniques operate at either a test case or test-suite level, defined as follows:

Definition "Test case"
A *test case* t consists of a set of preconditions P_t, inputs I_t, and expected results E_t.

For automated test cases, the inputs I_t are function calls, while for manual test cases, they are test steps documented in natural language. A test case is denoted as t_p if it passed in the most recent run or t_f if it failed. The *fault revelation capability* $P_{\text{fr}}(t)$ of a test case t represents its probability to fail in the presence of a fault $f \in F$ in the system under test, where F is the (obviously unknown) set of all faults in the system under test. The fault revelation function $R(t)$ returns the set of faults revealed by t: $R(t) = \{f \in F \mid t \text{ fails because of } f\}$.

Definition "Test suite"
A collection of test cases $\{t_1, \dots, t_n\}$ forms a *test suite* $T = \{t_1, \dots, t_n\}$.

Similar to individual test cases, $P_{\text{fr}}(T)$ denotes the fault revelation capability of T, indicating the probability that any test case within T will fail in the presence of a fault: $\exists t \in T : t_f$.

Automated and manual testing processes have a different nature. To investigate the transferability of optimization techniques from automated to manual testing, we must capture

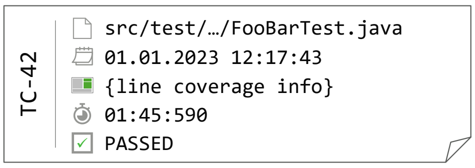


Figure 2.1: Example of a test report; test reports consist of test identifier, source path, execution timestamp, line coverage information, duration, and the result

diverse test execution processes, such as verifying one software version via a continuous integration pipeline (see also Sec. 2.1.2) or conducting a full test phase during manual release testing. For this purpose, we introduce the term test cycle:

Definition "Test cycle"
A *test cycle* refers to the execution of test cases with the same test goal.

The test optimization techniques considered in this work require data about each test case, which is collected from test reports generated during the execution of test suites. Figure 2.1 shows a stylized test report comprising an identifier, source path, execution timestamp, line coverage information, duration, and the result (e. g., *pass* or *failure*). Using Pretschner's terminology [132], we define the following terms:

Definition "Failure", "Error", "Fault"
A *failure* describes the deviation between the expected result and the actual behavior of the system under test. An *error* is the incorrect internal state of the system under test that leads to a failure. A *fault* is semantically incorrect code that may result in a failure.

The three terms—failure, error, and fault—are collectively referred to as *defects*.

2.1.2 Interfaces between Software Testing and Development

The system under test is central to the testing process and, consequently, plays a pivotal role for test optimization. This section introduces concepts from software development and static code analysis, which are key for optimization techniques discussed later.

2.1.2.1 *Software Development Workflow with Code Collaboration Platforms*

In modern software development, a *version control system* is essential for managing the code history including all changes and facilitating collaboration among developers. At the heart of a version control system is the *repository*, a storage location that maintains source- and non-source files along with their complete revision history. The most prominent version control system is git[1], which is why, we follow the terminology from this version control system. Each time a developer makes a set of changes to the codebase, they create a *commit*, which is a snapshot of those changes, stored with a unique identifier (also called *hash*), author information, and a timestamp. Commits are organized into *branches*, which allow developers to work on different features or bug fixes independently without affecting the *main line of development*. When a branch is ready to be integrated back into the main line, a *merge* operation

1 https://git-scm.com

is performed to combine the changesets, resolving any conflicts that may arise. Once local changes are committed and finalized, they can be pushed from the local *clone* to the *remote repository*, making them available for verification in the continuous integration pipeline and accessible for the rest of the development team.

Continuous integration enhances the development workflow by automatically building and testing code changes as they are integrated into the remote repository. A continuous integration *pipeline* consists of automated *jobs* that run during each integration event, for example, each push to a feature branch. These jobs can include tasks such as compiling code and running automated tests—which produces a test report for each test execution. To obtain *continuous deployment*, tasks can also deploy built applications to staging environments. Information from the version control system and continuous integration provide valuable information for test optimization, for example, the change history of a system under test or historical test reports.

Nowadays, version control systems, combined with continuous integration and deployment, form the backbone of *code collaboration platforms*. These platforms further enhance the development process. For instance, *pull requests* may require a reviewer's approval before merging branch changes into the main line. Pull requests also gather additional data on branch changes, including build and test results, reviewer comments, and annotations from static code analysis tools.

Figure 2.2 illustrates a code collaboration platform setup hosting a version control system (`Remote Repository`) and continuously integrating changes through build and test pipelines. Two developers clone the remote repository. Developer 1 (`Dev 1`) works on `Feature B`, while Developer 2 (`Dev 2`) works on `Feature C`. They persist local changes in their local repository (`working copy`) using the `git commit` command and share them with the remote repository via `git push`. Synchronization from the remote repository to the local working copy occurs through the `git pull` command. In this example, continuous integration is configured for the most recent commits on both main and feature branches. Thus, if `Dev 1` pushes changes from his local copy of `Feature B` to the remote repository, a pipeline automatically runs, building and testing the application. The `test` task stores test reports for future test optimization. Upon completing `Feature B`, `Dev 1` submits a pull request, which `Dev 2` might approve to merge the changes from `Feature B` into the `main` branch.

In test optimization, version control systems and code collaboration platforms serve as essential data sources. Many optimization techniques depend on historical data from these systems. For instance, Chapters 3 and 4 explore optimization techniques for both automated and manual testing processes that rely on source history and test execution records. The prioritization of test gaps introduced in Chapter 5 is strongly dependent on versioning information and historic coverage information, as test gap analysis requires source code and execution history for accurate test gap calculation.

2.1.2.2 *Static Code Analysis*

Static code analysis tools evaluate source code without execution, functioning as static testing tools (see also Sec. 2.1.1). The goals of these tools range from identifying simple bug patterns to conducting sophisticated security analyses [109]. Developers benefit from static code analysis through immediate feedback on code changes, improving code quality [155] and re-

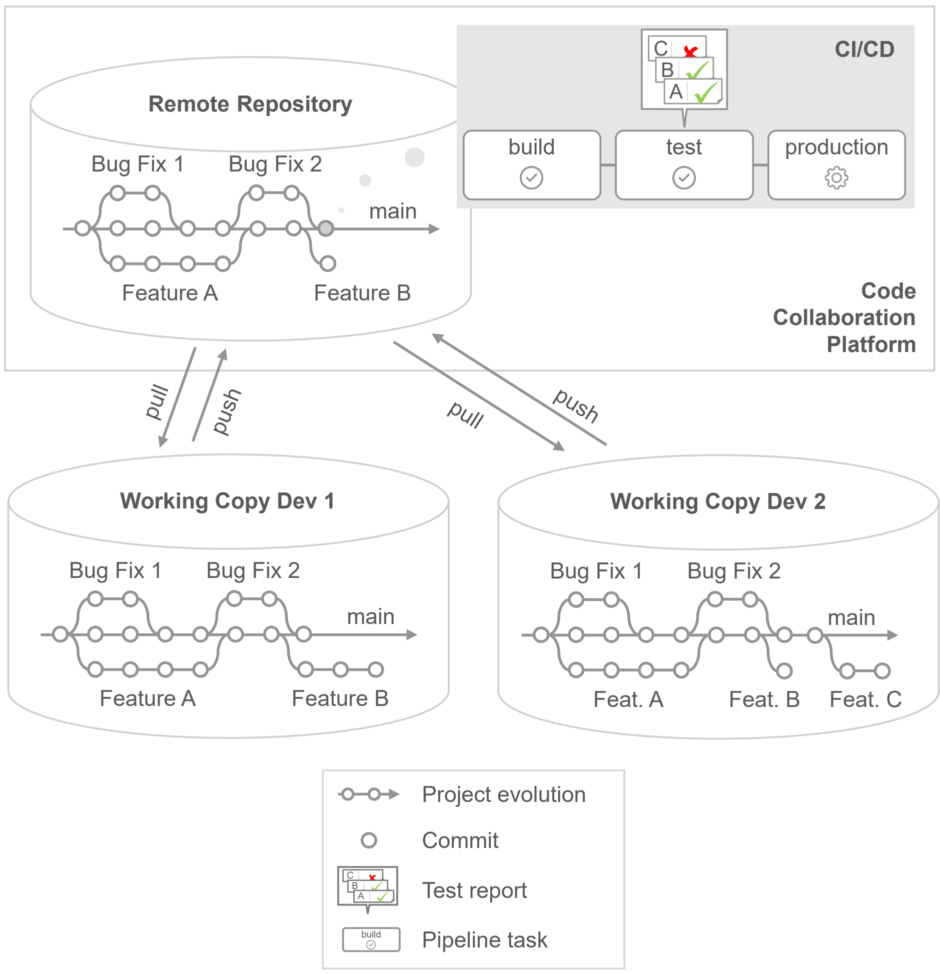


Figure 2.2: Exemplary development setup with code collaboration platform, including continuous integration and continuous delivery (CI/CD), and two local repository clones for Developer 1 (`Dev 1`) and Developer 2 (`Dev 2`)

ducing bugs [58]. We refer to static analysis results as *findings* (also known as *static analysis warnings*), which are classified by criticality—either *critical* or *normal*—based on severity and urgency.

Static analysis facilitates the measurement of quality indicators [155], such as the code size, either measured in *lines of code* (LOC) (including comments and blank lines) or *source lines of code* (SLOC) (excluding code comments and blank lines). Since software systems evolve, the evolution of quality is of interest, too. For instance, the *findings churn* summarizes the number of findings *added*, *unfixed in modified code*, and *resolved* across a set of commits.

Static analysis can also be employed to identify central code (e. g., classes or features) of a software system [55, 147, 149], which are arguably more important to test. To achieve this, a graph representation known as a *dependency graph* $G = (V, E)$ represents the source code, with $V = \{v_1, \cdots, v_n\}$ as the set of vertices (such as classes or implementation files). The edges $e_{i,j} = (v_i, v_j) \in E$ signify dependencies between vertices v_i and v_j, like *import* or *using* dependencies. Various methods, including the PageRank algorithm [116], can identify the most *central* vertices within G.

We utilize findings and code centrality measures as risk indicators to prioritize test gaps (see also Chapter 5).

2.2 Optimization Techniques for Software Testing

The central goal of this thesis is to explore methods for reducing time-to-feedback from testing environments in industrial development contexts. To this end, the broad research field of test optimization tackles manyfold levers to reduce time to feedback. Before delving into specific optimization approaches in Chapters 3, 4, and 5, we provide an overview of test optimization techniques categorized by their targets: single (i. e., current or next) test cycle, test cases, and entire test suites. By summarizing techniques aimed at defect risks, we discuss how testing effectiveness can be enhanced. This section frames the bigger picture of test optimization before discussing differences between automated and manual testing and their impact on optimization techniques.

2.2.1 Optimization Techniques for a Single Test Cycle

We explicate optimization techniques for a single test cycle. Typically, the results of these optimizations are re-evaluated for every cycle, such as a test task in the continuous integration pipeline or a manual testing phase. The most prominent optimization techniques for single test cycles are test case prioritization and test case selection, which we discuss first. We then introduce test impact analysis, which combines the principles of these two techniques.

2.2.1.1 *Test Case Prioritization*

Test case prioritization aims at ordering a test suite $T = \{t_1, \ldots, t_n\}$ to optimize the execution order of test cases with respect to a specific optimization goal. In this thesis, test case prioritization aims at optimizing the time to feedback by executing fault-revealing tests t_i as early as possible. Technically, test case prioritization approaches strive for an optimally ordered set $O_{tcp} \subseteq T$ where all failing tests are executed before all passing tests:

$$O_{tcp} = \left\{ t_i \,|\, \forall t_i, t_j \in T, 1 \leq i,j \leq n \;:\; t_i \text{ fails} \wedge t_j \text{ passes} \Rightarrow i < j \right\} \tag{2.1}$$

As test results may be unpredictable, heuristics have been devised to predict failing tests and to derive prioritization approaches from these [83]. Typically, heuristics approximate the fault revelation capability $P_{\text{fr}}(t)$ of the test cases. Prioritization becomes especially essential in manual testing, where strict resource constraints (such as time and work force) necessitate early fault detection.

2.2.1.2 *Test Case Selection*

Test case selection strives for choosing a subset T_{fr} from a test suite T with the goal of saving testing efforts. It involves gathering test cases deemed execution-worthy based on a heuristic. While various criteria, such as high code coverage or testing critical code segments, could define the worthiness of a test case, we concentrate our selection process on tests anticipated to potentially yield failures:

$$\forall t \in T_{\text{fr}} \subseteq T \;:\; P_{\text{fr}}(t) > 0 \tag{2.2}$$

This approach is *unsafe*, potentially missing some failing tests. In large and complex software systems, safe selection techniques are often infeasible or ineffective [31, 34, 127]. So, state-of-the-art selection techniques are unsafe [33], for example, because of dynamic dependencies [90], language boundaries [16], or non-code changes [112]. There has been substantial research on test case selection [81, 171]. Usually, an approximation of the single test case fault revelation capability is used to identify potential failures. In practice, the approximations are often based on source code changes [81]. Intuitively, a test case that covers changed code might reveal new faults that have not been found by previous test runs. A previously passing test case that runs through unmodified code is expected not to change its behavior. However, this is not always true, as seen with flaky tests (see also Sec. 2.2.5).

2.2.1.3 *Test Impact Analysis*

Test impact analysis combines test case selection and prioritization to quickly identify test failures based on code changes within a specific timespan $[t_{\text{base}}, t_{\text{end}}]$. It aggregates the code changes from the last release or sprint and produces an ordered set of test cases which—according to a heuristic—maximize the fault revelation capability P_{fr}. Formally, test impact analysis calculates an ordered set $O_{tia} \subseteq T$ such that

$$O_{tia} = \left\{ t_i \,|\, \forall t_i, t_j \in T, 1 \leq i, j \leq n \;:\; P_{\text{fr}}(t_i) > P_{\text{fr}}(t_j) \Rightarrow i < j \right\} \tag{2.3}$$

In the literature, there are several approaches that combine a test case selection and prioritization strategy. For example, Greca et al. [49] propose a hybrid test optimization approach, which combines the tools *Ekstazi* for test case selection [46] and *FAST* for test case prioritization [103]. A systematic literature review summarizes the field of machine-learning-based test case selection and prioritization [118]. We evaluate a variant of test impact analysis in automated and manual testing contexts from industry in Chapter 4 (see Section 4.1.2.1 for variant details).

2.2.2 Optimization Techniques for Test Cases

In the previous section, we discussed optimization techniques for a single test cycle. Here, we focus on techniques that optimize individual test cases, independent of test cycles: test case reduction, test case refactoring, and test quality monitoring.

2.2.2.1 *Test Case Reduction*

Smaller test cases promise many benefits, such as, faster execution times, improved understandability, and enhanced debugability [170]. Like test suite minimization (discussed in Section 2.2.3.1), test case reduction aims to decrease the size of a test case while maintaining its fault detection capability [3]. Formally, it minimizes the test inputs $i \in I_t$ of a test case t and identifies a reduced test case t_{red} such that

$$\forall i \in I_t \; : \; i \in I_{t_{\text{red}}} \subseteq I_t \implies \nexists f \in F \; : \; f \in R(t) \; \land \; f \notin R(t_{\text{red}}) \tag{2.4}$$

where $R(t)$ is the fault revelation function of t (see also Sec. 2.1.1.3). Depending on the testing process, that is, automated or manual testing, test case reduction works differently: For automated test cases, statements such as function calls are removed. For manual test cases, test steps are eliminated.

2.2.2.2 *Test Case Refactoring*

Test case refactoring focuses on improving the internal structure and quality of test cases without modifying their functionality [8]. Its goal is to make test cases more readable, maintainable, and reusable. Formally, test case refactoring transforms test cases t into t_{ref} such that

$$t_{\text{ref}} = \left(P, \Delta_I\left(I_t\right), \Delta_E\left(E_t\right)\right) \tag{2.5}$$

where Δ_I and Δ_E are behavior-preserving modification functions for inputs and expected results. Since the behavior of the refactored test case t_{ref} is intended to remain unchanged, $R(t) = R(t_{\text{ref}})$. Refactoring examples include reusing test steps to eliminate redundancy or parameterizing test cases to avoid duplication.

2.2.2.3 *Test Quality Monitoring*

Test quality monitoring is an activity within the development process [53] to identify *test smells* in manual test cases [65]. Similar to smells in automated tests [23, 120] smells in manual tests refer to quality deficits that adversely affect the comprehensibility and maintainability of the test suite. Soares et al. [145] collect the following test smells for manual test case descriptions:

- ambiguous test case
- conditional test case
- eager action
- misplaced action
- misplaced precondition
- misplaced verification

As a result of test quality monitoring, test smells in input values I_t and expected results E_t are resolved by applying cleanup functions $\sigma_I(I_t)$ and $\sigma_E(E_t)$, resulting in a deodorized test case t_{deo}:

$$t_{\text{deo}} = \left(P, \sigma_I(I_t), \sigma_E(E_t)\right) \tag{2.6}$$

The cleanup functions σ may alter the behavior of test cases, that is, the deodorized test cases t_{deo} are not necessarily refactored versions of the original test cases t. In fact, this property of σ is required to address some tests smells, for instance, the ambiguity of test cases.

2.2.3 Optimization Techniques for Test Suites

Beyond optimizing test cycles (see Sec. 2.2.1) and individual test cases (see Sec. 2.2.2), optimization levers arise from the relationships among test cases, such as redundant test cases or those with an optimal cost-benefit relationship. Here, we discuss two optimization techniques based on these relationships: test suite minimization and Pareto testing.

2.2.3.1 *Test Suite Minimization*

Test suite minimization aims to identify a minimal test suite T_{min} which minimizes the test suite size of the original test suite T, while maintaining its fault detection effectiveness [168]. Thus, the fault revelation capability of both test suites T and T_{min} are identical: $P_{\text{fr}}(T) = P_{\text{fr}}(T_{\text{min}})$, if a safe minimization approach is used. For this purpose, redundant test cases or similar test cases that are unlikely to detect different faults are eliminated from the test suite. Formally,

$$\forall f \in F : \exists t \in T \land f \in R(t) \implies \nexists t' \in T_{\text{min}} \subseteq T \land f \in R(t') \tag{2.7}$$

where $R(t)$ refers to the fault revelation function (see Section 2.1.1.3).

Test suite reduction [133] is closely related with test suite minimization, with the terms often used interchangeably in literature. However, there is a subtle difference in their goals: test suite size reduction versus minimization. In their systematic review on test suite reduction, Rehman Khan et al. [133] discuss four approaches: greedy, search, clustering, and hybrid. Large-language models can also be used to minimize test suites [119].

2.2.3.2 *Pareto Testing*

Pareto testing is deduced from the Pareto principle, which often suggests an 80/20 rule: for instance, 20% of code might contain 80% of faults [131]. Similarly, a small percentage of time (less than 20%) can detect a substantial portion of failures (more than 80%) [26]. Formally, Pareto testing calculates an ordered subset T_p of a test suite T containing n test cases that runs within a specified cost limit of L. Each test case t has a cost function $C(t)$, representing execution costs, such as time or human effort. Initially, test case prioritization is applied (see Sec. 2.2.1.1), then the maximal number k of test cases fitting within the cost limit of L is determined:

$$k = \underset{1 \leq i \leq n}{\arg\max} \left(\left(\sum_{1 \leq l \leq i} C(t_l) \right) \leq L \right) \tag{2.8}$$

We evaluate a variant of Pareto testing in Chapter 4 and introduce a specific instance in Section 4.1.2.2.

2.2.4 Reducing Risk of Defects

This next set of optimization technique targets a different objective than earlier techniques. Test gap analysis, test gap prioritization, and defect prediction aim to enhance testing effectiveness by optimizing defect identification.

2.2.4.1 *Test Gap Analysis*

Test gap analysis is one possible answer to the question of test resource allocation. It reveals untested changes in the code of the system under test, which are known to be particularly defect-prone and should receive special attention in the testing process. The entity of interest for test gap analysis is often code at the *function or method* level (see Sec. 5.1.2).

Figure 2.3 illustrates a typical test gap analysis result for code changes within a multi-month release cycle of a large software system of our partner Munich Re (before they aligned their testing activities along untested code changes). The entire code base visualized as a tree map highlights test gaps in yellow and red; in this instance, there are *thousands* of test gaps. The sheer volume of code changes and testing activities makes identifying the most critical test gaps challenging, which is needed to direct the limited testing resources to the mitigation of the largest risks.

Test gap analysis is conducted over a designated timeframe $[b, t]$, defined by a baseline b and an endpoint t. It identifies test gaps, which we define as:

Definition "Test gap"
A *test gap* is a function or method which has been changed within a timeframe $[b, t]$ and its most recent version within $[b, t]$ has not been tested.

A function is deemed changed if at least one line of code is modified within $[b, t]$. This excludes behavior-preserving changes, such as variable renaming, and entities too trivial to test, such as simple variable access functions (getter and setter) [113]. A function is considered as tested if, following the most recent change in $[b, t]$, at least one line of the function has been executed by a test.

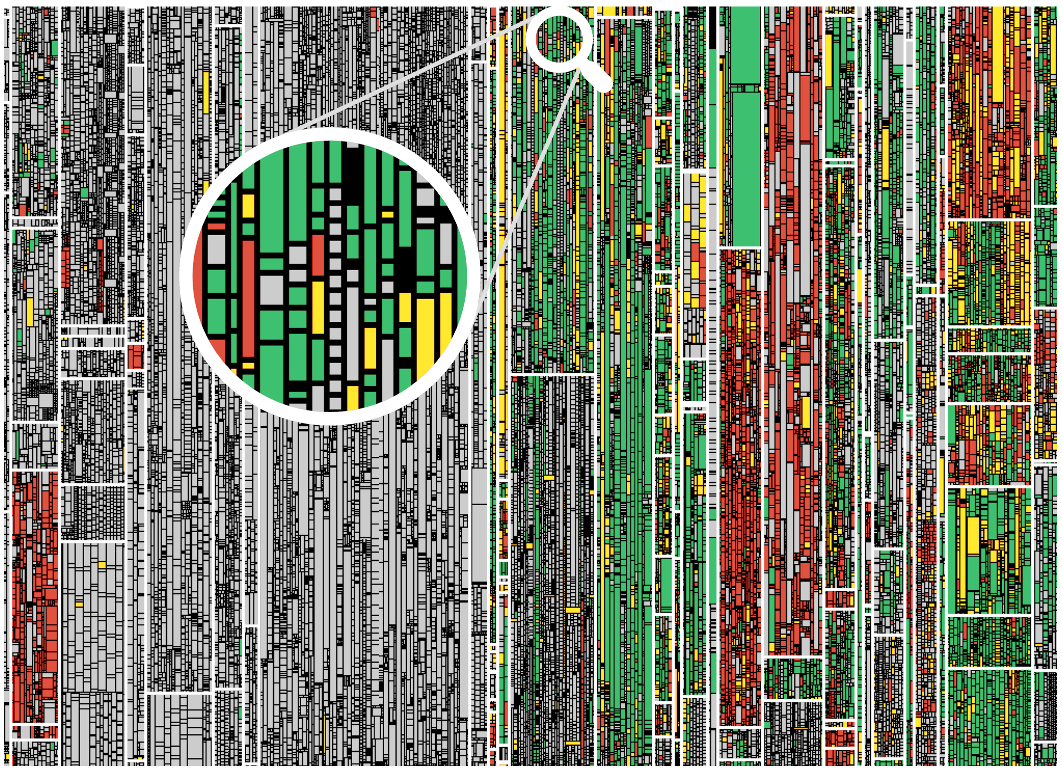


Figure 2.3: Test gap analysis results for an industrial business information system of our partner Munich Re for all code changes within a release cycle of several months. The tree map visualizes the hierarchical components, classes, and functions of a software system. The size of rectangles corresponds to size in lines of code; the color indicates ⬜ unchanged code, 🟩 tested changes, 🟨 untested modifications, and 🟥 untested new functions, where the latter two are test gaps. In this example, there are thousands of test gaps.

Formally, test gap analysis calculates a set $G_{[b,t]}$ consisting of all methods m_i added or modified within $[b,t]$ that have not been executed in test since the last change:

$$\begin{aligned} G_{[b,t]} = \Big\{ m_i \mid \exists\, \text{lastMod}(m_i, b, t) \wedge \\ \nexists\, \text{lastExec}(m_i, b, t) \,:\, b \leq \text{lastMod}(m_i, b, t) \leq \text{lastExec}(m_i, b, t) \leq t \Big\} \end{aligned} \tag{2.9}$$

where $\text{lastMod}(m_i, b, t)$ returns the timestamp of the last modification of a method m_i within $[b,t]$, and $\text{lastExec}(m_i, b, t)$ returns the timestamp of the last execution of m_i during testing within that period.

Experience in practice has shown that this simple notion of function coverage is capable of providing a valuable overview of untested changes [77]. Stricter test criteria, which take into account how thoroughly a function was tested, do not provide sufficient benefits to outweigh their cost [77]. They may even be counter-productive as they produce more complicated results, which require more effort to interpret. Note that the idea behind test gap analysis does not claim that unchanged and tested parts of the system do not contain any errors, but that the chances are higher to detect defects in untested changes than in other parts of the system.

2.2.4.2 *Test Gap Prioritization*

Test gap prioritization determines which test gaps $m_i \in G = \{m_1, \dots, m_n\}$ should be addressed first, because they pose the largest risks. Technically, once the risk r_{m_i} of a test gap m_i is established, G can be ordered by test gap risk:

$$\forall m_i, m_j \in G, 1 \leq i,j \leq n \,:\, r_{m_i} > r_{m_j} \Rightarrow i < j \tag{2.10}$$

The most prominent risk of test gaps arises from faults that could not have been detected because no test executed the faulty code. Other risk factors may also be considered; for instance, in Chapter 5, we discuss prioritizing test gaps by factoring in the centrality of untested code changes.

2.2.4.3 *Software Defect Prediction*

If we already knew where faults are buried in the code, testing to ensure correctness would diminish. This forms the basis of software defect prediction: forecasting which code entities $c_i \in C = \{c_1, \dots, c_n\}$ are most likely to contain a defect $f \in F$. To achieve this, a defect probability estimation $P_{\text{F}}(c_i)$ is employed to rank code entities:

$$\forall c_i, c_j \in C,\ 1 \leq i,j \leq n \,:\, P_{\text{F}}(c_i) > P_{\text{F}}(c_j) \Rightarrow i < j \tag{2.11}$$

A crucial decision in the evaluation of defect prediction techniques is the ground truth of defects. Pearson et al. [123] report that artificial faults are not sufficient to mimic real faults. Instead, they show that real faults need to be used to evaluate the performance of defect prediction research meaningfully.

Like with test gap analysis, it is an established best practice to use function-level defect prediction, which shows better performance than relying on coarse-grained units such as files or modules [40, 122]: Despite a great number of studies in this area, function-level defect prediction is still unsolved as it provides low precision for cross-project classifiers and, when evaluated under realistic circumstances, existing approaches do not significantly outperform a random classifier [121, 122]. As of now, our industrial partners need more actionable approaches such as test gap analysis and prioritization of test gaps to direct their testing efforts.

2.2.5 Bigger Picture of Test Optimization

In this chapter, we have explored several test optimization techniques relevant for the forthcoming chapters. To broaden the scope, we outline the bigger picture of test optimization by briefly introducing additional research areas and discussing their relevance to our work.

Mutation Testing Mutation testing is a technique used to evaluate the quality of software tests by automatically introducing small changes, so-called *mutations*, to the source code of the system under test [22, 60]. These mutations are designed to simulate common errors or mistakes developers might make, for example, an off-by-one error or usage of a wrong operator. The goal is to determine if the existing test suite can detect these changes, thereby assessing the suite's effectiveness. If the tests fail to catch the mutations, it indicates areas where the tests may be insufficiently rigorous or comprehensive. Our work on test gap prioritization (see also Chapter 5) is related to mutation testing since it also aims to identify gaps in testing. But the fault model of both techniques is different: mutation testing assumes that the mutations correspond to common programming mistakes of the system under test's developers, while the prioritization focuses on untested code changes because they are known to be more error-prone than unchanged code. Mutation testing is much more expensive and often considered infeasible in practice [175], with notable exceptions for tech-savvy organizations like Google [125, 126].

Detection of Flaky Tests A failing test is intended to send developers the clear signal of a mismatch between expected and actual system behavior, which they need to fix. Ideally, they either need to fix a fault—addressing the system under test behavior to match requirements—or the expected result of the test needs to be adjusted to reflect latest requirement changes which the system under test already complies with. Unfortunately, test cases may fail sporadically for reasons that lie outside of the system under test's source code. Such tests are called flaky tests, which are defined as:

Definition "Flaky test"
A test is *flaky* if it yields inconsistent results, that is, passing or failing unpredictably without changes to the code or test environment.

Flaky tests can be observed for all major languages, for example, C, Java, JavaScript, Python, Go [6, 50, 63]. The detection of flaky tests involves identifying and analyzing these tests to determine the root causes of their unreliability. Techniques to detect flaky tests improve testing reliability by ensuring that test failures indicate genuine defects rather than environmental issues or test brittleness. The detection of flaky tests is no test optimization technique per se, but it aims to save developer time by avoiding confirmation of flaky tests and tries to maintain software engineers' confidence in automated test results [41, 173]. Test optimization techniques that rely on historical test failures need to filter flaky tests—if there are any—to obtain meaningful results.

Fuzzing Fuzzing is an automated software testing technique that involves providing invalid, unexpected, or random data inputs to a program to discover security vulnerabilities and bugs [11, 93]. The purpose is to identify how the software behaves under stress and if it can manage incorrect inputs gracefully. Fuzzing is particularly useful for detecting vulnerabilities associated with buffer overflows, memory leaks, and crashes, thus improving the software's robustness and security profile [51]. So, fuzzing targets specific quality attributes which are beyond the scope of this work. In this thesis, we focus on enhancing effectiveness and efficiency of human-in-the-loop testing.

AI in Software Testing The integration of AI in software testing aims to enhance manyfold test concepts. For instance, there are studies on applying AI on different testing levels (unit, integration, and system testing), supporting various test activities (e. g., regression testing, mutation testing). AI can be applied in all steps of the testing process, for example, test case generation, test planning, test case design and specification. [4] AI is applied to many existing problems in the software engineering domain, and this also applies for software testing. In this thesis, we apply different methodology which can be applied more easily in industry contexts. For example, our methodology does not require to share business secrets like code or test data with foreign AI models and it requires less computing resources than common AI approaches.

2.2.6 Optimization of Automated vs. Manual Testing

Software testing processes exhibit significant variability, impacting the available optimization levers. One of the major differences in testing processes is whether testing is conducted automatically or manually. While exploring the differences between automated and manual testing is a central aspect of our work [53, 56] and is discussed in detail in Chapters 3 and 4 in this dissertation, here we outline key differences between the two methodologies and implications for the optimization techniques previously discussed.

The test execution mechanism fundamentally differs between automated and manual software testing. In automated testing, a test framework runs test cases within a predefined environment, such as a continuous integration pipeline or, with more degrees of freedom, an integrated development environment. Conversely, manual testing involves human testers inspecting the system under test in a potentially non-standardized test environment, such as a test stand or in a spontaneous exploratory test session. This complicates data collection: versioning information may be absent in manual test environments, coverage information retrieval requires significant effort, and mapping coverage information to test cases poses challenges because manual test cases often have implicit *begin* and *end* events. In Chapters 3 and 4, we explore the differences between automated and manual software testing processes within industrial contexts examining both organizational differences, such as, testing resources, and process-related differences, such as implemented test levels and activities.

While test optimization is a large research field that predominantly focuses on automated testing, the resource constraints of manual testing are even more compelling; manual test execution costs in terms of time and money are typically orders of magnitude higher (see also Chapter 4.3). Thus, optimizing manual testing processes is even more crucial than for long-running automated test suites. However, manual testing's inherent nature presents barriers to adopting the optimization techniques discussed in this chapter. These barriers often relate to the *human in the loop*, which can impact the quality of data required for optimization. Chapters 3 and 4 discuss the prerequisites, benefits, and limitations of various test optimization techniques. Most of them were initially proposed in the literature for automated testing, and adapted in our research for manual testing.

3 Manual Testing Process and Optimization Levers

This chapter shares material with a prior publication [53].

This thesis aims for optimization of industrial human-in-the-loop testing processes. In this chapter, we focus on manual testing processes, which are inherently costly and slow, necessitating their optimization. The literature discusses only few, highly tailored optimization approaches for manual testing. For another testing process, that is automated testing, a plethora of literature on various optimization techniques exists, but the general transferability to manual testing remains unclear. This chapter investigates the transferability of optimization techniques from automated testing to manual contexts and how to leverage their ideas and concepts in human-in-the-loop testing processes. To maximize impact, we develop guidelines for practitioners on selecting suitable optimization techniques, and share results of two industrial case studies demonstrating improvements in fault detection probability, test feedback time, and test creation efforts when following our guidelines.

Background Manual software testing is tedious, costly, and involves significant human effort. Yet, according to a survey, it is still widely applied in industry [7]. Despite the availability of advanced test automation techniques, previous research reports that manual software testing often complements automated testing [36, 91]. In fact, manual testing strategies can arguably detect other software faults than automated strategies [19]. Depending on the project's context, automation might be too costly [163], too complex [157], or even impossible [152], so there is no way around manual testing in the foreseeable future.

With an increasing number of test cases and execution frequency, due to shortening release cycles, long-running test suites impede the software development process [67, 168]. There has been significant research effort on optimizing automated testing, for example, on regression test optimization [29–31, 46, 61, 62, 90, 96, 127]. Still, only few research efforts attempt to transfer techniques such as regression test selection [28, 111], regression test case prioritization [66, 87], or failure prediction [67] to manual software testing. Transfer is hindered, among other things, by missing required data (e. g., unavailability of code coverage information [66]): In contrast to automated testing, manual testing processes are not necessarily integrated with version control or continuous integration systems, test (reporting) frameworks, and build or code instrumentation tools. In fact, it is often precisely manual tests that hamper the rapid development of systems, so optimizing them is even more important [66, 67].

Research Gap While the few existing studies on optimizing manual testing investigate the design and evaluation of specific techniques in specific contexts, it is unclear for which automated technique(s) an existing manual testing process is an eligible target: What data are available, easily producible, and can be leveraged in which ways? Consequently, to foster adoption of manual test optimization, practitioners need to understand what techniques are applicable and how to integrate them in their existing processes and infrastructure.

Solution To address this gap, we qualitatively analyze the prevalence, characteristics, and problems of manual testing activities and processes by surveying 38 test practitioners from 16 companies and different project contexts. The goal is to discover and systematize characteristics of manual testing that deviate from automated testing and that hinder or enable optimization of manual testing. We aim at deriving an actionable set of guidelines that empowers practitioners to quickly identify potential for optimization in their own context and reveal what researchers shall address. For this purpose, we investigate the transferability of optimization techniques from the literature and further derive techniques based on levers identified in our survey. We synthesize our findings as guidelines in the form of an *annotated model of manual software testing processes*, which highlights integration points for optimization techniques and summarizes associated prerequisites and caveats. By means of case studies on two industrial software projects from different domains we show that, using our guidelines, test feedback time and test suite maintainability can be improved.

Contributions Our contributions in this chapter are the following:

- *Developer Survey.* Evidence that manual testing is deliberately employed without the intention of full automation, underlining the need for optimizing manual testing. We provide quantitative and qualitative insights on how software is tested manually in practice.
- *Optimization Guidelines.* A set of guidelines rooted in an annotated process model and derived from our developer survey to implement nine optimization techniques for manual testing. We explain how to leverage existing processes and highlight integration points.
- *Industrial Case Studies.* Demonstration of the guidelines' usefulness in two industrial case studies. We pinpoint levers that can reduce test feedback time and test creation efforts.

3.1 Related Work on Optimizing Manual Testing

Studies from 2011 and 2013 on the state of software testing practice report that more than 90% of survey participants test their software manually [35, 45]. While participants of these studies see room for improvement with regard to their testing strategy (e. g., through better automation), they lack the resources to implement these. We have argued that optimization of manual software testing processes requires attention, as it addresses such scenarios where manual testing is inevitable [152, 157] or deliberately employed [36, 91].

Test optimization is widespread in automated testing [29–31, 46, 61, 62, 90, 96, 127], but techniques are often not transferable to manual testing due to missing data (e. g., code coverage information) [67] or unsuitable testing processes (e. g., only black-box access during testing) [66]. Despite these difficulties, several researchers have applied techniques to optimize different aspects of manual testing [8, 28, 66, 67, 76, 87, 111]. For example, Hemmati et al. [66] studied prioritization for manual regression tests on releases of Mozilla Firefox. In general, these techniques are often tightly coupled to specific testing processes or rely on specific data whose availability depends on the context.

We aim at guidelines for developers and testers that identify where existing optimization techniques can be used in practice based on their associated prerequisites. In addition, we pinpoint the challenges that arise in manual testing guiding further research in this area. Therefore, in what follows, we thoroughly review existing work and collect prerequisites and caveats for common optimization techniques, as shown in Table 3.1. The optimization techniques are later consolidated with findings from our empirical study in Section 3.2.4 to provide a holistic view on manual test optimization.

Table 3.1: Prerequisites and caveats of existing techniques to optimize manual testing

Ref.	Prerequisites	Caveats
	1. Test Case Prioritization	
[66]	Textual test descriptions, test failure history	Less effective in traditional development approaches
[87]	Textual test descriptions, test failure and execution history, expert labels to prioritize tests, test–requirement links	Labels and links are hard to obtain in retrospective and, if available, maintenance requires discipline
	2. Test Case Selection	
[76]	Test traces, familiarity of testers with code base	Under-specification of tests leads to unstable traces
[28]	Textual test descriptions, static code analysis tool, program profiler	Accuracy of static analysis low (90%)
[111]	Test traces, adjustment of system to separate traces for parallel testing	Unsuitable in the case of large or frequent changes
	3. Test Gap Analysis	
[15]	Test traces, version control data	Up-to-date test traces are costly, data granularity is critical
	4. Test Case Reduction	
[67]	Textual test descriptions, test failure history	Test cases need to have similar textual descriptions and there must not exist flaky tests to enable reduction
	5. Test Case Refactoring	
[8]	Textual test descriptions, individual test steps, expert labels for test suites	Varying effectiveness depending on the test suite
	6. Test Quality Monitoring	
[65]	Textual test descriptions	Parameterization requires experience

Test Case Prioritization Hemmati et al. [66] were the first to study regression test case prioritization (see also Sec. 2.2.1.1) for manual black-box system testing on releases of Mozilla Firefox. They found that in agile development environments, historical riskiness (i. e., how often test cases have detected faults before) is an effective surrogate for prioritizing tests when compared to approaches based on text mining test-case descriptions. Lachmann et al. [87] proposed to use machine learning to learn from test execution history (i. e., failures and execution time), requirements coverage, and test case descriptions to prioritize manual system tests. Their approach is more effective than random ordering, but requires labels, reflecting how important a test case is, which are obtained from test experts.

Test Case Selection Juergens et al. [76] report on an industrial case study that demonstrates challenges of applying test case selection (see also Sec. 2.2.1.2) to manual system tests based on method-level test traces. They suggest to use a semi-automated process, where testers could reduce the set of tests with domain knowledge. However, one caveat of this strategy is that testers need to know how to map code modifications to test cases, which, in general, is not the case. In addition, the common under-specification of manual tests leads to unstable test traces, which reduces the effectiveness of the technique. Eder et al. [28] propose an approach for regression test case selection that harnesses static analyses of the tested system's source code and manual system tests written in natural language to recover trace links between the two. Their evaluation, performed on one system and four test cases, indicates that their technique outperforms random selection of test cases, but even 90% correctly linked source code methods may be insufficient in practice. However, calibrating and evaluating the approach still involves a program profiler, which limits its transferability. In a case study on manual end-to-end testing of legacy Web applications, Nakagawa et al. [111] show that their simple test case selection approach based on method-level test traces yields test effort reductions compared to manual selection. However, it is not suitable in the presence of frequent or large code changes due to the performance penalty of dynamic analysis.

Test Gap Analysis Buchgeher et al. [15] describe a semi-automated approach for manual regression test case selection. Although their selection technique reveals deficiencies in effectiveness, it provides practical benefits for test gap analysis (see also Sec. 2.2.4.1), that is, finding modifications not covered by tests. Alongside, Buchgeher et al. state that selecting tests solely by code coverage leads to unnecessarily large sets of test cases, version control data is too coarse grained for their purposes, and keeping up-to-date coverage data for manual tests is costly.

Test Case Reduction Hemmati et al. [67] investigate text mining-techniques coupled with failure history-based analysis for failure prediction of system-level manual acceptance tests. Their technique can be used for test case reduction (see also Sec. 2.2.2.1), that is, for test suite maintenance by minimizing the test suite permanently, but also for selection and prioritization. Accordingly, their technique outperforms a naïve history-based model. It is the only work we found that explicitly states applicability for test case reduction, although such techniques are often overlapping with prioritization and selection [168].

Test Case Refactoring Bernard et al. [8] aim at improving tool support for refactoring manual test cases (see also Sec. 2.2.2.2) increase test suite maintainability, (e. g., through guided test suite minimization). For this purpose, they employ various text mining and machine learning algorithms on test steps in test case descriptions and report time reductions for refactoring and for execution of the refactored test suite. To apply the technique to a test suite, testers have to supply complexity estimates of the test suite and refactoring objectives; results vary depending on these objectives and the maturity of the test suite.

Test Quality Monitoring Hauptmann et al. [65] motivate test quality monitoring (see also Sec. 2.2.2.3) since they show that manual tests written in natural language often suffer from quality deficits, leading to decreased maintenance and comprehension. Their case study results show that their language models are able to detect test smells, yet require parameterization based on experience with the maintenance of natural language tests.

In summary, we are unaware of any previous work that investigates which optimization techniques (see Table 3.1) are applicable in practice, given an arbitrary existing manual testing process. Moreover, empirical studies on the state-of-practice in manual testing are relatively outdated with the most recent one being from 2013 [45].

3.2 Developer Survey and Guidelines

In this section, we provide details of our semi-structured online interview, following the suggestions of Jedlitschka et al. [73], and we derive a set of guidelines for optimization of manual testing processes that synthesize our findings.

3.2.1 Research Areas and Questions

With our interviews, we target several research questions from three research areas: We are interested in the reasons for the implementation of manual testing processes, outline their characteristics, and derive viable optimization techniques.

RA_1: Rationale behind Manual Testing

First, we need to understand what kind of manual testing processes are implemented in practice, why practice relies on this resource-intense way of testing, and what hinders test automation.

$RQ_{1.1}$: Why is software tested manually and what technological and organizational challenges hinder test automation? To be able to identify suitable optimization potential, we need to summarize why practitioners rely on manual testing. Additionally, there might be technological and organizational reasons for why test automation—as one of the more obvious optimization approaches—is not used.

$RQ_{1.2}$: Which testing activities are carried out manually in practice? There are many different kinds of testing which can be performed manually. We survey what testing activities (e. g.,

exploratory and regression testing) are carried out manually to be able to tailor optimization approaches to different needs.

RA_2: Characteristics of Manual Testing

Second, we investigate characteristics of manual software testing, how much effort is actually invested into manual testing, and which optimization techniques are already applied in practice.

$RQ_{2.1}$: How much effort is invested into manual software testing? This research question aims at determining the optimization potential with respect to testing accuracy and costs.

$RQ_{2.2}$: How does manual software testing integrate with the development process? We want to shed light on the interfaces and interdependencies of manual testing with the development process to uncover related optimization potentials.

$RQ_{2.3}$: How are test cases selected for execution and how are tests assigned to testers? Test case selection is a well-known optimization technique for automated tests, and we investigate in which circumstances it can be used in practice. The assignment of tests to testers needs to be understood because this reveals optimization constraints that might not be relevant for automated tests.

$RQ_{2.4}$: What are technical and organizational characteristics of (sub-) systems that are tested manually? We would like to understand patterns that encourage or hinder manual testing.

$RQ_{2.5}$: Do flaky tests exist in manual test suites and, if so, how do testers handle them? Flaky tests are a well-known problem for automated tests [94]. If flaky tests are also an issue for manual testing, an optimization goal would be to reduce the test flakiness, possibly with techniques different from automated testing.

RA_3: Optimization Techniques in Manual Testing

Finally, we aim at summarizing optimization techniques for manual testing.

$RQ_{3.1}$: Do manual test teams aim at test automation? How much time do they plan to invest? We investigate whether test practitioners strive for automation of their test suite and how much effort is expected and invested for it.

$RQ_{3.2}$: What potential for optimization of manual software testing exists and what are its prerequisites? This is our core research question. In Section 3.1, we summarize existing optimization techniques and their prerequisites. With this research question, we enrich our research-oriented perspective by collecting actively used optimization techniques reported by our participants. Following our previous discussion on the eligibility of optimization techniques, we also summarize associated prerequisites and caveats in practice.

3.2.2 Participants

In August 2020, we contacted 115 test engineers, testers, developers, test architects, test leads, and test managers of industry partners. We followed the strategy of closed invitations [159] to choose respondents based on their roles. $N = 38$ responded to our survey within two months. The response rate of 32.5% is relatively high, and we lead this back to our close partnership with our research partners. Most of our participants work in Germany, but there are also several participants from Canada (1), Italy (1), Romania (1), Switzerland (1), and the US (2). The participants work for organizations of different sizes, including medium-sized companies with a few dozen employees, as well as large organizations with tens of thousands employees. Their business domains include communication, network security, finance, health technology, public transportation, information technology, manufacturing, and hardware development.

3.2.3 Questionnaire and Conduct

We designed a questionnaire to address the above research questions. In Table 3.2, we list all survey questions, map them to our research questions, and mark open (🗎) and closed (☑) questions. Most of our survey questions were open so that the participants could explain their context. We used SoSci Survey to host our questionnaire and provided it in English and German (the native languages of most of our participants). All questions were optional.

3.2.4 Results

To analyze the answers of the survey, we used an open card-sorting technique [71]. To this end, we looked iteratively for higher-order patterns in the open answers of participants for each question. Overall, we spent 25 × 2 hours (per open question 🗎) = 50 hours on categorizing 633 answers.

We structure our discussion along our research areas and questions. For each research question, we present descriptive statistics of our closed survey questions (if applicable), followed by a summary of the identified categories and how often these were mentioned by participants. To enrich our discussion, we weave in quotations of responses where appropriate. We conclude this section with interpretations and insights we gained.

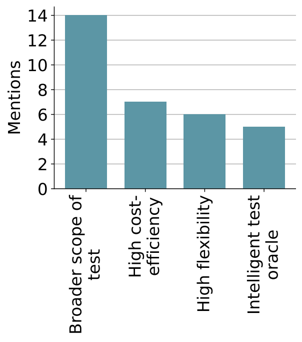
RA$_1$: Rationale behind Manual Testing

In the following, we delineate why manual testing is still applied in industry and what prevents practitioners from automating tests.

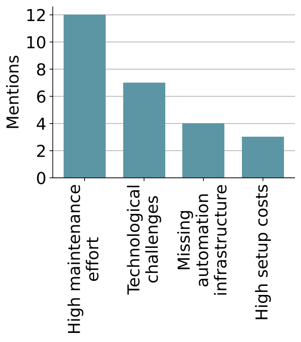
RQ$_{1.1}$: Why is software tested manually and what technological and organizational challenges hinder test automation? Figures 3.1a and 3.1b summarize the frequencies of given answers about reasons for why software is tested manually. They are grouped into advantages of manual testing and disadvantages of automated testing. We found that manual testing is deliberately employed not only because of comparatively low ramp-up costs and high flexibility, but also due to its broader scope: its exploratory character and the often associated intentional underspecification of tests. Accordingly, practitioners deem manual testing to be "closer to reality, more context-specific, and up-to-date" and more suitable when "complexity is high and requirements are blurry." Moreover, certain industries, such as the medical technology sector, prescribe manual testing.

Table 3.2: Survey questions to answer the research questions

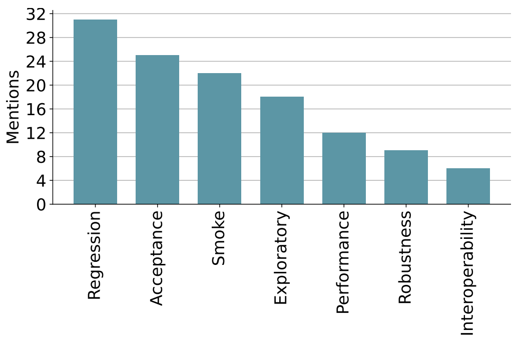
RQ	Survey question	Type
1.1	What advantages do you see in manual compared to automated software tests?	🗎
1.1	What factors force you to test manually?	🗎
1.1	What conditions and obstacles make test automation difficult or impossible?	🗎
1.2	Which test activities are performed manually?	🗎
2.1	How large is the manual test suite overall?	🗎
2.1	How many testers are there in your project?	🗎
2.1	How many test cycles take place per year?	🗎
2.1	How many test cases are executed per cycle?	🗎
2.1	How long does it take on average to run a test case?	🗎
2.1	How long does it take to execute the entire test suite?	🗎
2.2	Which events trigger the execution of a test case?	🗎
2.2	Is a successful test execution an acceptance criterion for change requests?	☑
2.2	How do developers find out about test failures?	🗎
2.2	When is a failed test case retested?	🗎
2.3	Is the entire test suite executed in every test phase?	☑
2.3	If not, how are test cases selected and prioritized for a test plan?	🗎
2.3	How are test cases assigned to individual testers?	🗎
2.4	Which interfaces are used to test the system under test technically?	🗎
2.4	Which technology-related challenges exist?	🗎
2.4	How is the System Under Test organizationally tested?	🗎
2.4	What organizational challenges are there?	🗎
2.5	Are there flaky manual test cases?	☑
2.5	How do you deal with flaky tests?	🗎
3.1	How should your testing process develop in the coming years?	🗎
3.1	Are there considerations or specific plans to carry out tests more automatically?	☑
3.1	By when should the automation be completed?	🗎
3.1	How much time is currently invested in the automation of manual test cases?	🗎
3.2	Which steps need to be taken for each test case?	🗎
3.2	Is the execution time recorded for each test case? If yes how?	🗎



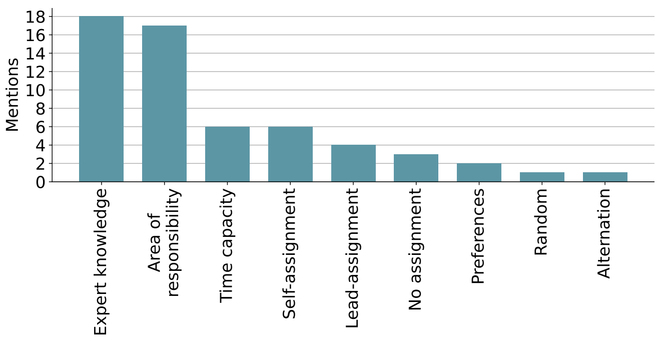
(a) Advantages of manual testing



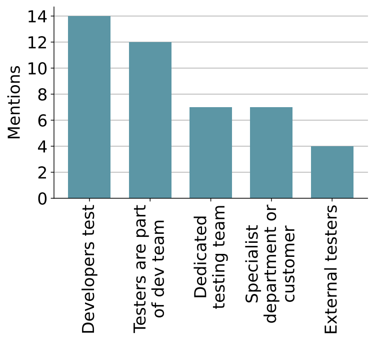
(b) Drawbacks of automated tests



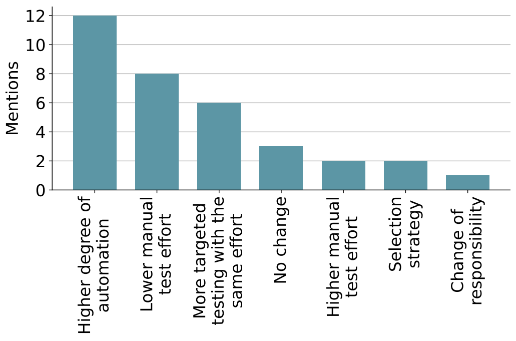
(c) Test activities



(d) Assignment criteria for test cases



(e) Manual testing organization



(f) Expected evolution of the manual testing process

Figure 3.1: Charts for survey answers

Regarding technological and organizational challenges that hinder test automation, we find the following obstacles to be prevalent (number of mentions in parentheses): Lack of time (8), lack of budget (6), limited know-how (6), limited technology (6), for example unstable testing tools or tedious creation of test data in SAP systems, interfaces to external systems (4), and high change frequency (4). One participant stated that the evolution of the software forced them to return to manual regression testing "because the [technical] environment of test automation is outdated."

$RQ_{1.2}$: Which testing activities are carried out manually in practice? Figure 3.1c shows the frequencies of different testing activities that are carried out manually. Except for exploratory testing (e. g., including *test-as-you-code*), manual tests are specified in natural language. Manual testing seems to take place at higher levels of abstraction (integration- or system level); only two participants report conducting manual tests on the unit level.

SUMMARY RA_1. *According to our participants, manual tests are more flexible than automated tests in what is tested and easier to set up. They are mostly used for regression and acceptance testing.*

RA_2: Characteristics of Manual Testing

Next, we explore the characteristics of manual testing processes of our participants.

$RQ_{2.1}$: How much effort is invested into manual software testing?

Figure 3.2 shows the distribution of manual test suite sizes and test team sizes, number of test cycles per year, number of tests per cycle, and duration of a single manual test and the entire manual test suite. The test suite sizes show a large bandwidth, between 5 and 30,000 (sic!) test cases. Interestingly, the company with the largest test suite builds software for medical devices and does not follow agile development practices. A test manager with a small test suite stated that "this is much too little. Since the construction as well as the recording of the test results costs a lot of time, some things are [...] tested quickly and only bugs are reported to DEV accordingly."

Also, the testing teams are of different sizes, with a median of 6 testers. The teams run from 1 to up to 40 test cycles per year, with a median of 4.5 cycles. Still, some testers indicate that these numbers may vary because "we claim to be an agile company, so it's difficult to give a number of times this process happens." Each cycle contains, at least, 2 and up to 4,500 manual test cases, with a median of 300 test cases per cycle, sometimes this "depends on the number of change requests—for each cycle, the number of test cases differs." The median for the duration of executing a single test case is 20 minutes, and the median for running the whole test suite is 235 person hours, with a maximum of 992 man hours.

Overall, the survey responses reveal that our participants invest a lot of resources into manual testing.

$RQ_{2.2}$: How does manual software testing integrate with the development process?

Triggers for test execution are: scheduled test phases (17), finished feature tickets (16), and deployments to test environments (14). Surprisingly, more than 25% of the participants (8 out of 31 answers) state that successful test executions are not always a necessary acceptance criterion for change requests. That is, in some cases, change requests are closed even though tests failed, which might render test execution useless.

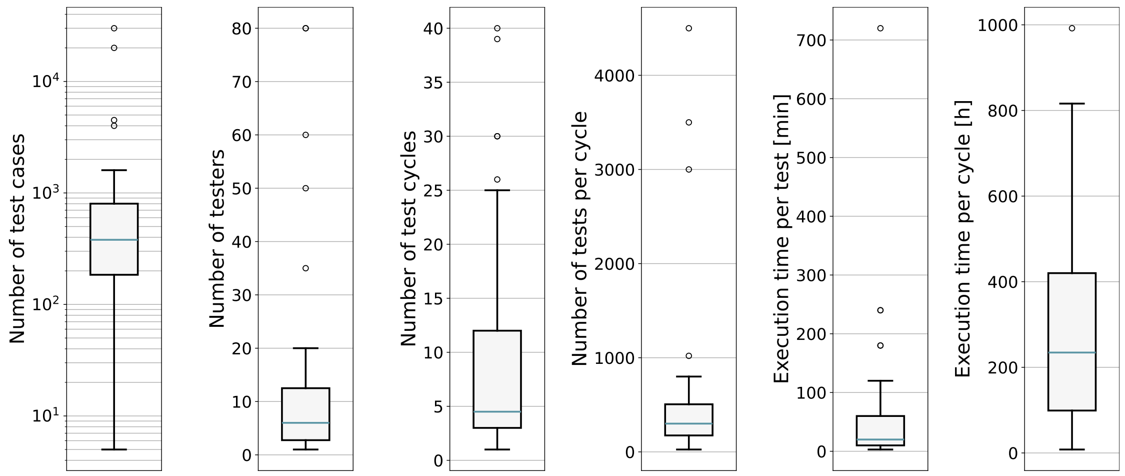


Figure 3.2: Distributions of manual test process characteristics

If a test has failed, 27 teams re-test directly after the code fix, while 10 teams re-test in the next test phase.

RQ2.3: How are test cases selected for execution and how are tests assigned to testers?

While 15 participants always execute the whole test suite, for example, because "from a customer point of view, we MUST run the 52 validation tests (which are appropriate to them), otherwise our software is potentially not valid for their use," some teams clean up their test suite before running it to avoid executing outdated tests, as one participant proposes: "all test cases that are not obsolete are performed in the annual test. This selection is performed every year." 20 other participants manually select particular test cases for execution. Their selection is based on code changes (6), tester experience (6), feature criticality (6), requirements (4), time constraints (4), or test failure history (3). Only 3 participants prioritize their selection explicitly, based on experience (2), or based on licensing or hazard relevance (1).

Figure 3.1d shows how test cases are assigned to testers, where tester experience (18) and areas of responsibility (17) dominate other assignment criteria.

RQ2.4: What are technical and organizational characteristics of (sub-) systems that are tested manually?

Most of our participants run their manual tests using the system under test's GUI (28). There are also other testing environments, for example, a browser (14), hardware in the loop (3), external systems (2), and simulators (1). Regarding tooling for running manual tests, participants adopt network communication tools (6), for example, curl, SoapUI, and Postman. Other tools mentioned by our participants are LoadRunner (1), Tosca (1), scripts (1), and Excel (2), which might also be used to manage their manual test cases.

According to our participants, the largest technology-related challenge is interference with other test environments (17), for example, because of non-isolated test systems which are used concurrently by multiple testers. Frequently, there are issues because of interactions of the system under test with other systems or applications (15), and remote test environments (12). Furthermore, different hardware combinations (4), legacy technologies (2), several test environments (2), hardware in the loop tests (1) and network latency (1) were highlighted as technology-related challenges.

Figure 3.1e shows how manual software testing is organizationally arranged. In many cases, several groups are responsible for manual tests, for example, developers test their changes in a first stage locally on their machine before a dedicated test team verifies the changes in a later stage. Some participants report that, during a test phase, people from business departments take part in testing, still, "they all come with different enthusiasm for testing", which makes it harder for test managers to plan test activities thoroughly.

Our participants highlight many organizational challenges. One major issue is lack of time in business departments for running tests (10). Furthermore, participants point out that there is a lack of domain knowledge or testing skills (7). Additional challenges are different time zones between test and development teams (6), as well as communication and documentation challenges due to different native languages (6). For some participants, the organizational spread between test and development over different organizations (2) is another challenge. In the case of fixed release cycles, a participant complains that there is lack of time for testing (3) "because we are the last but the release schedule is fix." That is, if anything delays the test execution, less time can be invested into solid validation. Another participant claims that—because different organizations are responsible for test and production environments—"the test environments do not match production environment enough, meaning it's possible that tests are passing but failing in production."

Other organizational challenges include different languages in specification, code and test cases (2), lack of time for training (1), coordination of testing (1), long time-to-fix (1), restricted testing environment (1), varying service providers over time (1), and, transforming development processes (1). Moreover, the domain can also pose challenges, for example, "medical technology is a strictly regulated domain", or might require special testing approaches "if I need a parallel test, there is a team session and everyone clicks on '1-2-3' at the same time." Perhaps interestingly, in the context of regulated medical technology, "agile development teams test on lower test levels", whereas manual testing is performed afterwards by the "test center for system test", implying a rather rigid (non-agile) development process such as the V-Model.

$RQ_{2.5}$: Do flaky tests exist in manual test suites and, if so, how do testers handle them?

Flaky tests—tests that may non-deterministically fail and pass with the same program version—are a well-known problem for automated tests [88, 94]. They are commonly first detected if a previously passing test, that is clearly unrelated to code changes introduced to the system, suddenly fails [88, 89]. Most of our participants report that they do not encounter flaky tests (20), while others report that flaky results appear from time to time (11)—five participants are not sure whether there are flaky tests in their test suite. Only five participants answered the question about how they deal with flaky tests: two participants re-run tests that are deemed flaky. Three do nothing, because deviating results are "explained away", another participant puts this more diplomatically, "most of the time, the deviation turns out to be an unnoticed difference in the procedure or in the data. The tests are intentionally described vaguely in the procedure in order to cover different procedures that are supposed to produce the same result."

Summary RA_2. *Our participants use manual tests extensively. More than half of the participants manually select only subsets of tests for execution. Tests are often assigned on the basis of experience of testers and areas of responsibility. There are many technological challenges including non-isolated test environments and the interaction of the system under test with other systems. Moreover, there are organizational challenges including lack of domain knowledge in testing teams and lack of testing skills in business departments.*

RA_3: Optimization Techniques in Manual Testing

Finally, we summarize our findings on optimization potentials for manual testing and how to leverage them.

$RQ_{3.1}$: Do manual test teams aim at test automation? How much time do they plan to invest?

Finally, we report on automation and optimization potentials identified in our survey. Figure 3.1f shows how our participants expect their manual testing process to evolve. Most frequently, a higher degree of automation is desired (12) and lower manual test efforts are expected (8). One of our participants appears to be quite frustrated about low investments into software testing, because she feels that "testing is somehow out. Nowadays, everyone tells us that a bug will simply be fixed when it appears in production." But there are also many participants who expect more targeted testing with the same effort (6) or even higher manual test efforts (2). The participants mentioned two process optimizations: implementation of a selection strategy (2) and a change of responsibility for testing (1), that is, a "shift left of our automated test cases from downstream quality assurance to development." Only a few participants (3) expect no change.

In our survey, we explicitly asked whether our participants aim at automation of their manual tests so that it becomes clear whether the implementation of additional optimization techniques can pay off in the long run. Only very few aim at automation of the entire manual test suite (2). Most participants aim at automation of some manual tests only (20). One participant points out that their goal is the "optimization of test efforts—this can mean automation, but does not have to be." Some participants also aim at no automation at all (9). Contrary to our intuition, even though most of our participants are repeatedly testing their system under test via its GUI, there are technical and organizational reasons for not automating manual tests: For instance, "frequent changes on the GUI" that disallow maintaining automated GUI tests and, according to a participant, it is "difficult to predict how much effort automation will cause because sometimes a small thing only works with extreme effort and therefore makes it difficult to plan."

For those who aim at (partial) automation of their test suite, we asked two additional questions to learn about their automation schedule and investments into test automation. 21 participants answered the question on when the automation is planned to be finished, but most of them have no specific plan when automation will be finished (18). The three participants who have a schedule plan to finish the automation of their test suite within the next 1–3 years.

Regarding the resources that are currently invested into test automation, only few invest, at least, one full-time equivalent (4). The other participants claim that, at least, one person works one day per week (5) or, at least, one day per month (5) on test automation. A handful of participants is investing no effort into test automation (5), even though they plan to automate tests in the future.

RQ$_{3.2}$: What potential for optimization of manual software testing exists and what are its prerequisites?

In Section 3.1, we have approached this question from a *scientific perspective* by reviewing existing work on manual testing. This way, we have identified six techniques listed with their associated prerequisites and caveats in Table 3.1.

From our empirical study—taking a *practical perspective*—we identify further levers for optimization and derive respective optimization techniques: First, several participants report that there are test steps that need to be taken for each test case. Among these login to the system under test (14), creating and loading test data (10), and setting up the system under test (5) are the most common. However, only a single participant noted that they have "tests for which recurring activities are modeled with shared steps." Hence, we identify an optimization lever as the prevalence of *repeated, similar test steps*, which could be tackled by *reusing test steps*, (e. g., by means of shared test steps). This can reduce duplication and increase test suite maintainability.

Second, we found that 9 participants track the test duration either manually (2) or automatically (7). It would not make sense to track it if it did not vary among test cases and test runs. RQ$_{2.1}$ suggests that there is, in general, a large spread in test duration. At the same time, in RQ$_{2.3}$, we found that time capacity is among the three most common test case assignment criteria. Consequently, we deem the prevalence of *varying test duration* between tests to be an optimization potential that can be exploited by *test schedule optimization*: If test duration is recorded and varies, a test schedule can be generated that meets time capacity, resource, or test precedence constraints, while minimizing total testing time. Since test execution scheduling has been studied for automated testing already [106], the most straightforward approach is to transfer these techniques to manual testing and to study their effectiveness.

Third, throughout our survey and specifically in RQ$_{1.1}$, we observed that non-exploratory manual tests are often deliberately under-specified to nudge exploratory testing. This sounds contradictory at first, because it potentially leads to non-determinism and false-negative or false-positive test results; but it seems to be one of the most popular features in manual testing, as one test case can express an entire equivalence class. Thus, one optimization lever would be to implement *flexible execution paths and test oracles* that allow the design of under-specified test cases which are still useful.

Table 3.3 lists the optimization levers that we identify, three derived optimization techniques with associated prerequisites and caveats. Together with Table 3.1, these make up the set of techniques that we integrate in a manual testing process model in the next section.

SUMMARY RA$_3$. *The overwhelming majority of our participants does not plan to automate their entire manual test suite; GUI test automation is often no option for technical and organizational reasons. Therefore, optimizing their manual testing processes is advisable. From our study, we identify 3 optimization levers.*

3.2.5 Guidelines for Optimization

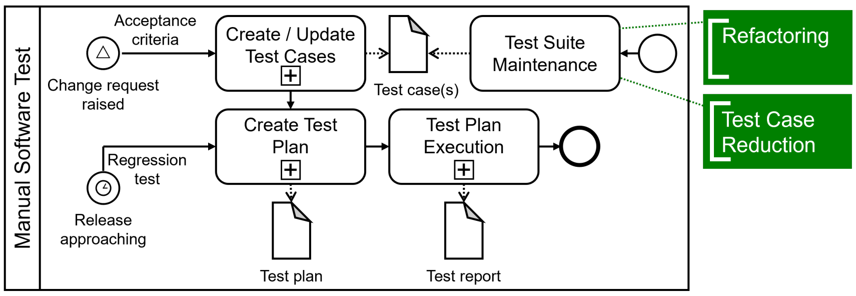
We aim at a set of actionable guidelines that empowers practitioners to quickly identify optimization potential in their context. Therefore, we have collected characteristics of manual testing processes in our survey. In addition, we have collected and derived optimization techniques for manual testing with associated prerequisites and caveats from related work and practice (see Tables 3.1 and 3.3).

Table 3.3: Prerequisites and caveats of derived optimization techniques based on identified existing levers in practice (extension of Table 3.1)

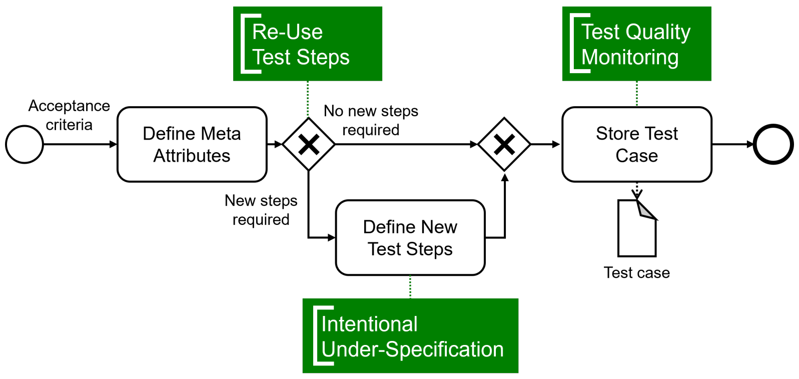
Levers	Prerequisites	Caveats
	7. Re-use Test Steps	
Repeated, similar test steps	Possibility to identify and manage test steps	Over-use of *shared* test steps
	8. Test Schedule Optimization	
Varying test duration	Measuring and documenting of test duration	Time constraints, expert knowledge constraints
	9. Intentional Under-specification	
Flexible execution paths and test oracles	Deterministic test oracles per execution path	False positive or negative test results, flaky tests

To embed these findings into an actionable set of guidelines, we devise an annotated empirical process model for manual testing. We modelled the testing processes described in the survey answers and merged them into one general manual testing process model. Although based on the empirical findings from our study, we deliberately keep the process model generic to allow practitioners to easily adopt it to their needs. We use a standard business process modelling notation (BPMN) to model the specifics and variety of manual testing processes that were described by our participants. Practitioners can instantiate the process model by identifying events, actions, message flows, and artifacts of their testing process. Based on this instantiation, practitioners are guided in the selection of the optimization techniques that are most relevant to them, for example, because they address bottlenecks in their process. Using Tables 3.1 and 3.3, specific optimization approaches can be selected, based on prerequisites and acceptable caveats, and implemented in their process. For example, in the case of manual regression testing, the trigger of the manual software testing process might be an approaching release. In Figure 3.3a, this triggers the sub-process *Create Test Plan*, which is unrolled in Figure 3.3c, and its first activity is the identification of relevant test cases. The annotation shows that test case selection techniques can be used to optimize this activity. Table 3.1 collects prerequisites and caveats of three test case selection strategies, and it guides practitioners in their assessment of the applicability of the strategies in their context.

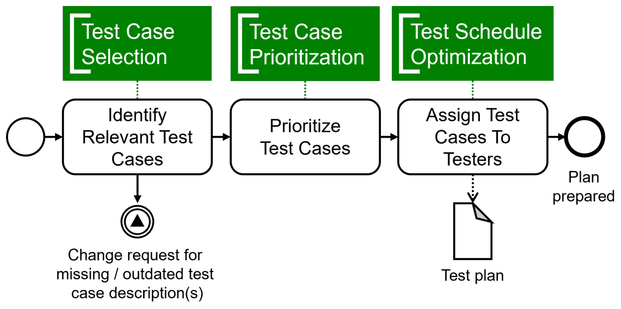
Manual Testing Process Depending on the specific test process, there are different start events (○▸) that trigger the manual testing process (i. e., acceptance criteria or a scheduled regression test phase—other manual testing activities can also be covered by our model). Activities in Figure 3.3a labelled with ⊞ are sub-processes, which are explained in more detail



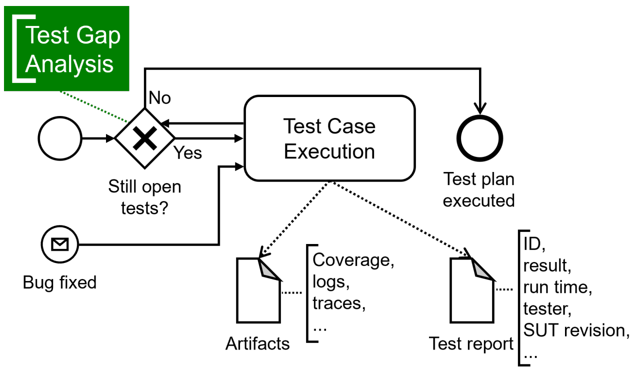
(a) Manual software test process



(b) Test case creation


Test Case Selection
Test Case Prioritization
Test Schedule Optimization
Identify Relevant Test Cases
Prioritize Test Cases
Assign Test Cases To Testers
Plan prepared
Change request for missing / outdated test case description(s)
Test plan

(c) Test plan creation



(d) Test plan execution

Figure 3.3: Optimization potentials (green) in the empirical manual testing process

in the following paragraphs and figures. The optimization techniques *Test Case Refactoring* and *Test Case Reduction* can be applied most easily during test suite maintenance.

Test Case Creation Figure 3.3b depicts the *Test Creation* sub-process. Test steps can be *Re-Used* when tests are specified and require the same steps that are already documented for an existing test. When new test steps are defined, *Intentional Under-Specification* can be applied. That is, the test can be defined generically such that several cases are covered. For example, if there are multiple ways to trigger a functionality, the test can deliberately not specify which way to use in the test. When the test case is stored in the test management system, the *Test Quality Monitoring* can be triggered.

Test Plan Creation Figure 3.3c shows the *Test Plan Creation* sub-process. Initially, the set of test cases that should be executed needs to be identified, optionally using *Test Case Selection*. Next, a prioritization of test cases needs to be done (which can be optimized using *Test Case Prioritization*). Finally, tests need to be assigned to testers where *Test Schedule Optimization* techniques can optimize matching testers and tests while considering relevant constraints.

Test Plan Execution Figure 3.3d shows the *Test Plan Execution* sub-process. The *Test Gap Analysis* can be used to determine whether test end criteria have been fulfilled. For example, it may reveal additional test opportunities from untested code changes. In case there are large amounts of test gaps, a prioritization of test gaps might be of help. In Chapter 5, we present an approach to prioritize test gaps by estimated risk.

3.3 Two Industrial Case Studies

To demonstrate the applicability and usefulness of our guidelines, we conducted two industrial case studies with testing teams from different contexts (i. e., domain, company size, test process, and technologies). We instantiated the process models of Section 3.2.5 to identify applicable optimization techniques. Together with the test leads, we then validated the suggested optimization techniques, and they decided which of these to implement. In the following, for each case study, we first introduce the study subjects to provide necessary background information. Then, we document the instantiation of the process model and, finally, we summarize the feedback of the test teams when we presented our results to them in Table 3.4.

3.3.1 Case Study 1: User Acceptance Testing

Our first study subject adopts user acceptance testing. In what follows, we provide additional background information, discuss the applicability of optimization techniques in their context using our process model, and conclude with feedback from the testing team.

3.3.1.1 *Background Information*

Our first study subject is owned by Munich Re[1], an international company in the finance and insurance domain with approximately 40 thousand employees. The business information system is customized in ABAP, the custom code base counts 2.1 million source lines of code. At the time the interviews were conducted, a team of five testers did manual user acceptance testing. There are approximately seven releases per year, each release has a predefined duration of 6–8 weeks. For each release phase, the set of change requests (*product backlog items*), which the product owner committed on and which were prioritized by the users for the current release, needs to be tested. That is, the manual software test process is triggered by new change requests, for example, by product management or users. The software life cycle management platform Azure DevOps with the plugin Azure Test Plans[2] is used to manage test cases and results.

3.3.1.2 *Applicability of Optimization Techniques*

Following our process model in Figure 3.3a, we were able to suggest five optimization techniques for our first study subject, which we discuss next.

Test Case Creation Test steps are not re-used, but structurally identical test cases are typically filed as parameterized tests for which different input and expected output values are given. This uncovers the first optimization potential: *re-use of existing test cases and steps* from former releases that have checked change requests in the same code methods. The idea is that test cases can be re-used entirely or with small modifications (e. g., new input values) if they test changes in a method that a previous test already verified. This requires that testers know which code is expected to be changed for the current change request, and testers need to be able to identify former tests that have executed this code.

The second and third optimization technique during test case creation (see Fig. 3.3b) offer no additional optimization potential: intentional under-specification of tests is not applicable for user acceptance tests in this case study, as user acceptance tests are not meant to be re-executed in future release phases per se. Test cases are already automatically checked for documentation quality issues, for example, ambiguous formulations or redundancies[3].

Test Plan Creation In the current testing process of the study subject, test cases are never re-used, which prevents optimization techniques such as test case selection, test case prioritization, and test schedule optimization. Yet, *test case selection* would help to identify relevant test cases if test cases or, at least, test steps are re-used. To benefit from *test case prioritization* and *better scheduling opportunities*, a proxy for the costs of test executions needs to be monitored, for example, the duration of test runs which is already tracked in the study subject's test management system.

1 CQSE is a contracting partner of Munich Re, the background data provided is from 2021

2 Azure Test Plans: `https://azure.microsoft.com/de-de/services/devops/test-plans/`

3 At the time of conducting this study, the tool Scout from the company Qualicen has been used. In the meantime, this tool is not available anymore. The last release was version 5.4-1: `https://www.qualicen.de/release-qualicen-scout-5-4-1/`

Test Plan Execution The optimization technique during test plan execution, a test gap analysis, is already used by the team[4] to reveal untested changes that should not be deployed to the production environment before a test happened.

Test Suite Maintenance Tests are currently not re-used, so, we see no benefit of *test case refactoring* for this study subject. Some tests appear to be partially redundant, so *test case reduction* is promising.

3.3.1.3 *Developer Feedback*

Based on our recommendations, the test lead decided to implement the *re-use of test steps* in their manual testing process. Regarding the previously mentioned prerequisites, the development process has been changed as follows. First, the development team passes information on which code is planned to be changed to the test team. Second, the authors implemented test-wise coverage recording for the team, so that similar former test cases can be identified. For this purpose, the non-isolated testing environment is profiled in a user-specific way, which helps identify re-use opportunities. The testers highlighted that they like the newly created "transparency regarding which code is being executed by their manual tests." The test lead pointed out that "it would have been great to have this tool from the very beginning of the project, where even more tests were run." Now, the system under test is so large that many test runs (and thus, code changes) are needed until all actively maintained code regions are profiled. The team has started to re-use test cases where possible, even though, typically, not the entire test case can be re-used.

According to the test lead, at the end of a test phase, she again runs the *test case selection* on changes of the current release. This outputs a set of test cases that contains usage scenarios for the changed code, and thus, additional testing opportunities. Hence, she is not only using the selection as originally intended, but uses it as inspiration for additional testing. She considers this helpful because it "lowers the risk of missing relevant test cases" and increases the likelihood of detecting faults before deploying the system under test to production. As far as *test case prioritization* is concerned, the test lead stated that "the order of the selected tests does not matter that much" because she "checks all selected tests to see if the team missed testing opportunities." She thinks that *test schedule optimization* "might be helpful for manual testing in general", but for her project, she doubts that "the input data is accurate enough." In contrast, she liked the idea of *test case reduction* because they often have to test similar functionality via different interfaces, and she would like to "reduce [...] redundancies."

3.3.2 Case Study 2: Regression Testing

Our second study subject pursues a different goal with their testing than the first study subject, that is, regression testing. The implications on applicable test optimizations of the different testing process become clear when instantiating our process model, as we will discuss in the following.

4 Teamscale: `https://teamscale.com/`, see also Haas et al. [54]

3.3.2.1 *Background Information*

Our second study subject is an application from IVU Traffic Technologies, one of the world's leading providers of public transport software solutions. The company employs more than 700 people worldwide. We focus on the manual regression testing process for one software product (primarily written in C++, with more than 700 thousand source lines of code) that is concerned with duty planning. At the time the interviews were conducted, one tester was manually testing the product full time and thirteen additional testers provided targeted testing support for releases. The test management software in use is TestLink[5], an open-source tool that is modified to suit the company's needs.

Table 3.4: Developer feedback: ✓ has been implemented, 🕒 can be implemented in the future, and × will not be implemented

Study Subject 1			
User acceptance tests	5 testers	6–8 week cycle	2.1 M SLOC ABAP
Applicable Optimization Techniques		**Feedback**	
Re-use of existing test cases and steps		✓	
Test case selection		✓	
Test case prioritization		×	
Test scheduling optimization		×	
Test case reduction		🕒	

Study Subject 2			
Manual regression tests	1+13 testers	12–16 week cycle	700 K SLOC C++
Applicable Optimization Techniques			**Feedback**
Test case prioritization			✓
Test case selection			×
Test case reduction			×
Test case refactoring			🕒
Test quality monitoring			🕒
Test plan optimization			🕒

3.3.2.2 *Applicability of Optimization Techniques*

Again, following our process model of Section 3.2.5, we were able to suggest 6 optimization techniques that are applicable for the second study subject.

Test Case Creation The test management software does not support the management of *individual* test steps, which prevents a re-use of similar test steps. Furthermore, existing tests

5 TestLink: `http://testlink.org/`

are already deliberately under-specified to enable more exploratory testing. The first applicable optimization technique is *test quality monitoring*: Test cases of the subject are constructed using natural language descriptions, which can easily be checked by automated monitoring tools for textual quality analysis.

Test Plan Creation Minor releases are tested only with a set of manual *smoke* regression tests (~30 test cases). For major releases, a larger test suite (~360 manual test cases) is executed, in addition. In general, there is no individual prioritization or selection of test cases. However, a subset of test cases called "developer tests", which cover substantial functionality, are first executed, to prevent blocking other test cases. As the name suggests, these tests are executed by developers during development before the testing phase begins.

Testers implicitly create a test history by marking tests as "passing" or "failing" during their execution. These test reports form a valuable artifact for optimization of the test plan. Both *test case selection* and *test case prioritization* can benefit from failure prediction models that solely rely on such information as shown by prior research on automated [31] and manual testing [66, 67, 87]. In addition, the textual descriptions could further be leveraged using natural language analyses [67, 87].

Test schedule optimization is not directly applicable, as the requirement of measuring test duration is not fulfilled, yet. However, we still assume that there are two other levers for optimized test scheduling: First, test assignments can be easily automated as they are currently manually derived from prior test plans. Second, test cases in the test management software may contain links, which define precedence or resource constraints. We propose to use existing automated techniques for generating an optimized test schedule that satisfies these constraints [106].

Test Plan Execution Test gap analysis is infeasible, as there are no test traces recorded during testing.

Test Suite Maintenance Since test descriptions and test failure history are available, the optimization techniques *test case refactoring* and *test case reduction* are applicable. They can be applied to create a reduced test suite that is easier to maintain, query, and extend [8].

Developer Feedback Together with the test lead, we identified *test case prioritization* to be the most promising of the proposed optimization techniques: Accordingly, it makes sense to execute those tests first that found bugs before, "in the expectation that they will be more likely to find bugs again and thus start fixing them sooner". Therefore, we implemented a simple prioritization strategy, where tests that have failed before are executed first. This proof-of-concept already reduced the feedback time compared to the current random ordering of tests.

We decided to discard *test case selection* and *test case reduction*, as the test lead pointed out that existing test cases "in principle already represent a rather coarse-meshed coverage of the most important features", making further selection or reduction unnecessary.

Closer consideration of *test quality monitoring* and *test case refactoring* is generally of interest, as it is already known by the developers of the subject project that the "nature of test case descriptions has evolved over time, test cases vary widely in the quality and scope of the descriptions”. However, implementing such techniques has lower priority than test case prioritization inside the testing team. Finally, *test schedule optimization* is already informally done in the subject project by manually keeping track of which test cases were executed by which tester before. Yet, automated assistance in the test assignment could still be helpful: “It would be conceivable to provide guidance to testers in selecting unknown test cases through tags on the test cases (e. g., required specialized knowledge).”

3.4 Discussion

We discuss the lessons learned from our case studies as well as potential threats to validity.

3.4.1 Case Studies

Both test leads find the recommendations of our guidelines useful. For our first study subject, the test process and environments were changed on our recommendation derived from our optimization guidelines, so that test steps can be re-used, which saves test creation efforts. Another optimization is employing more in-depth testing because selection of former test cases is used as inspiration for additional tests. In the second case study, using our guidelines, we were able to identify and exploit the potential of reducing test feedback times by test case prioritization based on test failure history.

Overall, during the case studies, our guidelines provided a goal-oriented structure for the discussion of bottlenecks in manual testing and helped the developers to focus on most relevant optimization techniques. Thus, they are well suited for discussions with testing practitioners to understand their process and tools, and help to communicate levers of optimization techniques.

Our guidelines summarize optimization techniques that are suitable to address bottlenecks in manual testing. From our case studies, we learned that the evaluation of techniques for their practical applicability is guided well by the presented prerequisites and caveats. In both case studies, the guidelines have shown to be effective: For the first study subject, re-use of existing user acceptance tests has been improved, and a variety of tests has been increased by test case selection. In the second case study, feedback time could be reduced by prioritizing tests based on failure history.

Nevertheless, further research on optimization approaches for manual testing is necessary. Our guidelines can be extended towards this goal, and we are happy to receive merge requests in our supplementary repository (see also Sec. A.1).

3.4.2 Threats to Validity

Internal Validity A threat to internal validity arises from the Rosenthal effect [135]: The framing of our survey questions could have influenced our participants, for example, by stating unbalanced advantages and disadvantages of manual testing. We chose the formulations of our survey neutrally, and we did several rounds of pretests with academic experts as well as testing professionals from our target group to reveal potential misleading formulations and misunderstandings. We refer the interested reader to our supplementary repository (see Section A.1) for more information and replication.

The set of guidelines presented in Section 3.2.1 is not meant to be complete. We focus on optimization techniques and levers that we have identified in our survey.

External Validity We selected the participants of our survey from a small target group. We deliberately chose this group because, this way, we could validate answers and clarify open questions with participants to get a better and clearer understanding of manual testing processes in industry. Nonetheless, manual testing might be used in other ways, which limits the generalizability of our results—a common issue in empirical software engineering [144]. In particular, answers given in the survey indicate context-specific challenges, such as regulations for the development of medical technology or complexity of GUI testing, which need to be further investigated. Still, the different project backgrounds and processes provide deep insights into the variety of manual testing.

From the survey answers, we derived an empirical process model, which might not be applicable to every testing process. Yet, our two case studies show that the optimization levers and techniques, as well as their prerequisites and caveats are helpful for practitioners to identify optimization potentials in their testing processes.

In general, case study research [138] is not meant to generalize, but our two case studies nonetheless demonstrate the potential of our guidelines to assist developers and test professionals in identifying useful optimization techniques for their manual testing process.

3.5 Conclusion

Manual testing is widely used in industry, despite the high cost of the human effort required. With increasingly short software release cycles while operating large manual test suites, there is a growing need for optimization of manual test processes. Yet, existing optimization techniques from automated testing are often not directly transferable, because it is unclear how to integrate them into manual testing processes and required data are often missing. Since there is no precise understanding of the practices and processes of manual testing across industry, pitfalls and optimization potential are generally unknown.

In this chapter, we have surveyed 38 testing professionals from 16 companies and different project contexts to qualitatively analyze the prevalence, characteristics, and problems of manual testing activities that enable or hinder optimization. The result of this empirical study is a set of guidelines embodied in an annotated process model that implements nine optimization techniques for manual testing. We discussed prerequisites and caveats for each technique, as they have been described in the literature or reported by practitioners during our study. We further demonstrated by means of two large-scale industrial case studies that our guidelines are useful and actionable to identify untapped optimization potential. Our two case study subjects implemented the re-use of tests, test case selection, and test case prioritization techniques. According to the test leads of our study subjects, their teams benefit from a higher likelihood of detecting faults, a reduced test feedback time, and an increased re-use of manual test cases.

4

Optimization of Automated and Manual Software Tests in Industrial Practice

This chapter shares material with a prior publication [56].

In Chapter 3, we found that several optimization techniques for automated testing can be transferred to manual testing, such as test case prioritization and test case selection. The two case studies discussed before shed light onto benefits and limitations of several optimization techniques, but further research is necessary to gather evidence of the effectiveness of specific optimization techniques and their applicability in manual testing processes. To address this gap, we conduct in this chapter a field experiment to gain deep insights into the practical benefits and limitations of two specific test optimization approaches which combine test case prioritization and selection for automated and manual testing.

Background Software test suites grow with their systems under test [169]. So, for large software systems, the corresponding test suites are typically large [128]. Large test suites, no matter whether for automated or manual testing, take substantial time to run. Besides being expensive to execute, long test suite run times also prevent early and meaningful feedback to developers [32, 67, 168]. In the study outlined in Chapter 3 among 38 testing teams of industrial software projects, we found that a single execution of a manual test suite takes, on average, 1.5 person months, in extreme cases even up to six person months [53]. The test suites of our industry research partners for this study (see Section 4.2.2) run, on average, a whole work week for automated tests, and, on average, five person months for manual tests. They suffer from late feedback and lack of resources to run all tests in reasonable time.

In Chapter 2, we outlined several techniques for improving test feedback times for long-running test suites. In this chapter, we focus on the two common techniques *test case selection* (see also Chapter 2.2.1.2) and *prioritization* (see also Chapter 2.2.1.1). While test case selection and prioritization are well-understood for automated tests [29–31, 46, 61, 62, 90, 96, 118, 127], the transferability to manual testing is challenging. Inherently, manual tests are conducted less frequently. This leads to substantial code changes between test cycles, and functionality usually tested on the system level, resulting in each test covering a significant portion of code. Beside their different nature, manual tests fail to fulfill prerequisites of existing optimization techniques [53], for example, manual tests may be under-specified, resulting in different execution traces for multiple runs of the same test case. Additionally, execution traces may not be easily separable for different test runs if manual test environments are not isolated. Finally, different test processes, software development and test environments, and test suite run times dictate how aggressive an optimization technique needs to be to provide fast feedback while still keeping the fault detection rate as high as possible.

Research Gap There is a lack of evidence of the extent to which test optimization techniques for automated testing can be applied to manual testing processes in industrial practice, and what limitations need to be accepted.

Solution To address this research gap, we, first, analyze the processual differences between manual and automated testing in five large-scale industrial software projects, and we investigate whether their different test processes require different optimization strategies. Our aim is to discern the implications of the implemented test processes for the suitability of different optimization approaches. Second, we conduct a field experiment [150] applying two optimization techniques to five industrial software projects that implement automated or manual testing. Specifically, we apply two general language-agnostic optimization approaches to ease setup and allow for comparison of results between our study subjects: (1) test impact analysis, a code-change-dependent test case selection and prioritization approach [78], and (2) Pareto testing, a technique that performs a static test case selection that maximizes test coverage while minimizing test execution time.

Conduct and Results To identify process differences between manual and automated testing, we have conducted a survey among our industry research partners. In our field experiment, we use historic test runs of our real-world study subjects to evaluate the costs and benefits of two general and language-agnostic test optimization techniques. Our results show that they are applicable in practice for automated and manual testing processes. For automated tests, 80% of failures are detected by the optimized test suites, on average, while saving 66% of execution time, compared to 81% failure detection rate for manual test suites and a time saving of, on average, 43%. All five industry partners that participated in our empirical study have adopted test impact analysis or Pareto testing into their processes following our results.

Contributions Our contributions in this chapter are the following:

- *Field Experiment on Five Industry Projects.* We investigate two language-agnostic test case selection and prioritization techniques on a set of five test suites suffering from long execution times from industrial software projects from various domains, using different technologies, implementing manual and automated test processes.
- *Differences between Automated and Manual Testing.* We carve out differences that are relevant for test suite optimization between automated and manual testing processes by surveying five test engineers and by querying the corresponding test suites.
- *Experiment on Test Optimization Applicability in Practice.* We conduct a field experiment to learn to which extent the optimization techniques are applicable to solve the issue of long-running test suites.

- *Analysis of Test Histories.* We analyze real data from more than 43,300 test cases, including test-wise coverage and, in total, 2,622 test failures from the study subjects' test histories to gain insights on the effectiveness of the optimization techniques.
- *Practice-oriented Guidelines.* We provide a set of lessons learned enabling practitioners to implement the most suitable optimization technique in their testing process.

4.1 Foundations for Our Study on Optimization of Manual Tests

Our empirical study (see Section 4.2) pursues the goal to create transparency on testing strategies in practice and investigates the applicability of two optimization techniques for automated and manual testing. In this section, first, we discuss related work from the field of manual test optimization, focussing on the transferability of automated testing, and the typical study setups for manual test optimization research. Afterwards, we introduce specific variants of optimization techniques that we established in Section 2.2 and which we implemented for our empirical study.

4.1.1 Related Work on Manual Test Optimization

We briefly extend the discussion on transferability of optimization techniques from automated to manual testing from Section 3.1 with a differentiation of the study presented in Section 4.2 from related work. This is followed by a discussion on the scope of studies presented in related work and how our study differs from prior research.

4.1.1.1 *Transferability of Optimization Techniques from Automated to Manual Testing*

Although manual testing is widely used for large, complex, and regulated systems [53], prior research on test optimization has primarily focused on automated test suites (see also Chapter 3). As the execution time of a manual test case takes typically several orders of magnitude longer than for an automated test, the underlying issue of unmanageable test suite run times is even worse for manual testing [37, 53]. Consequently, there are several approaches that transfer results from automated test optimization to manual testing. Test case selection techniques [24, 28, 53, 76, 111, 174] as well as test case prioritization [5, 66, 87, 143] have been discussed, but only on a single or several similar subjects. In our empirical study, we include subjects from different domains relying either on automated or manual testing processes and implementing their system under test in different languages. Other optimization techniques refer to failure prediction [47, 67], test automation [86], and test suite reduction [24, 141].

4.1.1.2 *Study Setups in Manual Test Optimization Research*

In Section 2.2, we gave an overview about different test optimization techniques and provided references to core contributions of the field. Also, the optimization techniques we are using in this work are based on well-researched approaches. In this section, we delineate the setups of our work from prior research.

Evaluation Focused on Open-Source Systems Most of the work evaluating the optimization techniques discussed so far focused on seeded faults and open source systems. More recent work incorporates other oracles, too, but still rely on open source systems as their study subjects for availability reasons. Peng et al. investigated several approaches for test case prioritization [124]. They combined and analyzed coverage, cost, historical failure, and information-retrieval-based prioritization approaches, and evaluated them on a large set of open source projects using faults from the projects histories. Cheng et al. prioritize test cases for cloud configuration testing and evaluate their work on 5 open source docker images [18]. Wang et al. combine test case selection based on code and configuration changes, and use the same subjects as Cheng et al. to evaluate their approach [161]. Yaraghi et al. prioritize test cases in continuous integration contexts and evaluate their work on a set of more than 400 open source projects [165]. We conduct an empirical study on large industrial systems which come with their own difficulties and might show different behavior from what has been seen in open source projects.

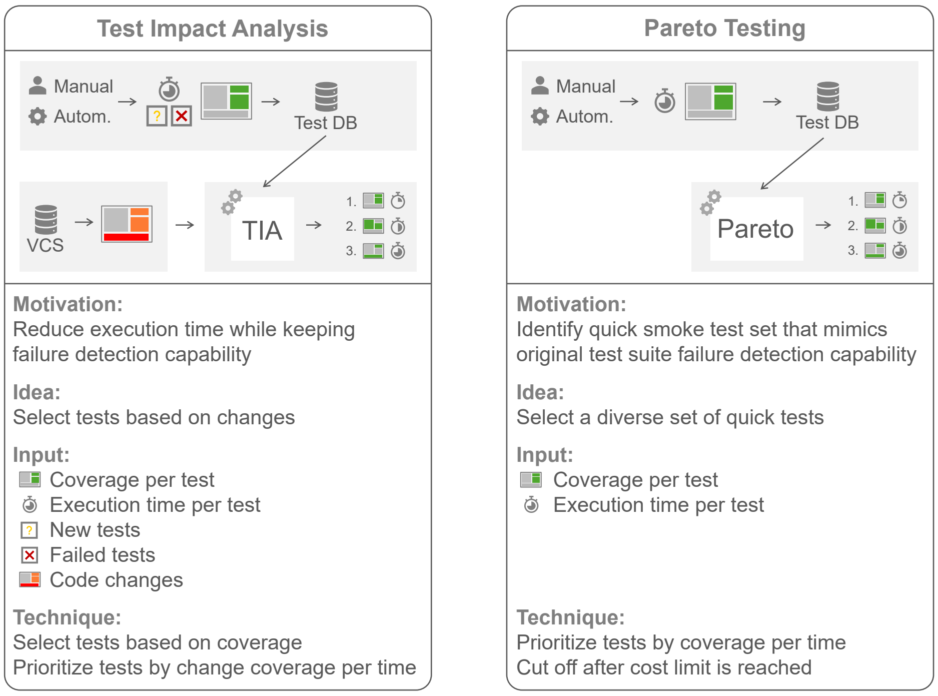
Evaluation Based on Industry Systems There have been several publications focusing on test case selection or prioritization for industry projects. In most cases, they are using a single industrial subject, often complemented by additional open source or generated subjects. For example, Marijan et al. apply a test case selection approach that focuses on coverage and historical failures on an industry project that is supplemented by generated subjects based on the industry project [97]. Their approach is based on eliminating redundancy, that is, selecting tests that do not cover the same parts of the source code and do not fail together. There are also some studies looking into Google's [101] and Facebook's [96] approaches to test optimization. These studies deliver impressive results but are based on internal Google and Facebook projects, respectively, and do not consider applicability in other contexts an important factor. Recent work on cross-language regression test case selection for C++ binaries [34] and on severity-aware prioritization of system-level regression tests [164] focuses on specific technologies and evaluates them in one specific industrial context. In summary, the focus of our work is different from most previous work in that we evaluate common test optimization techniques that fit well with our industry partners. Collaborating with them, we conducted a field experiment in a highly realistic environment across a wide range of industry systems, encompassing various languages and technologies.

4.1.2 Optimization Techniques for our Field Experiment

For our field experiment, we selected two optimization techniques. First, we provide an overview of the requirements that the techniques need to meet. These criteria were collaboratively defined with our study subjects, as discussed in Section 4.2.2. While all subjects suffered from the same problem—long-running test suites—there was a broad range of criteria concerning technologies, processes, and diverse economic, social, and legal requirements. We structured our field experiment to ensure some level of comparability across the subjects' contexts. Achieving this involved selecting optimization techniques that could be universally applied. This posed a significant challenge for several reasons. Firstly, we aimed for programming language-agnostic techniques to cater to the diverse range of real-world software systems. Secondly, our optimization techniques needed to be suitable for both automated and manual testing processes, each presenting their unique challenges (as detailed in the introduction of Chapter 4 and discussed in Section 4.4). Thirdly, our study subjects varied in size and complexity, and had slightly different optimization goals. Some focused on reducing test execution time by running a selective, change-focused subset of tests, while others prioritized running a diverse subset of tests for smoke tests. Fourthly, our subjects demanded well-established, intuitive optimization techniques that yield explainable and trust-worthy results from their perspective. Additionally, compliance with regulations was vital for systems in highly regulated industries. For instance, adherence to data protection regulations and stringent limitations on the processing of personal data (such as that of testers) based on legislation were critical considerations.

Recent techniques based on machine learning (for example, Yaraghi et al. [165]) are ruled out by these requirements, as many of the features are either not available for manual tests or not applicable in industrial contexts. For manual tests, first, features based on test source code are not available since the tests are documented in natural language. Second, since manual tests are executed far less frequently than automated tests, the number of historic test execution data available for each test case is limited. Third, in the case that historic test executions are available, features such as execution time and coverage vary more widely than they do for automated tests, which reduces their usefulness for a learning-based approach. For automated tests, we faced the additional challenge of large data sizes with some of our study subjects. In the case of Time, the test coverage report for a single test run was 13 GB in size. Since this accumulates quickly over many test executions each day, the developers only keep historic data for a single week. This, again, limits the possibility of using these data for learning-based test optimization. Finally, some data such as the committers for a file or the experience of a developer cannot be used due to data protection regulations.

Given the resource-intensive nature of manual software testing, we limited the selection to two optimization techniques for the field experiment to manage evaluation costs per study subject effectively. We selected test impact analysis because—besides fulfilling the criteria outlined above—it is based on selection and prioritization approaches that have demonstrated effectiveness in the literature [9, 17, 43, 136]. In our preliminary experiments, examining test impact analysis mostly on much smaller and open-source subjects with automated tests, we found that it performs well: For instance, the test suites optimized by test impact analysis maintain more than 90% of the original test detection capability, while saving 64% of test execution time, and have a median relative time-to-first-feedback of 2% [78]. These results encouraged us to investigate the applicability, benefits, and limitations of test impact analysis in large industry contexts including automated and manual testing, which we focus on in this work. It is important that the approaches are based on an intuitive fault model so that their results are easily understandable by the testers of our industry partners: Most bugs are introduced by code changes [134], so we assume that selecting all tests that cover changed code is intuitive to most people who work in software development. On top of the intuitive selection criteria, test impact analysis allows us to trace which tests were selected for a single changed method which, again, can give testers confidence in the technique. In ad-



Test Impact Analysis

Motivation:
Reduce execution time while keeping failure detection capability

Idea:
Select tests based on changes

Input:
- Coverage per test
- Execution time per test
- New tests
- Failed tests
- Code changes

Technique:
Select tests based on coverage
Prioritize tests by change coverage per time

Pareto Testing

Motivation:
Identify quick smoke test set that mimics original test suite failure detection capability

Idea:
Select a diverse set of quick tests

Input:
- Coverage per test
- Execution time per test

Technique:
Prioritize tests by coverage per time
Cut off after cost limit is reached

Figure 4.1: Overview of the chosen test optimization techniques: test impact analysis (left) and Pareto testing (right)

dition, our test impact analysis implementation is based on a tooling platform that allows us to handle various programming languages and testing frameworks and environments [54, 137].

Our choice of Pareto testing was based on the existing implementation of test impact analysis, which provides a very solid basis. The idea of Pareto testing (i. e., we can only detect faults in parts of a system that we cover with tests), is also easily understood. Pareto testing reuses parts of the implementation but reduces the hardware and implementation requirements for data collection and processing that test impact analysis incurs, especially for very large projects. This reduction was a requirement for continuous use for some of our industry partners. Since its implementation is based on test impact analysis, it also benefits of the above-mentioned wide support for popular languages, frameworks, and testing environments.

Details of the two techniques will be outlined in the following sections.

4.1.2.1 *Test Impact Analysis*

We instantiate a test impact analysis optimization technique from Section 2.2.1.3 as follows: In our context, test impact analysis selects test cases that execute code that has been modified within $[t_{\text{base}}, t_{\text{end}}]$, which failed in their most recent run, or which are new. The prioritization is based on a greedy cost–benefit calculation, where the costs $C(t)$ refer to the test execution time, opposing the benefit $B(t)$ of additional change coverage [114, 137]. The algorithm calculates a prioritization where for each rank $r \in [1, n]$ the test with the best cost–benefit ratio is chosen iteratively: So, in every iteration $i \in [1, n]$, $O_{tia}[i] = t_i$ is chosen such that

$$t_i = \underset{t_j \in T \setminus O_{tia}}{\arg\max} \frac{B(t_j)}{C(t_j)} \tag{4.1}$$

An overview of test impact analysis can be found on the left of Figure 4.1.

4.1.2.2 *Pareto Testing*

We use Pareto testing (see also Sec. 2.2.3.2) as a simpler alternative to test impact analysis, since practical usage at industry partners showed that obtaining and processing the input data for test impact analysis can be a substantial effort. For instance, a single test run at one of our study subjects produces 13 GB of coverage data. They implement continuous integration for all feature branches, have a large monolithic system, and more than three hundred active developers. Altogether, this setup leads to terabytes of coverage data that would need to be processed on a daily basis to run test impact analysis. Moreover, executing only a dynamically calculated subset of tests (e. g., based on the results of test impact analysis) is not always supported by the test runners implemented in industry, which can hinder the integration of test impact analysis in continuous integration setups. To overcome these drawbacks of test impact analysis, Pareto testing provides a more simplistic approach that can be set up and productively used with less effort. The Pareto test list T_p is not intended to be recalculated for every single test cycle, but less frequently (e. g., nightly or weekly), which reduces the costs of continuous coverage collection and integration into build pipelines. As a consequence, the underlying test case prioritization technique needs to be independent of a particular changeset.

We provide a brief overview of Pareto testing on the right of Figure 4.1: Pareto testing draws on ideas from the area of test suite minimization and test case prioritization [142, 168]: it collects a diverse set of quick tests. For this, it uses a prioritization technique that orders tests to always include the one that adds the most additional coverage in the least amount of time. Once everything has been covered, the existing coverage is reset and starts again. The goal of this is to order the tests so that failing tests are run early. While the prioritization mechanism is the same as for test impact analysis (see Equation 4.1), Pareto testing is independent of a timespan or change set, and it does not include recently failed or new tests. Second, the highest prioritized tests from the list are picked and included in the test case selection. Pareto testing is related to test suite minimization in that it selects a change-independent set of most relevant tests. However, it is not quite the same, since the selection does not aim at removing redundant tests permanently.

Formally, Pareto testing calculates an ordered subset T_p of a test suite T that runs within a cost limit of L. Each test case t has a cost function $C(t)$, representing its execution time. First, a test case prioritization is applied, aiming for an ordering O such that failing test cases t_f are executed before passing ones t_p (see Equation 2.1). Then, the maximal number k of test cases fitting into the cost limit of L is determined:

$$k = \underset{1 \le i \le n}{\arg\max} \left(\Big(\sum_{1 \le l \le i} C(t_l) \Big) \le L \right) \tag{4.2}$$

The selected test cases $t_1 \dots t_k$ that comprise T_p are then the first k elements of the ordered tests O.

4.2 Empirical Study: Survey and Field Experiment

We conduct an empirical study consisting of (i) a survey of our subjects' test leads and (ii) a field experiment to evaluate the selected test optimization techniques on our subjects' systems. By means of the survey, we collect insights into how testing is implemented in their industry contexts and how they intend to evolve their processes. We use these results to highlight differences between automated and manual testing. In our field experiment, following Stol and Fitzgerald [150], we adhere to their definition where the study takes place in a natural setting, that is, a realistic software development environment, involving changes directly impacting the studied entity, that is, the testing process. By this means, we investigate the applicability of two test optimization techniques for both testing strategies.

4.2.1 Research Areas and Questions

We address research questions (RQ) from two research areas (RA): (1) differences between automated and manual testing and (2) comparison of test impact analysis and Pareto testing.

RA_1: Test Strategies

Our study subjects employ automated and manual software testing (see Section 4.2.2). As the underlying processes are quite different—which might affect optimization potentials and goals—we compare them on the basis of our subjects.

$RQ_{1.1}$: What test activities are performed and what are major characteristics of the test processes? We collect relevant data and provide an overview over the testing activities of our subjects to better understand their test processes and goals.

$RQ_{1.2}$: How much testing and maintenance effort is invested into automated and manual testing? We investigate the divergent maintenance and execution efforts between automated and manual testing in order to uncover optimization potential.

$RQ_{1.3}$: What are major bottlenecks in the testing process? We aim to understand the bottlenecks in our subjects' lengthy test processes to identify the most effective optimization potentials.

$RQ_{1.4}$: What are costs and benefits of the current testing process? We obtain a baseline for the evaluation of our optimization approaches (RA_2) and share quantitative data on the effectiveness of individual test cases and test suites to shed light on the structural differences between automated and manual tests.

RA_2: Test Optimization

We apply two optimization techniques, a test case selection and prioritization (test impact analysis) and a minimization technique (Pareto testing), to automated and manual test suites of our subjects to learn when to apply which approach.

$RQ_{2.1}$: To what extent does the optimization technique influence the fault revelation capability of automated or manual tests? We focus on unsafe optimization techniques, which may not execute all potentially failing tests. As discussed in Section 4.1.2, this is reasonable in an industry context. Still, we strive for optimization techniques that preserve the fault revelation capability of the test suites as much as possible.

$RQ_{2.2}$: What reasons lead to missed test failures for the optimization techniques? To uncover the reasons behind missed test failures by our optimization techniques, we aim to expose their practical limitations. Additionally, we seek to determine if these limitations vary between automated and manual testing.

$RQ_{2.3}$: What are costs and benefits of the optimization techniques? Using this research question, we aim to provide guidelines when to use what optimization technique in practice, possibly including a differentiation between automated and manual testing.

4.2.2 Study Subjects

Our empirical study aims at improving our understanding of test optimization in industrial practice. To obtain meaningful results, we work with data from private, public and even public-sector companies from different fields relying on different implementation and testing technologies, see also Table 4.1. The project sizes range from a few ten thousands lines of code and teams with less than ten developers to several million lines of code with more than a hundred developers. The third column of Table 4.1 on test processes denotes whether we used their automated (A) or manual (M) test suite in our field experiment. Automated tests refer mostly to lower test levels, that is, Unit (U) or Integration (I), while manual tests are on the System (S) level. In the last column, we list the number of versions we analyzed for each subject. Depending on the type of tests, that is, automated or manual, a version refers to a different interval, as described in Section 4.2.5. The big differences in the number of versions are caused by the very different testing processes. While the automated tests of our subjects are in some cases run daily, the manual tests are only executed a few times a year and the collection of data required continuous support from our side, which limited the number of versions we were able to investigate. Also, the size and complexity of data of subject Time limited the number of versions we could analyze there.

Next, we introduce our subjects, their background, and their motivation for test suite optimization. We investigate the test processes in more detail in Section 4.3.

Time is a software vendor (name changed for anonymization), has more than 600 employees, and a revenue of more than €100 million. In our field experiment, we are considering one of their core products. For this product, they have a very large test suite of 80,000 test cases, but even though they run their tests on 50 machines in parallel, it is not possible to test all maintained versions every night.

Table 4.1: Overview of study subjects

Company	Domain	Test Proc	Test Levels	Team Size	SLOC	Lang	Versions
TIME[1]	Time Mgt.	A	U, I, S	50	8 M	Java	2
BVK	Finance	A	I, S	20	300 K	Java	543
DOLBY	Audio	A	U, I	10	28 K	C	111
ILP	ERP	M	S	5	831 K	C#	1
ZEISS	Optics	M	S	90	6 M	C#	1

[1] Subject anonymized due to NDA

BAYERISCHE VERSORGUNGSKAMMER (BVK) is Germany's largest pension group under public law and has about 1,490 employees. They build software for internal use and their customers. The project we are working with is running fast unit tests frequently, but the integration tests take around 10 h. The teams run nightly automated regression tests and to get faster feedback on their changes, they already employ Pareto testing during the day.

DOLBY works in the audio domain and has approximately 2,330 employees worldwide and offers a broad range of audio encoding, decoding and compression solutions. Some projects are using tests which need to cover lots of different configurations of audio systems, which opens up a large space of possible configurations for test cases. To avoid long feedback cycles, they were already using test impact analysis to select only the tests and test configurations that are relevant for recent changes.

ILP develops ENTRA®ERP, an enterprise resource planning system and is the smallest company among our study subjects with nine employees. They focus exclusively on manual testing, which is continuously performed—with more change-focused testing before a release, where the testing expert in the team selects tests that might be affected by the code change. Their test process includes both structured manual tests via Azure DevOps [107] and ad-hoc testing.

Carl Zeiss Microscopy (ZEISS) is a company that is manufacturing microscopes and has approximately 3,000 employees. We are looking at their ZEN MICROSCOPY Software which interfaces with their microscopes and provides control and configuration options, as well as visual results. They test their interfaces extensively with frequent automated tests, as well as manual tests on the actual hardware. They aim to test changed code as it is more likely to reveal bugs. For this purpose, they perform a manual expert-driven selection and prioritization of test cases.

On our supplementary Web site (see Section A.2), we present a table summarizing the study subjects along with some key statistics, for example, team size, the system under test's size, and the main implementation language.

4.2.3 Operationalization

We employed two methods to answer our research questions. Firstly, we designed a questionnaire that we sent to representatives of the testing teams of our subjects (see also Sec. 4.2.4). Secondly, we analyzed data from the subject's test-suite management systems, including historical test results and coverage. The following section outlines how we utilized these data sources to address our research questions. More details on the data analysis are available on the supplementary Web site (see Sec. A.2).

RA_1: Test Strategies

We conduct a survey targeting the test leads of our subjects to capture the automated and manual testing processes currently in place. This includes the process characteristics as well as its intended evolution. See Table 4.2 for an overview on the mapping between the questionnaire and our research questions.

Table 4.2: Survey questions to answer the research questions

RQ	Survey question
1.2	How many test engineers (e. g., testers, test developers) are there in your project?
1.2	How many test engineers spend their whole working time on testing?
1.2	How many test engineers work with the automated test suite?
1.2	How many test engineers work with the manual test suite?
1.1	Which test activities are performed via automated tests?
1.2	How much time do you estimate is invested into maintaining the automated test suite?
1.3	What are bottlenecks in your automated test process?
1.3	Are there flaky automated test cases?
1.1	Which test activities are performed manually?
1.1	How big is the manual test suite overall?
1.2	How many manual test cycles take place per year?
1.2	Is the entire manual test suite executed in every test cycle?
1.2	How many test cases are executed per manual test cycle?
1.1	Which events trigger the execution of a manual test case?
1.2, 1.4	How long does it take to execute the entire manual test suite?
1.2	How much time do you estimate is invested into maintaining the manual test suite?
1.3	What are bottlenecks in your manual test process?
1.3	Are there flaky manual test cases?

$RQ_{1.1}$: What test activities are performed and what are major characteristics of the test processes? We ask our survey participants about their automated and manual testing processes and what specific testing activities they implement, for example, regression testing or user acceptance testing. In addition, we consider the test suite size and the trigger events for test executions. If a manual test suite is maintained, we query its size and test execution triggers in the survey. For automated test suites, we obtain these data from the subjects' test management systems.

$RQ_{1.2}$: How much testing and maintenance effort is invested into automated and manual testing? We collect a set of statistics on our subjects' teams. To learn about the effort invested into automated testing, we query the execution time per test case and per test cycle from the test management systems. For manual tests, we ask in our questionnaire how often our subjects

execute how many test cases as well as the time it takes to run a single test cycle. Additionally, we ask about the efforts that are spent for maintaining the manual and automated test suites.

$RQ_{1.3}$: What are major bottlenecks in the testing process? Our survey contains two open questions on the bottlenecks our subjects perceive in their testing process, as well as two questions on the existence of flaky tests. We cluster the responses systematically and report the relevant insights.

$RQ_{1.4}$: What are costs and benefits of the current testing process? To obtain a baseline for the evaluation of the two optimization approaches, we approximate the testing costs by the execution time per test case and test cycle. We measure the coverage per test suite as proxy for test benefits, as well as the fault revelation probability per cycle, and the average number of test failures revealed per cycle.

RA_2: Test Optimization

To answer our research questions regarding test optimization, we rely on historic development and test data of our subjects (Time, BVK, Dolby, ILP, Zeiss). For this purpose, they provided us access to their version control systems, test result history, historic test traces, covering many months or even years of data.

$RQ_{2.1}$: To what extent does the optimization technique influence the fault revelation capability of automated or manual tests? We run both optimization techniques on historic versions from our subjects' test suites and investigate the fault revelation capabilities of the optimized test suites. An ideal optimization technique would preserve the original fault revelation capability; that is, previous test failures are still detected, while running only those tests that find faults. Still, since the investigated optimization techniques are heuristics, missed test failures are possible. Section 4.2.5 explains in detail the measurement setup for automated and manual test suites to obtain the fault revelation capabilities for test impact analysis and Pareto testing.

To answer $RQ_{2.1}$ for test impact analysis, we determine the fault detection rate. For Pareto testing, we obtain the same metric for a set of cost limits: As described in Section 4.1.2, Pareto testing takes a cost limit L as input parameter. This parameter is given by the subject's context. For our evaluation, we run the optimization technique with a set of cost limits $\mathscr{L} = \{1\%, 2\%, 3\%, 5\%, 10\%, 15\%, 20\%, 25\%, 30\%, 40\%, 50\%, 60\%, 80\%\}$.

For automated test suites, we report the detection rate of new failures (excluding subsequent failures), and we investigate whether applying test impact analysis and Pareto testing to an automated test suite leads to missed failing builds (this would imply that developers miss critical feedback as the optimized test suite passes while the original one failed). For manual tests, we do not report new test failures because data on new failures was not available at our subjects.

$RQ_{2.2}$: What reasons lead to missed test failures for the optimization techniques? To control the effort, we randomly select for both test impact analysis and Pareto testing a sample of $k = 10$ missed test failures (or all if there are fewer) for each subject and manually investigate why they were not detected.

$RQ_{2.3}$: What are costs and benefits of the optimization techniques? We quantify costs and benefits of test impact analysis and Pareto testing to be able to contrast them. The costs refer to test failures that were not detected by the approaches (see also $RQ_{2.1}$) and to a potential loss in

overall test suite coverage (see also RQ$_{1.4}$). The benefits arise from saved execution time and earlier feedback from failing tests, that is, we measure the time to first failure and calculate by which factor the optimization approaches are faster than the baseline. Since the original execution order was not available or was subject to change for subsequent runs (e. g., for manual tests), for consistency reasons, we use a random selection, rather than the default execution order, as a more challenging baseline for all subjects. To obtain the time to first failure for the random baseline, we used a sample size of 1,000. For both optimization approaches, we distinguish between automated and manual testing, if applicable. As far as test impact analysis is concerned, we state for each subject how much time is saved relatively to the total execution time. For Pareto testing, the cost limit L relates directly to the time saving. As we run our experiments with a set of cost limits $\mathscr{L}$, we first identify an optimal cost limit, which balances the time investment $\sum_{i=1}^{k} C\left(t_i\right)$ and the detected failures F_L, to answer this research question. The optimal cost limit L_o minimizes the euclidean distance to the theoretical optimum, which detects all failures F without any costs, that is, in no time. In a plot of cost limits L and their fault detection rates, L_o would have the smallest distance to the top left corner of the chart. Formally, L_o can be written as:

$$L_o = \underset{L \in \mathscr{L}}{\arg\min} \sqrt{\left(\sum_{i=1}^{k} C\left(t_i\right)\right)^2 + \left(1 - \frac{F_L}{F}\right)^2} \tag{4.3}$$

For Pareto testing, we additionally use the APFDc metric for our cost–benefit analysis, which is a cost aware modification of APFD (Average Percentage of Faults Detected) that was introduced by Rothermel et al. [136]. We do not evaluate test impact analysis using this metric, since test suites optimized by test impact analysis violate the APFDc requirement of equal total execution times. While the original APFDc metric considers cost as well as fault severity, we use a simplified version considering only cost, since we have no values for fault severities, as has been done in previous work [95, 124]. Also, like Peng et al., we assume the worst-case, a one-to-one failure-to-fault mapping, since we lack the necessary access to our subjects' infrastructures. We compare the APFDc with a random approach based on a sample size of 1,000 (the highest sampling size found for similar studies [166]). APFDc indicates how quickly faults can be found by a test suite in a specific order:

$$\text{APFDc} = \frac{\sum_{i=1}^{m} \left(\left(\sum_{j=TF_i}^{n} C\left(t_j\right)\right) - \frac{1}{2} C\left(t_{TF_i}\right)\right)}{\sum_{j=1}^{n} C\left(t_j\right) \times m} \tag{4.4}$$

where n is the number of tests, m the number of test failures, and TF_i is the first test that reveals fault i. Again, we compare the APFDc value of Pareto testing to the more ambitious random order.

4.2.4 Questionnaire and Conduct

With our survey, we collect experience from the responsible test leads at our subjects to better understand how testing is currently done in the specific industrial contexts. We strive for a detailed account of their testing processes and their real-world context, while exploring potential pathways or opinions for change. To this end, we designed a questionnaire to get an overview about the currently implemented testing processes of our subjects, their specific characteristics, and to elaborate their wishes for process changes addressing RA_1. We asked a representative of each of our five subjects, typically the corresponding test lead, to answer the questionnaire beginning in February 2023; by May 2023 all subjects answered. On our supplemental Web page, there is a table mapping our questionnaire's questions to our research questions. We used open-ended survey questions, allowing participants to describe their context.

4.2.5 Measurement Setup

$RQ_{2.1}$ is central for our field experiment. Our research is concerned with the application of two optimization techniques, test impact analysis, and Pareto testing, across automated and manual testing processes, each inherently distinct. Subjects with automated test suites typically execute the entire test suite at regular intervals, such as weekly or nightly. In contrast, manual testing involves the phased execution of all tests amidst ongoing code changes. Tailored measurement setups are essential for both processes to determine the fault revelation capability of optimized test suites. Figure 4.2 illustrates which data we include in the calculation for automated test suites (a) and manual test suites (b). In what follows, we describe the setup disparities between automated and manual testing, elucidating variations in data collection methodologies for test impact analysis and Pareto testing.

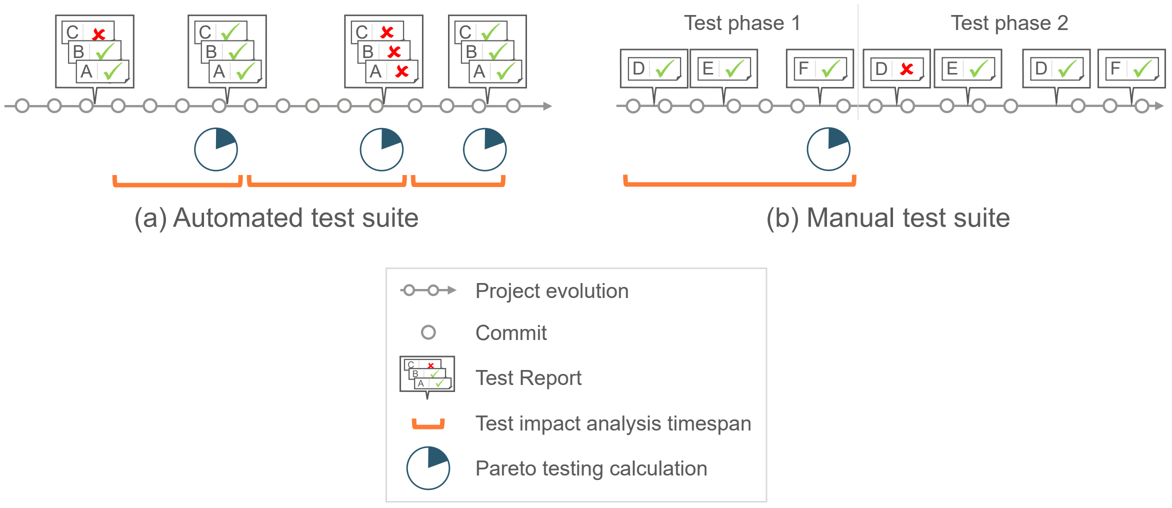


Figure 4.2: Measurement setup for $RQ_{2.1}$ to determine the fault revelation capability of optimized test suites

Automated Testing For automated testing, depicted on the left in Figure 4.2, we determine for each test run how many failing tests our optimization techniques would select if applied prior to the run. The first test report is excluded from our evaluation because we need an initial set of test coverage data to apply prioritization. We calculate a Pareto list for all (but the very first) test executions, denoted by ◔. For the calculation, we use the code and coverage state right before the next test report. For test impact analysis, we select test cases based on the code changes starting from the first commit after the previous test report up to the last commit before the next test report. In Figure 4.2, these commit intervals are denoted as ⎵ .

Manual Testing For manual testing, illustrated on the right in Figure 4.2, full coverage data is only available after the first test phase, as it requires, at least, one execution of all test cases. The Pareto list is calculated (◔) based on all test executions of the initial test phase. Consequently, the test impact analysis timespan (⎵) covers all changes for the subsequent test phase. For both optimization approaches, the goal is to reveal as many test failures of the subsequent test phase as possible. To obtain complete test reports, coverage recording per test case is required. So, we integrated our tooling deeply into the subject's test management tools to trace test begin and end to trigger and stop coverage recordings appropriately. We analyzed potential outliers with our subjects' teams and excluded manual test executions with implausible execution times (e. g., two seconds, or two weeks).

4.3 Results

In this section, we present our findings on the test processes of our study subjects (RA_1) and the cost and benefits of test impact analysis and Pareto testing (RA_2).

RA_1: Test Strategies

$RQ_{1.1}$: What test activities are performed and what are major characteristics of the test processes? Four subjects have an automated test suite used for regression testing (4), user acceptance testing (2), performance testing (2), robustness testing (1), smoke testing (1), and compatibility testing (1). Four subjects rely on manual tests (where ILP has no automated tests) for regression testing (3) and user acceptance testing (2). In addition, manual testing is used for performance testing (1).

Table 4.3: Test suite and testing team sizes (numbers rounded)

Subject	Test Cases		Test Engineers	
	Automated	Manual	Automated	Manual
Time	36,000	10,000	5	15
BVK	3,500	unknown	9	9
Dolby	2,000	0	4	0
ILP	0	800	0	3
Zeiss	unknown	1,000	5	15

The test suite sizes vary greatly (second and third column of Table 4.3): the relevant subjects have between two thousand and thirty-six thousand automated test cases. The manual test suites count between eight hundred and ten thousand test cases. All automated test suites are run regularly in a continuous integration pipeline, but the execution frequency ranges from runs per change to daily runs. Manual tests are run before scheduled releases, sometimes in every sprint, and on-demand for acceptance testing.

SUMMARY $RQ_{1.1}$. *All of our subjects with automated tests run them for regression testing, some perform additional automated test activities. The main purpose of manual tests are regression and user acceptance tests. The subjects' test suites tend to be large and are run frequently.*

$RQ_{1.2}$: How much testing and maintenance effort is invested into automated and manual testing? Table 4.3, column four and five, list the number of test engineers (e. g., testers and test developers) involved in the software testing process. Overall, there are 3 to 20 test engineers per subject. Automated test cases take, on average, 11 s; the whole automated test suite, on average, 40.9 h. Up to 30% of the development and testing efforts are dedicated to maintaining automated tests. In contrast, most of the manual test suites are run only once per release, and there are 2 to 10 releases per year. Between 10 and 10,000 test cases are run per test cycle. Running the entire manual test suite takes between a few person-days and 225 person-days. Maintenance efforts are relatively low, ranging between 1 and 18 person-days per year.

SUMMARY $RQ_{1.2}$. *Our subjects invest considerable resources into automated and manual software testing. For automated testing, maintenance efforts are relatively large; whereas, for manual testing, the execution costs dominate maintenance costs.*

$RQ_{1.3}$: What are major bottlenecks in the testing process? For automated test suites, we found three major bottlenecks in the testing processes: specifying test cases, poor test suite architecture, and third-party components, which lack quick support of new framework versions. In what follows, we illustrate them with quotes from individual participants. Related to specifying test cases, ZEISS mentions, for example, the time-consuming "creation of tests" or at BVK it takes much time to "define the test preconditions". Regarding the test suite architecture, a participant stated that they "have much more tests on system level than on unit level", that is, their test suite has the shape of an ice cream cone instead of a test pyramid [42]. ILP and ZEISS report resource capacity as major bottlenecks in their manual testing process, and BVK reports the test data configuration during test case creation as major bottleneck. Flaky tests are perceived occurring more frequently for automated test suites (two out of four subjects); no flaky tests are reported in manual test suites.

SUMMARY $RQ_{1.3}$. *Our subjects identify the test suite design and the time-consuming creation of test cases as major bottlenecks of automated test suites. Resource capacities and test case creation are the major bottlenecks of manual testing. Flaky tests are bottlenecks only for automated test suites.*

$RQ_{1.4}$: What are costs and benefits of the current testing process? Figure 4.3 shows violin plots of the execution times per test case and the execution time per test cycle (which does not necessarily run the whole test suite). On average, a single automated test case takes 11 s to run, whereas a manual test runs for 29 min, on average. A test cycle for automated tests requires between 4 h and 122 h, while manual test cycles require 16 h to 180 h. On average, the test suites cover 69% of the methods of their respective systems; there is no substantial difference between automated and manual test suites with regard to coverage. Notably, the probability of, at least, one failing test per test cycle ranges from 13% to 89% for automated

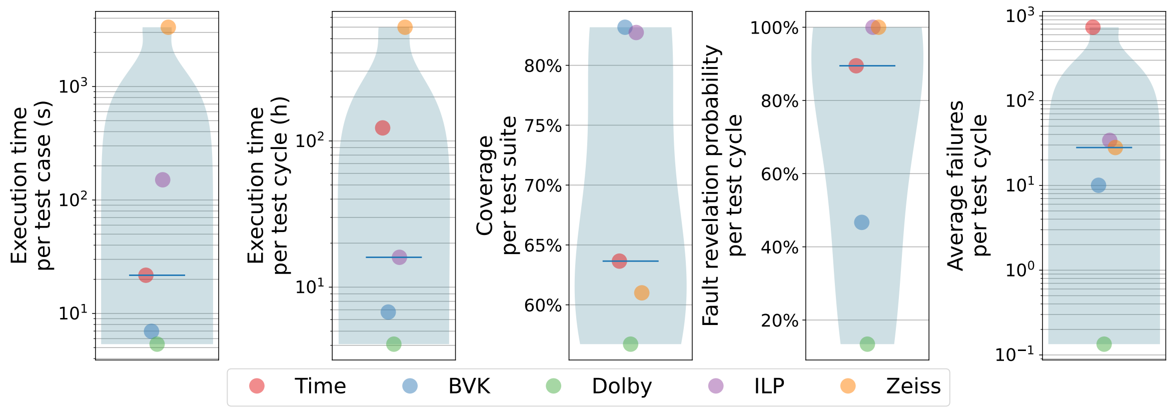


Figure 4.3: Statistics of our subjects' current testing costs (execution time per test case and test cycle) and benefits (coverage and failures) (RQ$_{1.4}$)

tests, while all manual test cycles revealed, at least, one failure. Overall, the probability of a failing cycle is 70%. On average, a test cycle revealed between 0.1 and 720 failures, where the extreme value of 720 stems from an automated test suite.

Summary RQ$_{1.4}$. *An automated test cycle of our subjects runs up to* 122 *hours, while the longest manual test cycle requires* 180 *hours. Automated and manual test suites cover approximately 69% of the methods. 70% of all test cycles produce, at least, one test failure, all manual test cycles have produced, at least, one failure.*

RA$_2$: Test Optimization

In what follows, we present the results of our field experiment where we implemented test impact analysis and Pareto testing in the test processes of our five industrial subjects and ran a historical analysis. Our subjects have recorded their test execution history including test failure information and test-wise coverage information for, at most, three months up to several years. As described in Section 4.2.5, this test history information allows us to answer the research questions on the test optimization approaches.

RQ$_{2.1}$: To what extent does the optimization technique influence the fault revelation capability of automated or manual tests? Figure 4.4 shows the fault revelation capability of test suites optimized by test impact analysis for all subjects. For all subjects, almost all build failures are detected. That is, when a build would fail with executing all tests, it would also fail with the optimized test set of test impact analysis. Only for subject BVK, test impact analysis misses 4 out of 542 build failures. For newly appearing test failures, on average, 76% are detected; an outlier is Dolby with only 21%. Overall, on average, 93% of all failures are detected.

We show the results for Pareto testing in Figure 4.5. Since test case selection is not based on changes, but on a chosen maximum execution time, we show failure detection over the range of cost limits $\mathscr{L}$ (see Section 4.2.3). We observe that the number of detected build failures is increasing very early with the test execution time, while the number of failures and new failures increase a lot slower. For BVK and Dolby, all build failures are detected fairly quickly for a comparably small $L \leq 10\%$. For Time, which has 18 different build components, with $L = 1\%$ all build failures from 16 components are detected. To detect all build failures for Time, we need to increase the test execution time limit to $L = 50\%$.

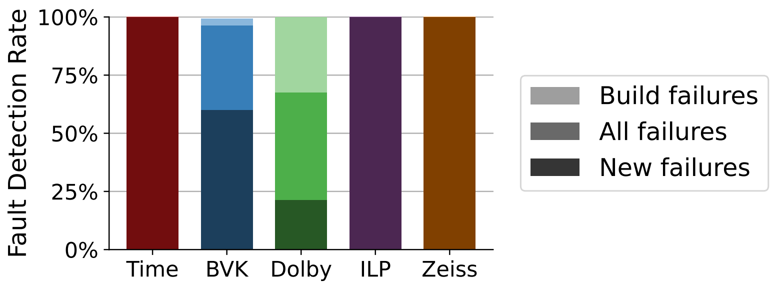


Figure 4.4: Fault revelation capability of test suites optimized by test impact analysis

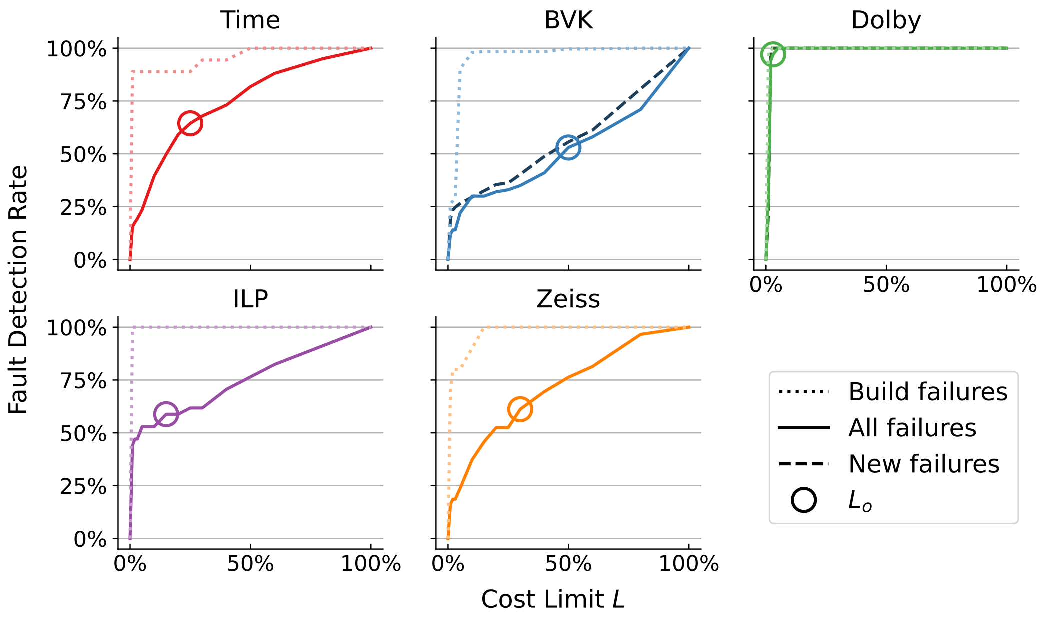


Figure 4.5: Fault revelation capability of Pareto testing-optimized test suites per cost limit L

The overall fault revelation capability varies: on the one hand, for Dolby, the cost limit $L = 5\%$ suffices to detect all failures. For BVK, on the other hand, $L = 80\%$ does not suffice to select all test failures. For BVK and Dolby, the fault revelation capability for new failures is similar to the overall fault revelation capability.

Summary RQ$_{2.1}$. *For both automated and manual testing, test impact analysis detects, at least, 60% and up to 100% of historic failures (93%, on average). For Pareto testing, the cost limit L influences the fault revelation capability considerably, up to 100%. On average, it detects 53% of failures with* $L = 10\%$.

RQ$_{2.2}$: What reasons lead to missed test failures for the optimization techniques? As shown before, both optimization techniques missed some historical test failures. We analyzed for each technique and subject, if possible, ten missed failures and why they were not selected. Below, we summarize our findings and provide examples for illustration purposes.

Test impact analysis detected all test failures for three out of five subjects. For the remaining two subjects, 16 out of 20 investigated test failures are missed due to non-code changes of the system under test, its test suite, or environment changes, which are beyond the scope of test impact analysis. It did not detect XML-specified test case changes or failures of an infrastructure test, which only checks the testing environment.

For Pareto testing, we determined an optimal cost limit L_o that balances test execution time and failure revelation. Hence, missing some test failures, mostly from longer-running test cases, is expected. We observed this in our subjects, with unselected test cases revealing failures taking 30%–250% longer than selected tests. For one subject, all missed failures belonged to test cases for which the preceding run also failed. These failures may produce incomplete coverage information if the execution was cancelled, which may impact the prioritization of Pareto testing.

Summary $RQ_{2.2}$. *For test impact analysis, 80% of missed test failures are related to a lack of coverage information for build or configuration files, which cannot be easily collected during test execution. For Pareto testing, some missed failures are to be expected due to the choice of L_o, and most missed test failures stem from long-running tests and previously failing tests.*

$RQ_{2.3}$: What are costs and benefits of the optimization techniques? In $RQ_{2.1}$, we addressed the fault revelation capability of test impact analysis (Figure 4.4) and Pareto testing (Figure 4.5). Figure 4.6 shows results regarding a cost–benefit analysis: (*a*) the relative coverage loss, (*b*) the relative execution time savings, (*c*) the speedup of time to first feedback compared with a random prioritization as baseline, and (*d*) the APFDc values of the optimized test suites. We examine these numbers in detail, since they require additional context to provide insights:

Test impact analysis for automated tests results in an average of 88% fault revelation of the full test suite. The main reason for the 12% loss in faults found are failures at BVK and Dolby, where tests fail for reasons unrelated to code changes. At BVK, this is mainly due to the test descriptions, which are XML files and thus outside the scope of test impact analysis. At Dolby, we found that many failing tests were caused by changes in the build, not by changes in the source code. The high loss of 43% coverage is mainly caused by many test runs at BVK, where test impact analysis on average selects very few tests. While this means that very little time has to be invested, it also results in very low coverage numbers. Since test impact analysis does not optimize for overall coverage but for change coverage, this is expected, when few changes happen in a time interval. In terms of benefits, test impact analysis saves 58% of test execution time, on average. Subject Time has a big negative impact on this value. The test executions that we analyze for Time have a longer interval of one week, even though their development is very active. This leads to the selection of all test cases, which reduces the overall time savings. In contrast, at BVK, we have short intervals and less development activity which allows for more time to be saved. Finally, the median time to first failure is 185 times faster compared to the random baseline, highlighting the effectiveness of change-based selection and, especially, prioritization in quickly identifying failures.

Test impact analysis for manual tests detects all historic failures and shows a negligible loss in coverage of, on average, 1% because of a comparably ineffective selection of, on average, 92% of the total test execution time. Test impact analysis selects this many test cases mainly because of two reasons. First, it needs to cover for both subjects a relatively long time span with many code changes, more than for our subject's automated test suites. Second, the manual test cases are end-to-end-tests, which cover much more code than, for example, automated unit tests. Since test impact analysis selects all test cases that cover code changes, a large set of code changes and test cases covering a lot of code lead to a large proportion of selected tests. Compared to a random baseline, the time-to-first-feedback is 4.75-times faster.

Pareto testing for automated tests has a lower fault detection rate than test impact analysis at 71%. Note that, for Pareto testing, this value depends on the optimal cost limits that we calculated (see Section 4.2.3). As shown in Figure 4.5, the cost limits $\mathscr{L}_0$ are between 0.03 for Dolby and 0.5 for BVK. Since test impact analysis selects tests based on code changes, it is expected to deliver a higher fault detection rate. The very low coverage loss of, on average, 1% is due to the fact that Pareto testing optimizes for coverage per time. The time savings are directly determined by our calculations, as mentioned in Section 4.2.3. For the automated test suites, we achieve savings of, on average, 74%. Since this selection is based on overall coverage, and not on change coverage, there is no risk of selecting all tests. The median time to first failure is 5-times faster than for the random baseline. While still a solid improvement, this is far lower than for test impact analysis. Finally, the average APFDc value of 0.79 is very solid when comparing to results of Peng et al. [124].

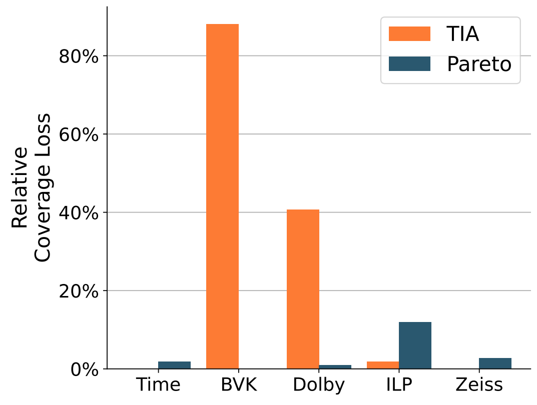
Pareto testing for manual tests results in a fault revelation rate of 60%, while saving 70–85% of execution time ($\mathscr{L}_0$ is 0.15 for ILP and 0.3 for Zeiss, see also Figure 4.5). We observe a small coverage loss of, on average, 5%. The time to first failure is 2.2-times faster than a random baseline, about half as fast as test impact analysis. The APFDc value of, on average, 0.7 is weaker than for automated tests, but still shows a good cost–benefit ratio.

Summary RQ$_{2.3}$. *For automated tests, test impact analysis maintains 88% of the fault revelation capability of the full test suite, while saving 58% of execution time. We observe a median speedup of 185 for the time to first failure compared to random ordering. For manual tests, test impact analysis selects almost all test cases, so all failed tests are detected, but the time saving is only 8%. We observe a median speedup of 4.75. For automated tests, Pareto testing detects 71% of the failures while reducing the execution time by 74%. We observe a median speedup of 5. For manual tests, Pareto testing detects 60% of the failures while reducing the execution time by 78%. We observe a median speedup of 2.2.*

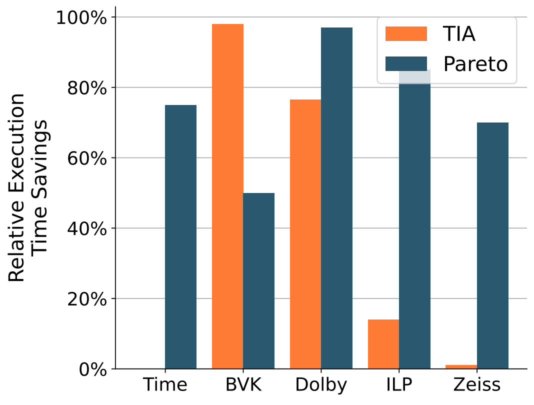
4.4 Lessons Learned

In what follows, we summarize lessons learned supporting practitioners optimizing their testing processes. Our results show that test impact analysis and Pareto testing help to reduce testing efforts in industrial-scale automated and manual software testing processes, while still revealing the vast majority of test failures (refer to results from RA$_2$). We are convinced that other industry projects can benefit from the investigated test optimization techniques for their own testing processes. To assist practitioners in evaluating the applicability of these techniques in their context, we enrich the following guidelines with insights into the test strategies of our study subjects (as per RA$_1$).

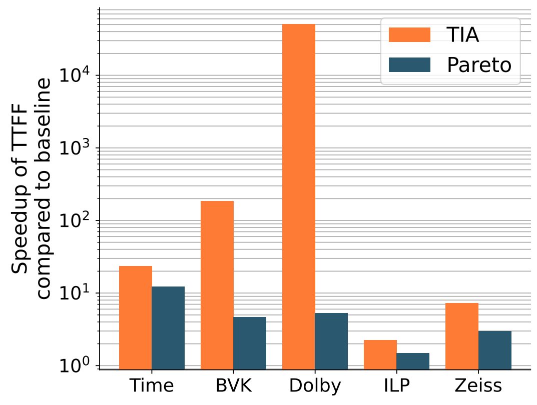
Test Optimization Technique Guidelines Test impact analysis is more sophisticated than Pareto testing; but it also comes with stricter requirements. Depending on the optimization goal, Pareto testing might be the better choice. It suggests a prioritized, diverse list of test cases within a given cost limit independently of code changes. This is useful to identify a



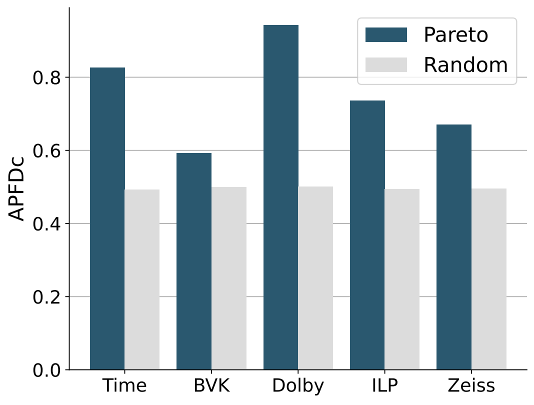
(a) Relative coverage loss



(b) Relative time savings



(c) Speedup of time to first failure (TTFF)



(d) APFDc (only relevant for Pareto testing)

Figure 4.6: Costs and benefits of test impact analysis (TIA) and Pareto testing (with optimal cost limit L_o; see also Figure 4.5)

(small) set of smoke tests, or (for very large test suites as for Time) to identify a list of test cases that can be run overnight. In both cases, it is advisable to schedule additional, less frequent test executions of the whole test suite to update the coverage data of the test cases. Test impact analysis, in contrast, suggests a list of test cases on the basis of code changes within a given time range. This is useful for risk-based testing, as done at BVK and Dolby, since modified code is more likely to contain bugs, which can only be revealed by test cases that execute the buggy code. It should be noted that, the more code has been changed, and the more test cases are required to cover all changes, the less effective is the selection of test cases. For example, for very large software projects with dozens or hundreds of developers on the same repository, for example Time, the amount of code changes of one day might already exceed the available time for a test cycle overnight. The results for $RQ_{2.3}$ have shown that test impact analysis maintains a higher fault detection rate than Pareto testing for automated and manual tests, which comes with the cost of collecting test-wise coverage information regularly.

Importance of Prioritization for Manual Testing End-to-end tests (such as most manual tests of our subjects) tend to cover more code regions per test than unit or integration tests. Moreover, manual tests are less frequently executed (2–10 cycles per year instead of daily or weekly runs, see $RQ_{1.2}$), which means that more changes need to be covered for test impact analysis. As a consequence, test impact analysis selected a large proportion of tests for both manual test suites in our subjects, which reduces the potential for time savings (see results of ILP and Zeiss for $RQ_{2.3}$). On the other hand, the subjects reported that the prioritization of test cases was very useful for them because it allows for great flexibility: The tests are executed in descending order of their fault revelation probability until there is no time left (e. g., end of planned test phase).

Variance in Manual Test Reports We observed for both subjects implementing manual testing (ILP and Zeiss) that data derived from manual test reports exhibit greater variance than those from automated tests. This is mainly due to the inconsistencies inherent with manual testers in initiating, terminating, and executing test cases, for example, test cases can be under-specified [53, 76]. For coverage- or time-based test optimization techniques, this means that a slight variance in the results needs to be expected and that the exclusion of outliers may produce more useful results.

Cost Limit Parameter for Test Impact Analysis In practice, for example, at BVK and Zeiss, the available time is often the limiting factor to run tests (see also the results for $RQ_{1.3}$). While Pareto testing has a cost limit by design, our implementation of test impact analysis does not consider such a cost limit. To address the need of an explicit optimization goal, we suggest for productive implementations of test impact analysis to add a cost limit input parameter, which is taken into account after the selection and prioritization of tests.

Transferability of Optimization Techniques Our field experiment shows that optimization techniques designed for automated testing can be applied for manual testing. To achieve uniform and comprehensive data, it is crucial to integrate test measurement tools, like a profiler, deeply into the tester's workflow. While the selection of manual test cases is less effective for our subjects, they find the prioritization very helpful. Overall, the optimization techniques provide useful results for automated and manual testing, and convinced our subjects to permanently implement them in their testing process. Concluding, this underpins the transferability of optimization techniques from automated testing to manual testing.

Post-study Optimization of Industrial Testing Processes Prior to our collaboration, the subjects were unaware of test optimization techniques, or there was no implementation available for their tech stack. Our language-agnostic implementations of test impact analysis and Pareto testing helped them evaluate these techniques within their own testing process. Notably, our field experiment's results convinced all five subjects to permanently implement an optimization technique in their testing process. For them, the benefits of optimization (e. g., the substantial resource savings and earlier feedback from failing tests) outweigh the costs (e. g., potentially missed failures and optimization data processing).

Recommendations for Researchers and Practitioners From our empirical study, we draw a number of recommendations for researchers and practitioners interested in implementing optimization techniques in large industrial software testing processes. Such processes involve many stakeholders with different experiences and stakes all of whom need to be on board and convinced of the benefits of the introduction of test optimization techniques for a successful implementation. Thus, it is vital to first understand the current testing process in detail (e. g., test strategies, tools, test environments, deployment strategies, test frequencies) and to discuss the optimization goals with the test management. This helps strengthen the understanding of people involved and to manage expectations, but also to fine-tune the optimization parameters.

We also recommend anticipating and prepare for technical challenges. Depending on the technical setup and the organization, these may be very different. Some likely issues that we have encountered during this empirical study and in other industry contexts:

- Access restrictions within test environments that needed to be addressed.
- For large systems, processing the volume of data required for the test optimization techniques poses a challenge.
- Non-isolated test environments can make it difficult to distinguish between concurrently running test cases.

Third, it is critical to involve the test teams that will be directly affected by the optimization results. Especially in the context of manual testing, using these techniques can have a big effect on the way teams perform their tests. They should be aware of the motivation for the process change, get an overview of the technical details of the optimization approach, learn about its impact on their testing process and have the opportunity to formulate their own expectations. Our experience working with manual test teams, in particular, has shown that there should be room for questions and suggestions for improvement, since the test teams usually have an excellent practical understanding of the process.

Finally, before rolling out an optimization technique to a larger testing process, we found that a dry run with a small subset of test cases or testers to identify potential blockers reduces the risk of major problems in the full rollout. In summary, we recommend treating the implementation of an optimization technique as a process change that requires active change and expectation management, as well as strong communication with all stakeholders.

4.5 Threats to Validity

Re-running failing tests threatens the construct validity of our field experiment. Our field experiment is based on historic test runs of our study subjects. This real-world data set contains some repeatedly failing test cases. Since test impact analysis selects previously failed test cases for a re-run, repeatedly failing tests can influence the fault revelation capability of optimized test suites in our field experiment. As re-running failing tests is part our subjects' testing processes, we reflected this in the behavior of our test impact analysis implementation. We discuss the impact of re-running tests on our results in Section 4.4.

Data quality presents a threat to internal validity. Both optimization techniques rely on testing data such as code coverage information. If this information is not accurate, it can lead to inaccurate test case selection and worse performance for test case prioritization. Thus, data quality is crucial for our field experiment. That is why we used field-tested tooling that is in productive use to measure test coverage, and we validated the data carefully with partners from our subjects to obtain meaningful results.

Non-code related failures threaten internal validity. For some subjects, automated test cases are specified in XML, which cannot be profiled by the profilers implemented at our subjects. As a consequence, test failures caused by test case modifications cannot be predicted by the optimization techniques, as there is no mapping between the XML test cases and the corresponding test executions. We encountered a similar situation with test cases that are concerned with build- or other non-code artifacts. In $RQ_{2.2}$, we have investigated limitations of the optimization techniques and discuss the impact of missing test specification and configuration data.

Flaky tests threaten the field experiment's internal validity as they might influence test impact analysis since it selects previously failed tests. While two subjects stated that they have flaky tests for their automated test suite, they only occur in low numbers.

The low number of versions for manual testing threatens conclusion validity. Recording the input data for manual test optimization proved to be a significant challenge in that we had to continuously support our study subjects with setup and usage of the infrastructure to obtain reliable results (see also Sec. 4.2.5). For both subjects with manual tests, it took several test phases, each of which took several months, until we got complete and reliable data. Even though the final number of versions for manual tests is low because of these constraints, the data provide unique insights for optimizing manual testing processes that are inherently challenging to study, which makes our study results all the more valuable for practitioners.

Variance in manual test times threatens internal validity. The time recordings of manually executed tests can be imprecise (see Section 4.4), which may impact both optimization techniques. We mitigated the threat of human errors by integrating our tooling as closely as possible into the regular testing process, so no additional actions needed to be taken by testers to record the data for the optimization techniques. In addition, we validated the manual test execution data carefully in close collaboration with our subjects to make sure that no invalid data go into our results. For example, we excluded outlier test cases with too small (a few seconds) or too large execution times (multiple days).

Generalizability of results in empirical software engineering research is often limited [144], which is a threat to external validity. This applies to our empirical study as well: Our subjects are not representative of all real-world software systems and their giant range of technologies and development and test processes. However, we considered a diverse set of five industrial software projects from different companies, domains and building up on different technologies. Our survey for RA_1 is meant to be a case study, so it does not claim to be representative for all industrial software engineering projects, nor that its results are generalizable across all industrial projects. We believe that it is important to understand the specific contexts of our subjects so that other researchers and practitioners can judge to what extent our results could also apply in their contexts, which is subject of an ongoing debate in the software community [14]. Still, in contrast to prior work on industrial systems, our empirical study on software testing optimization techniques is not tailored to individual subjects and, hence, is likely to be more transferable to other contexts. Our results show that the optimization techniques can be used for automated and manual testing processes, help to reduce time to feedback, and, in general, maintain the test suite's fault detection capability.

4.6 Conclusion

Software testing is a common practice in industry, including both automated or manual processes. In this chapter, we investigated to what extent optimization techniques that are typically used for automated software testing can be transferred to manual software testing. We have conducted an empirical study on five subjects from different domains, different tech stacks, varying regulatory requirements, and implementing different testing strategies. Their test processes are resource-intensive: up to twenty test engineers are involved in testing and a single test cycle runs up to four weeks. To carve out differences in their automated and manual testing processes, and their implications on optimizations, we conducted a survey among the test leads of our subjects. Then, in a field experiment, we applied two optimization techniques that select and prioritize test cases, test impact analysis and Pareto testing, and compared their costs and benefits in a historical analysis on our subjects' test suites. Our results show that both optimization techniques are applicable and effective for automated and manual testing, even on large industry systems, and yield execution time savings of up to 98% for automated tests and 85% for manual tests, while preserving a fault detection capability of up to 96%. In conclusion, test optimization strategies—such as test case selection, test case prioritization, and test minimization—traditionally used for automated tests can be effectively transferred to manual testing, with only manageable limitations to be considered. More importantly, our results have practical impact, since all of our subjects implemented them in their software testing processes.

5 Prioritization of Test Gaps by Estimated Risk

This chapter shares material with a prior publication [57].

Chapters 3 and 4 explored optimization potentials for manual testing processes. In this chapter, we focus on different aspects of human-in-the-loop testing, that is, the risk-based allocation of testing efforts and test completion assessment by test management and quality assurance roles. Due to the large amount of changes that could potentially cause defects, risky changes that might introduce a defect or affect mission-critical software can easily remain untested. At the same time, testing less risky changes is typically also less effective. The identification of risky changes is a huge challenge, which we face in this chapter. In the following, we outline an empirical study prioritizing test gaps by estimated risk, aiming for an as effective reduction of risks in the testing process as possible. For our study, we develop a score-based approach for prioritizing test gaps, which we evaluate by means of a multi-method study, consisting of a field study with our industrial partners and semi-structured interviews with quality assurance experts.

Background Functional correctness is crucial for the success and acceptance of a software product. A solid testing process is imperative to uncover defects before they are deployed in the field. Since resources are limited, especially for large software systems, it is important that test efforts are allocated such that the most critical defects are detected as soon as possible. This requires an estimation of which parts of the system are expected to be particularly defect-prone. Defect prediction research aims at revealing faulty code, often using static program analysis enhanced by heuristic search or machine learning [70]. Even though a large variety of studies has been conducted in this area, the results are often not generalizable [59], and the approaches perform poorly in real-world settings [121, 122]. As a consequence, they are rarely applied in practice [92, 160], with notable exceptions, though [151].

Addressing the notorious issues of defect prediction of our partners in industry (in particular, Munich Re and LV 1871), we strive for an approach that is viable in practice and matched the needs of our industry partners: a *prioritization* of *test gaps* by their *risk*. As defined in Chapter 2.2.4.1, a *test gap* is a method, function, or module that has been modified during a specific period of time (e. g., start of last development phase or iteration) and has not been executed in its most recent version during testing (e. g., automated unit test or manual acceptance test). Intuitively, defects are introduced by code changes, and defects cannot be detected if they were not tested. In this vein, the literature suggests that modified code tends to be more defect-prone [27, 84, 115, 140]. For example, Eder et al. found in an industrial case study that (1) despite a structured testing process, approximately half of the changes went into production untested, and (2) that untested changes contained up to five times more defects than other parts of the system. This clearly emphasizes the value of test gap analysis in the testing process. For this reason, test gaps are taken into consideration by test management to decide whether testing is completed.[1]

Problem Statement The number of test gaps that need to be investigated by test management and quality assurance depends on many parameters, especially on the number of code changes and the depth of testing. In practice, when test management conducts a test plan or and assesses test gaps as part of test-end criteria evaluation, there are typically dozens, hundreds, or even thousands of test gaps [77] that have not been covered by any test run. The risk of test gaps may vary greatly; for instance, test gaps involving logic modifications or data manipulations might be disguised among less critical changes such as refactorings [134]. Thus, obtaining an overview about test gaps and their risk requires significant effort, and the results may be subjective. Furthermore, in the context of large industrial software systems with changing development teams, it can be hard for individuals to have sufficient knowledge about the system to reliably assess the criticality of changes in the entire code base. Clearly, an *automatic prioritization of test gaps* would be most helpful to reduce the time necessary for the manual inspection and the risk of missing critical ones, which is also confirmed by our industrial partners. To ease adoption, we present guidelines for practitioners to implement a risk-based prioritization of test gaps in their testing process.

Research Gap There is a lot of research in the field of defect prediction, which aims for the identification of defective code [79, 92, 98, 172]. For our industrial research setting, unfortunately, the costs of applying state-of-the-art approaches outweigh the potential benefits—a notorious challenge for the application of defect prediction in practice [122, 160]. Moreover, state-of-the-art approaches typically do not consider prior testing efforts that are focussed by test gap analysis [70, 151, 172]. Another related field, *test case prioritization*, aims at finding failures as quickly as possible by prioritizing tests by their failure revelation probability [83]. Test case prioritization addresses a different problem, though, since it orders test cases, whereas test gap analysis reveals code which has not been tested by existing test cases. In particular, our industrial partners aim for a more general form of risk mitigation: While the probability of introducing a defect is a key risk factor [80, 104], the potential damage caused by a defect is an additional risk factor that needs to be taken into account. That is, core functionality needs to be tested particularly well because potential defects can cause major damage, such as degradation of core business processes—even though the probability of introducing a defect may be comparatively low. Since there is only little related work on prioritization or risk estimation of test gaps in industrial practice, it is still unclear what makes a test gap more risky than others. Hence, we seek to evaluate the feasibility of a simple prioritization approach that fits the setting and requirements of our industry partners and suggest improvements for future deployment.

1 There are several test gap analysis tools available, for example, Teamscale [77, 137] and Sealights [2].

Approach Based on key risk criteria identified from industrial developer experience and the defect prediction literature, we propose our *score-based* approach, called TestGapRadar, to automatically derive a risk-based prioritization of a set of test gaps. This way, a large list of open test gaps can be sorted and, for example, test management can analyze the riskiest test gaps first. To evaluate our approach, we conducted a multimethod study: We compare the prioritization results of our approach with manual test gap risk assessments that have been made by experts in eight real-world industrial projects. For this purpose, we use historic real-world risk assessments from Munich Re and LV 1871, two large companies within the financial domain, where quality assurance experts continuously assessed eight well-established and maintained software systems. We conducted semi-structured interviews with these experts to better understand deviations between our automatic ranking and their manual assessment. Overall, we found that our approach yields a test gap ranking that is shown to be correlated with risk (i. e., higher rank corresponds to more risk) and that the approach can achieve human (domain expert) level performance. Interestingly, quality engineers reported that insights from TestGapRadar allowed them to recognize where their prior assessment missed risks, especially for central code (i. e., code that is central in the program's dependency structure [55, 147, 149]). The quality engineers of our industrial partners acknowledged the significance and relevance of our test gap risk criteria and underline that TestGapRadar would help them in their daily work to identify risky test gaps more efficiently.

Contributions In summary, the contributions of the chapter are the following:

- *Automated Prioritization of Test Gaps.* We introduce TestGapRadar, an automated score-based approach that prioritizes test gaps by estimated risk supporting industrial development teams—including test management and quality assurance roles—in gaining a quick overview about the riskiest gaps.
- *Empirical Study.* We conducted a field study of TestGapRadar on thirty-one historical test gap reviews of open test gaps for eight industrial software systems providing insights into the applicability of risk criteria and process in our industrial setting.
- *Quality Assurance Expert Survey.* We conducted eight semi-structured interviews with six quality assurance experts that authored the test gap reviews of our industrial partners, showing that the automatic ranking of TestGapRadar is on par with expert rankings, and in some cases, even outperforms the expert ranking.

5.1 Related Work

There is only little related work that is specific to test gap prioritization. In addition, we discuss related work from the broader field of software defect prediction.

5.1.1 Test Gap Prioritization

We introduced test gap prioritization formally in Chapter 2.2.4.2. There are a couple of studies addressing the problem to prioritize test gaps, which we outline and discuss in the following.

Sailer [139] conducted a structured online interview to shed light on developers' criteria for assessing the risk of test gaps. In their study, they randomly selected a subset of test gaps from a six-week interval of the developers' application and asked the developers to assess and reason about the risk of test gaps. These rationales provide valuable insights for our work into what makes a test gap appear risky. The most-mentioned reasons are *change complexity*, *centrality* for high risk, and *refactoring* as rationale for lower risk. In contrast to Sailer who evaluates their approach on a single software system, we investigate the transferability of risk criteria and their applicability for test gap prioritization to other large and independent software systems from our industry partners.

Brandt et al. [13] used a fuzzer to generate partial tests and investigated whether developers from Mozilla would extend those to functional tests. They found that developers do not consider all test gaps test-worthy. To address this, they implemented a filter function to only generate partial tests for relevant test gaps. The filter function excludes test gaps that are single-line or are early-return. They found that developers consider test gaps irrelevant if the tested code is unlikely to be reached, deemed bug-free, or already covered by other tests. By means of the code centrality criterion, we also prioritize code down that is unlikely to be reached, which is similar to Brandt et al.'s approach. They focus on helping developers close test gaps, while we focus on helping test managers and quality assurance teams identify the most important ones. Their filter function employs a simple prioritization strategy. We present a more comprehensive approach for prioritizing test gaps, focusing on different roles in the software development process.

Ivankovic et al. [72] introduce the concept of productive coverage, a measure aimed at enhancing the actionability of code coverage information by identifying untested yet test-worthy code. This includes code that is unique within the codebase, not similarly tested elsewhere, and frequently executed in production. Their results highlight that developers consider the feedback from productive coverage beneficial; specifically, the visualization of productive coverage during code reviews has a greater impact on code coverage than a traditional line coverage visualization. By identifying the most relevant untested code changes within a changeset, their approach aligns with our goal of identifying most risky changes. But their focus is much more narrow, since they investigate comparably small code changes within a single changeset, while we focus on larger untested code regions, at least entirely untested methods, and analysing a longer test cycle, for example, an entire sprint. While their approach is designed Google's extensive monolithic code base, our model used a broader set of metrics and is suited for wide industrial application.

5.1.2 Software Defect Prediction

Test gaps are known to have a higher probability of defect than unchanged code [27], so, the field of software defect prediction [79, 92, 172] (see also Sec. 2.2.4.3) is related to our work, with notable differences, though. While a test gap could imply a defect, a defect is not necessarily a test gap. For our prioritization of test gaps, we are focusing on risk, namely, magnitude of impact and probability of event. In contrast, defects may be classified by severity levels [102], but these do not match test gap risk.

The respective models of defect prediction approaches are based on manifold metrics, for instance, source code complexity [52], code smells [117], history of changes [105], commit messages [69] and defects [85], organizational [110], process [21], or developer-centered metrics [25], ticket information [153, 154], or static analysis results [156]. While our prioritization approach relies on some of these metrics, the focus of our work is different since our industry partners align their testing process with untested code changes, for which we aim at a risk-based prioritization.

Like with test gap analysis, it is an established best practice to use function-level defect prediction, which shows better performance than relying on coarse-grained units such as files or modules [40, 122]: Despite a great number of studies in this area, function-level defect prediction is still unsolved as it provides low precision for cross-project classifiers and, when evaluated under realistic circumstances, existing approaches do not significantly outperform a random classifier [121, 122]. There are also more fine-grained approaches for defect prediction, for example, on the line-level [130, 167]. These cutting-edge approaches are still experimental and therefore—from our industrial partners' perspective—not mature enough for implementation in real-world development processes, yet. As of now, our industrial partners need more actionable approaches such as test gap analysis and prioritization of test gaps to direct their testing efforts.

5.2 TestGapRadar: A Score-based Approach

Test gap analysis yields an unsorted set of untested modified code units (functions, in our case). The goal of TestGapRadar is to rank this set of test gaps by their estimated risk. In what follows, we explain the selection criteria for the metrics used for ranking test gaps. Furthermore, we provide details on the metrics and their calculation and insights into the normalization of metric values. Finally, we describe how a risk score is computed, which is used to rank test gaps amongst each other. In this section, we focus on our approach in a general form; implementation details like weights and the choice of parameters that we used for our evaluation are outlined in Section 5.3.4.

5.2.1 Selection Criteria for Metrics

The basis for ranking is a risk score, which is computed for each test gap from a combination of metrics (see also Sec. 5.2.2). To simplify the setup for our industrial study subjects (see also Sec. 5.3.2) and allow for comparison of results, we aim for a lightweight, uniform approach for all study subjects. To this end, we are interested in to which extent a simple approach like ours is able to help practitioners in identifying risky test gaps. We selected product and process metrics that were also used by related work (see also Sec. 5.1) and which were feasible to obtain for all of our study subjects. We had to decide against criteria which did not meet the requirements of our industry partners. Specifically, a broad spectrum of technologies and processes, and diverse set of social and legal requirements needed to be fulfilled. For example, ABAP—a programming language used by several of our industry partners' SAP systems—comes with the limitation that there are no commit messages available as they are known from popular version control systems such as git. Furthermore, defect information is stored heterogeneously, that is, in different bug tracking systems using different bug reporting schemes, impeding its structured analysis. Lastly, developer-centered metrics might not always meet the compliance requirements of our industry partners, so they could not be taken into consideration for our work. For instance, the separation of responsibilities between internal and external employees is mandated by European regulation [38].

5.2.2 Overview of Selected Metrics

Table 5.1 provides an overview of metrics that TestGapRadar uses to estimate test gap risk. These product and process metrics were available at our industrial partners, while other data could not be obtained. They include different risk factors, that is, code criticality and complexity and static code analysis results. In the following, we discuss these risk factors, the corresponding metrics, and their computation.

Table 5.1: Metrics used by TestGapRadar to estimate test gap risk

Risk Factor	Metric (short)
Code Criticality	Code centrality (CEN)
	Changed functions (CHF)
Complexity	Length of reference function (LEN)
	Changed lines of code (CLI)
	Complexity of reference function (COM)
	Complexity change (COC)
Static Code Analysis Results	Added normal findings (ANF)
	Unresolved normal findings (UNF)
	Removed normal findings (RNF)
	Added critical findings (ACF)
	Unresolved critical findings (UCF)
	Removed critical findings (RCF)

5.2.2.1 *Risk Factor: Code Criticality*

We implement two metrics for code criticality: code centrality and changed files.

Code Centrality Sailer [139] found in their study that one of the most often mentioned reasons for critical test gaps, that is, gaps that need to be closed by testing, is code centrality. This matches our notion of test gap risk since defects in central code (e. g., core functionality) potentially cause great harm, and therefore test gaps in central code are considered more risky. For the purpose of test gap prioritization, TestGapRadar includes a metric for the *centrality of the function*. To compute the centrality of the function at time t, we rely on static analysis of dependencies between sources, as suggested in the literature [55, 147, 149]. We apply the PageRank algorithm [116] on the dependency graph of a software system to rank the nodes by their relevance (i. e., centrality). The dependency graph is traversed by either following a link or randomly jumping to another node. We used the random jump probability of 0.0001, as suggested by Steidl et al. [149]. Additionally, we take inverted edges into account to reflect that a function may not only be important if many other functions depend on it, but also if it depends on many important functions. For this work, we set the weight of an inverted edge to $\frac{1}{3}$ as compromise between the suggested values of $\frac{1}{2}$ and $\frac{1}{4}$ [147].

Changed Functions In the context of our industrial partners, large change sets, which affect many files, often occur in system- or component-wide refactorings of the source code, which do not contain functional changes. These refactorings introduce less defects than smaller change sets and thus, the defect density of large change sets is typically smaller than for small change sets. This is in line with the literature in which change sets with many changed files have been found to introduce less defects [105]. Since we are working on the more fine-granular level function level, we count the number of *changed functions* per change set. We decided against using more complex metrics as most of these have been shown to have a high correlation with lines of code [48, 108].

We use a simple heuristic to implement this metric. The idea is, the more functions are modified within a change set (e. g., issue, ticket, change request), the more likely it is a less risky change, for example, a refactoring. For the change set that contains the test gap, we calculate the function churn c_i, that is, the number of modified functions. The metric grows, damped by the power function, up to a function churn threshold c_t. The metric is bound by 1 and calculated by

$$v_{\text{ref}} = \min\left(\left(\frac{c_i}{c_t}\right)^4, 1\right)$$

For the size of the change set, we define a custom normalization function to model the influence on test gap risk. We do not expect a linear correlation of size of the change set and test gap risk. For instance, we consider the influence of a medium and a big change set on the test gap risk as similar, but the difference to a small change set is notable. Note that, for our multimethod study (see Sec. 5.3), we manually validated that this heuristic has assigned high changed functions metric values only to test gaps arising from a refactoring activity.

5.2.2.2 *Risk Factor: Complexity*

We implement four metrics to model the risk factor complexity: *length of reference function* and *changed lines of code* serve as metrics for code complexity, while test complexity is measured by *complexity of reference function* and *complexity change*. The *reference function* is the version of the function before it became a test gap, that is, either the version at the baseline b or, if the function has been tested after the baseline, the last-tested version. If the function has been added after the baseline, the reference state is the empty function $\varnothing$. We call the function at version t the *end state* of the function.

Code Complexity In accordance with several defect prediction approaches [92], we use different metrics for the complexity. The first two metrics refer to code complexity: *length of the reference function* and the *number of changed lines* in the function between reference and end state. We hypothesize that long and complex functions are harder to understand and change and, hence, more defect-prone. In addition, long and complex changes are more defect-prone than small and simple ones [64]. The length of the reference function is the number of source lines of code, that is, excluding comments.

There are multiple ways of computing a metric value for number of changed lines. Code changes are typically displayed as unified diff, that is, the added and deleted lines between both versions. A change to one line is represented as a combination of one added and one deleted line. The same, however, is true for a change that deleted one line and added another, that is, two changed lines. Prior work used the sum of deleted and added lines as value for this metric [85]. In contrast, we doubled the weight of added lines to take our industrial partner's experience into account, which shows that adding lines is more defect-prone then deleting lines (see also Sec. 5.3.4).

Test Complexity To obtain an indication of how many tests might be needed to test a function [162], we use *cyclomatic complexity* as defined by McCabe [100]. The *complexity change* is the difference in cyclomatic complexity of the function between end state and reference state. In particular, this means that the value can be negative if the change reduced the function's complexity.

5.2.2.3 *Risk Factor: Static Code Analysis Results*

We use the number of static code analysis results (i. e., findings) as a proxy to gauge the diligence with which code changes were made. Our industrial partners rely on Teamscale [54] for static code analysis. Teamscale differentiates between new findings and unresolved findings in modified code, both of which are expected to be fixed in the contexts of our industrial partners who have a quality control process in place [148]. Findings are categorized into critical quality deficits, such as bug patterns, and normal quality deficits, such as incomplete documentation. We distinguish between six different finding metrics (the third character of metric abbreviations from this risk factor in Table 5.1 is `F`), that is, the cross product of two severity levels and three finding states. First, we distinguish between the two severity levels critical (the first character is `C`) and normal (`N`), because we consider critical findings to be more risky than normal ones. Second, we separate the finding states added (the second character is `A`), unresolved findings in modified code (`U`), and removed findings (`R`). Hasty code changes that have not been carefully reviewed (e. g., by a static analysis tool) can introduce new defects, increasing the risk of a test gap. This is the case for code changes that add *new findings*, either critical or normal ones, or that *do not remove existing findings*, where the latter ones are expected by our industry partners to be less risky because their quality control process ensures that unresolved findings are less relevant. In contrast, changes that *remove findings* improve code quality and therefore may be less risky. For each test gap, we count the number of findings that were affected by the code changes in that test gap.

5.2.3 Normalization of Metric Values

The selected metrics have different value ranges. To balance their influence on a score-based test gap prioritization, it is necessary to normalize them. We choose the value range of $[0, 1]$ for all metrics except COC. COC is capable of taking negative values and, as such, is mapped to the interval $[-1, 1]$. For each metric, the set of values S is normalized with respect to its maximum and minimum value. The normalization aims at a relative prioritization of test gaps inside one set. However, it is not possible to compare the metric values between different sets of test gaps, as their normalization is based on different maximum and minimum values. In particular, this means that TestGapRadar does not determine the risk of single test gaps. Instead, we aim for an approach that *ranks a set of test gaps based on their estimated risk*.

5.2.4 Computation of Risk Score

In the final step, the normalized values of all metrics are combined into one risk score for every test gap. This is used as basis for prioritization. Every test gap m has a set of normalized metric values V_m. The set W contains the corresponding metric weights (see also Sec. 5.3.4 for details on W we used in our multimethod study). Each metric has exactly one weight, which is the same for all test gaps. Thus, with the number of metrics $k = |W| = |V_m|$ the risk score r_m for a test gap m is

$$r_m = \sum_{i=1}^{k} V_m[i] \cdot W[i].$$

5.3 Multimethod Study

We conduct a multimethod study to evaluate the practical applicability of TestGapRadar, our score-based approach for risk-based prioritization of test gaps presented in Section 5.2, in an industrial setting. Initially, in a field study[2], we compare the risk estimations with test gap reviews of eight software systems across two industrial partners, and we use this data set to compare our approach with a random ranking strategy as baseline. Subsequently, through semi-structured interviews, we discuss our approach with the industrial quality engineers who were involved in the test gap reviews. We follow the guidelines of Jedlitschka et al. [73] to report on our research.

5.3.1 Research Questions

RQ$_1$: How does TestGapRadar perform as compared to risk assessments of quality assurance experts?
In RQ$_1$, we aim at comparing the risk assessments of TestGapRadar with historical risk as-

2 Following Stol and Fitzgerald [150], a *field study* "refers to any research conducted in a specific, real-world setting to study a specific software engineering phenomenon".

sessments provided by two industry partners in the form of test gap reviews. In these, quality engineers analyze open test gaps for risky gaps and report them in their review. We analyze whether test gaps, which were labelled as risky by quality engineers, are highly ranked by TestGapRadar.
RQ_2: Which metrics of the risk score are most important and are the weights robust? Our approach calculates a risk score that is influenced by numerous metrics. We study the individual metric importances to learn which of the metrics have the highest impact on detecting test gap risks. For this purpose, we investigate which metrics of TestGapRadar are decisive to identify risky test gaps from the historical risk assessments. To shed light on the robustness and reliability of TestGapRadar, we perform a sensitivity analysis and a scenario analysis.
RQ_3: How much better is TestGapRadar compared to a random ranking strategy? We compare our approach to a random ranking strategy as baseline, to validate whether a sophisticated approach like ours pays off by better test gap prioritization results. That is, test gaps labelled as risky in test gap reviews are assigned a higher rank, with the ranking indicating a higher estimated level of risk.
RQ_4: Do quality engineers find that the test gap prioritization process can support them in their day-to-day work and if so, how? We conducted semi-structured interviews with the authors of the original test gap reviews, that is, six professional quality engineers of our industry partners, to discuss the practical value of our work. First, we investigate whether they agree with the metrics used for the risk-based prioritization of test gaps. Second, we explore the reasons and practical implications behind deviations observed in RQ_1. Third, we query whether and in which ways our approach could support them in their day-to-day work.

5.3.2 Industrial Study Subjects

For the purpose of our evaluation, we have selected eight industrial study subjects from our industrial partners. An overview of all study subjects of our multimethod study is given in Table 5.2; the provided contextual data meets the criteria by Hall et al. [59]. All study subjects are industrial[3], closed-source systems which have been in successful use for many years and are still actively developed and maintained. The subjects are internally used software systems implementing core business processes or products, and are of mediocre (100 K LOC) to large size (1,900 K LOC). Their implementation relies on different technologies, all of which are supported by our language-agnostic approach. For all study subjects, a well-established issue tracking and testing process is in place. Some subjects adopt automated testing in a CI environment, others focus on manual testing in dedicated testing environments, and some adopt both approaches. All subjects stem from two large, independent players in the finance and insurance domain from Germany, which is strictly regulated by the European Union [39]:

Munich Re[4] is one of the world's leading providers of reinsurance, primary insurance and insurance related risk solutions. It has about 43,000 employees, and a revenue of more than 52.9 billion Euro. Conscious of the great responsibility for software quality, they have a stan-

3 The names of the individual software systems have been anonymized on request of the providing industry partners.
4 The data provided is from 2024

dardized development process, which includes test gap analysis, but without prioritization of test gaps. A dedicated team is reviewing code and test work of all software systems in the portfolio manually, resulting in the monthly assessment reporting. For our study, we used the test gap review data from five systems within Munich Re (no. 1–5).

LV 1871 is a German specialist for life and pension insurance. It has ca. 500 employees, and generates 7 billion Euro in revenue. Emphasizing code quality, the company works with an external team of quality engineers for code retrospectives and test gap analysis. For this study, we used test gap reviews from three of their software systems (no. 6–8).

The development processes of both industry partners include quality control metrics, including external, manual reviews of test gap analysis results (see also Popeea-Simeth et al. [129]). In fact, we chose the systems of our industrial partners as study subjects because external quality assurance experts conduct handcrafted *test gap reviews* that assess test gaps based on risk. Our industrial partners implement test gap reviews for many years already, so that we can use this valuable historic information as reference data in our study. For our 8 subject systems, we use a series of up to 7 test gap reviews from 2023 (see also Table 5.2), pointing to 181 risky test gaps (out of a total of 2,039 test gaps). We evaluate whether test gaps that were identified as risky are ranked high by our score-based approach. For transparency, we note that the external quality assurance experts work for the same company as some authors. No study author was involved as interviewee in our semi-structured interviews, though.

Table 5.2: Overview of study subjects

					# Test Gaps	
Company	Subject	LOC	Lang.	# Reviews	Risky	Total
Munich Re	1	1,600 K	C#	5	59	77
	2	140 K	C#	7	29	161
	3	370 K	ABAP	3	21	29
	4	560 K	ABAP	4	9	32
	5	1,900 K	ABAP	4	26	53
LV 1871	6	310 K	Java	3	5	622
	7	100 K	Java	3	28	1,052
	8	150 K	Java	2	4	13

5.3.3 Study Design and Operationalization

We apply multiple methods to answer our research questions of Section 5.3.1. Figure 5.1 provides an overview of the study data we used to answer the research questions. In the following, we provide an overview about our study data, that is, historical test gap reviews, and explicate the study design and operationalization for our research questions.

Historical Test Gap Reviews We test the performance of TestGapRadar on study subjects from our industrial partners. We use existing test gap reviews T as reference data for

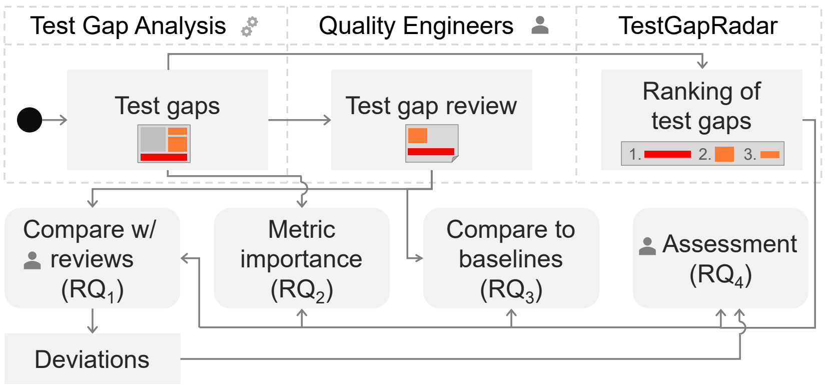
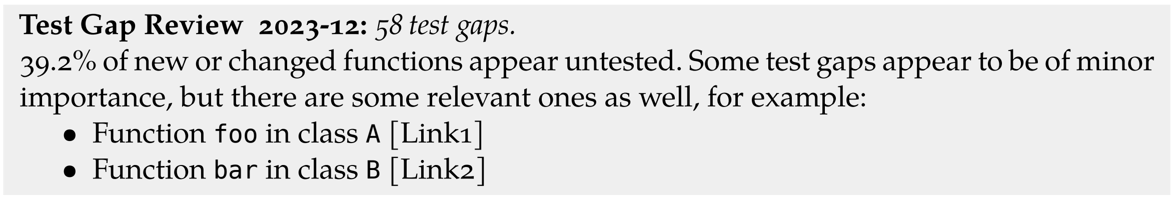


Figure 5.1: Study data used to answer our research questions



Test Gap Review 2023-12: *58 test gaps.*
39.2% of new or changed functions appear untested. Some test gaps appear to be of minor importance, but there are some relevant ones as well, for example:

- Function `foo` in class `A` [Link1]
- Function `bar` in class `B` [Link2]

Figure 5.2: Example for a test gap review of testing endeavor in December 2023 (highlighting indicates existence of risky gaps). The two anonymized functions `foo` and `bar` are labelled as *risky* by the quality engineers that authored the review.

our study (see Sec. 5.3.2). A test gap review refers to all open test gaps since the review baseline b (cf. Sec. 5.1.1). Typically, it spans over a period of 1–3 months, depending on the amount of development activity. In all subject systems, test gap reviews are conducted regularly by an external party, to ensure neutrality. A review states the number of open test gaps and the proportion of open vs. closed test gaps. In addition, it lists test gaps that the reviewer considers *risky* and, consequently, which they recommend being closed.

Definition "Riskiness of test gaps"
A test gap is classified as *risky* if it has been identified as pertinent in a test gap review. Pertinent test gaps present a notable risk of introducing defects and are advised by quality engineers to be resolved by the relevant development and test teams. Conversely, a test gap is categorized as *less risky* if it has not been mentioned in a test gap review or if it has been referenced in a review where the quality engineer indicated a low risk of defect introduction.

An exemplary test gap review of the testing endeavor in December 2023 from one of our study subjects is given in Figure 5.2. Note that test gap reviews point to test gaps that are considered (most) risky. So, this list may not include all test gaps, and less risky test gaps are not mentioned.

RQ$_1$: Comparison with Manual Assessments To answer RQ$_1$, we adopt two methods. First, we conduct a correlation analysis between the experts' test gap risk assessments from test gap reviews T and TestGapRadar's risk scores R. We examine the correlation using Kendall's τ and the associated p value [82] (computed using SciPy [158]). Kendall's τ is defined for two rankings x and y with P concordant pairs, Q discordant pairs, and T ties only in x and U ties only in y:

$$\tau = \frac{P - Q}{\sqrt{(P + Q + T) \cdot (P + Q + U)}}$$

Second, we investigate the risk score rankings of risky and less risky test gaps. For this purpose, we calculate an agreement value $v \in [0, 1]$, where values closer to 0 stand for a better ranking agreement (that is, risky gaps are ranked higher than less risky gaps). As basis for v, we sum up the ranks $\text{rank}(t)$ of risky test gaps $t \in G_r \subset G$, where G are all test gaps of a test gap review, and divide them by the number of test gaps $|G|$ and the number of risky test gaps $|G_r|$ to obtain v':

$$v' = \frac{\sum\limits_{t \in G_r} \text{rank}(t)}{|G| \cdot |G_r|}$$

The best possible minimum $\min(v')$ is:

$$\min(v') = \frac{\sum\limits_{i=1}^{|G_r|} i}{|G| \cdot |G_r|}$$

All risky test gaps are ranked above all other test gaps. For example, for $|G_r| = 3$ and $|G| = 7$, we obtain $\min(v') = (1/7 + 2/7 + 3/7)/3 = 0.29$

The worst possible agreement value $\max(v')$ is:

$$\max(v') = \frac{\sum\limits_{i=|G|-|G_r|+1}^{|G|} i}{|G_r| \cdot |G|}$$

All risky test gaps are ranked below all other test gaps. For example, for $|G_r| = 3$ and $|G| = 7$, we obtain $\max(v') = (5/7 + 6/7 + 7/7)/3 = 0.86$.

To obtain the agreement value v, v' is min–max scaled to $v \in [0, 1]$ (with a mean of 0.5):

$$v = \frac{v' - \min(v')}{\max(v') - \min(v')}$$

We visualize the agreement values v for all test gap reviews by means of a kernel density plot, and we explicate median ranks of risky and less risky test gaps. For the purpose of illustration, we report a false-low rate of test gap review rankings by considering the fraction of test gap reviews exhibiting a poor agreement value ($v \geq 0.5$) against all test gap reviews.

To investigate the ranking performance on the level of individual test gaps, we plot the rankings of test gaps labeled as risky and less risky as a kernel density plot. To allow comparison between rankings of different test gap reviews, we use relative ranks $\bar{R} \in [0, 1]$, where the highest rank is mapped to 0, and the lowest rank corresponds to 1. To provide the reader with an intuitive means to compare the overall ranking performance between risky and less risky test gaps, we depict their median values. Lastly, we use the Mann-Whitney U test to test the null hypothesis that the ranks of test gaps deemed risky in the test gap reviews determined by their risk score ($\in R$) and the same ranks of less risky test gaps stem from the same distribution.

RQ*₂*: Metric Importance We implement a multifactorial ANOVA (analysis of variances) to answer RQ$_2$. That is, we employ ANOVA to assess the influence of the independent variables (i. e., the score metrics) on the dependent variable (the test gap risk assessments from T). The null hypothesis is that there is no relation between an independent variable and the dependent variable. If the p value is below $\alpha = 0.05$, we reject the null hypothesis and assume that there is a relationship between the independent and the dependent variable. To this end, we report the corresponding F and p values. In a post-hoc analysis employing linear regression, we investigate the strength of these relationships. In particular, we report the p values, the coefficient values, and R^2 for the regression models. Additionally, we report results of the correlation analysis in the form of a correlation heatmap.

We report global sensitivity indices as suggested by SoboÍ [146], since they allow for decomposition of ranking contributions from individual parameters [12]. We implement the sensitivity analysis using SALib [68] and use $N \times (2D+2)$ model evaluations, where $N = 2,039$ is the number of samples and $D = 12$ is the number of metrics. The first-order sensitivity indices S_{1_i} represent the effect of each metric on the risk score variance when all other factors remain constant. The total-order sensitivity indices S_i^{tot} capture both the individual effects and the interactions with other metrics. For the scenario analysis, we iteratively vary weights of parameters with the highest and lowest influence on the risk score and report the scenario performance that we measure by the median relative test gap ranking of risky test gaps $\tilde{R} \in [0,1]$ (see also RQ$_1$).

RQ$_3$: Comparison to Random Baseline To answer RQ$_3$, we compare the ranking performance of our score-based approach TestGapRadar with a *random strategy*: The random strategy simulates 1,000 test gap analysis sessions for all test gap reviews without any indication about the risk available. That is, we assign all test gaps from the test gap reviews a random risk score in $[0,1]$ and rank them by this random score. From the 1,000 simulations, we calculate the average rank of risky gaps and the ranking variance. To assess the ranking performance, we consider the median relative ranks $\tilde{R} \in [0,1]$ of test gaps deemed risky and their variance var(R). Additionally, we use a Mann-Whitney U test to test the null hypothesis that test gaps deemed risky in the test gap reviews ($\in T$), ranked by the risk score ($\in R$) of TestGapRadar and the ranks determined by the baseline stem from the same distribution.

RQ$_4$: Expert Assessment We answer RQ$_4$ based on semi-structured interviews with the six industrial quality engineers who conducted the original test gap reviews used in the earlier research questions. Details of the interview questions can be found on our supplementary Web site (see also Sec. A.3). All interviewees are experts in the field of software quality. Their professional experience in coding, software testing, and quality consulting activities ranges from four to twenty years. All of them have a Master's degree in software engineering, two of them even have a PhD in software engineering. They are experts in test gap analysis tools and have been using them in their daily work for years.

For each study subject, we chose the test gap review with the *lowest* agreement between the review and the risk-score-based ordering of test gaps for our interview. We ensured that the set of chosen test gaps is as diverse regarding associated change types and risks as possible. To foster a lively conversation allowing for deep dives into the data, we conducted for each study subject a joint semi-structured interview with author and reviewer of the test gap review between April and June 2024. Our primary goals were to shed light on the experts' reasoning behind risk assessments, to gain insights into reasons for deviations between R and D, and to collect feedback on practical applicability of our approach.

Each semi-structured interview consisted of three parts: First, we asked the participants to construct a pairwise comparison of three to four test gaps of the assessment based on their subjective risk. For this task we selected test gaps where the professional assessment did not match the automatic ranking. Second, we discussed our test gap prioritization, in general, and specifically the TestGapRadar's generated ranking for the three to four test gaps. Third, we ask about the expert's background and their feedback on our research. The interview sessions took ca. 30 minutes, each.

We applied qualitative content analysis methods [99] to systematically analyze the semi-structured interview data. This involved employing the QCAmap tool by Mayring et al. [99] for qualitative content analysis and applying inductive techniques for data categorization.

5.3.4 Implementation and Calibration

We have implemented TestGapRadar and used a data-driven approach to tune the weights of the risk score function (see also Sec. 5.2.4). For this, we used preliminary (training) data of former test gap reviews (pre-2023) from our industrial partner Munich Re that we obtained in the initiation phase of our research effort. In this pilot study, we fine-tuned the weights of our approach so that risky test gaps from pre-2023 test gap reviews are ranked highly. We considered our sample size too small for automated fine-tuning mechanisms, so we used manual fine-tuning instead. Manual fine-tuning helps us provide rationales for the weights, which increases trust in the tool and makes the prioritization results more understandable, which is key for broad adoption in practice. To mitigate the risks of subjectivity and potential biases in weight selection, we investigate the robustness and reliability of weights as part of RQ2, and provide guidelines for practitioners to integrate TestGapRadar into their testing process (see Sec. 5.4). Generally, we applied the risk score function consistently for all study subjects. We avoided overfitting by using separate training and test data sets.

Initially, all weights were assigned a default value of 1. The rationale behind weight selection, ensuring explainability of the prioritization, includes:

1. setting positive weights as defaults, while indicators of improved code quality (e. g., readability) are allocated negative weights (i. e., CHF, RCF, RNF);
2. assigning higher weights to factors frequently cited by developers as critical in related studies on test gap prioritization [139] receive a higher weight (i. e., CEN, CLI, COC);
3. prioritizing code change metrics over reference function metrics motivated by the notion that added complexity signifies greater risk than existing complexity (i. e., CLI, COC);

4. providing greater weight to metrics capturing critical findings than those for normal findings, with new findings deemed riskier than existing ones (i. e., ACF, ANF, UCF, and RCF).

The final weight calibration is outlined in Table 5.3.

Table 5.3: Overview of metric weights used for TestGapRadar in the evaluation

Metric	Short	Weight
Code centrality	CEN	2
Changed functions	CHF	-1
Length of reference function	LEN	1
Changed lines of code	CLI	2
Complexity of reference function	COM	1
Complexity change	COC	2
Added normal findings	ANF	2
Unresolved normal findings	UNF	1
Removed normal findings	RNF	-1
Added critical findings	ACF	4
Unresolved critical findings	UCF	2
Removed critical findings	RCF	-2

The changed functions metric has a parameter, t, which refers to the size of a change set so that it is considered as less risky. In preliminary experiments, we found that $t = 100$ modified functions per change set are suitable to identify refactorings for our study subjects.

5.3.5 Results and Discussion

In what follows, for each research question, we present the results of our multimethod study and discuss them.

5.3.5.1 *RQ₁: Comparison with Manual Assessments*

We use Kendall's τ to investigate whether the test gap assessments from test gap reviews and the normalized risk scores correlate. We find a small [20], negative monotonic correlation between T and R ($\tau = .29$, $n = 2039$, $p < .001$), meaning that risky test gaps from test gap reviews receive higher rankings from TestGapRadar.

Figure 5.3 shows a kernel density plot of the agreement values v from the 31 test gap reviews from our eight industrial study subjects. Note that the distribution is right skewed, that is, most of the test gaps from test gap reviews are correctly ranked. In total, TestGapRadar ranked risky test gaps for 3 out of 31 test gap reviews too low (i. e., "false-low", $v \geq 0.5$), so the false-low rate is below 10%.

Figure 5.4 shows a kernel density plot of the relative ranks $\bar{R}$ of all 2,039 test gaps from the 31 test gap reviews from our eight industrial study subjects. Risky test gaps are ranked higher (i. e., better) by TestGapRadar than less risky test gaps. The median ranking for risky

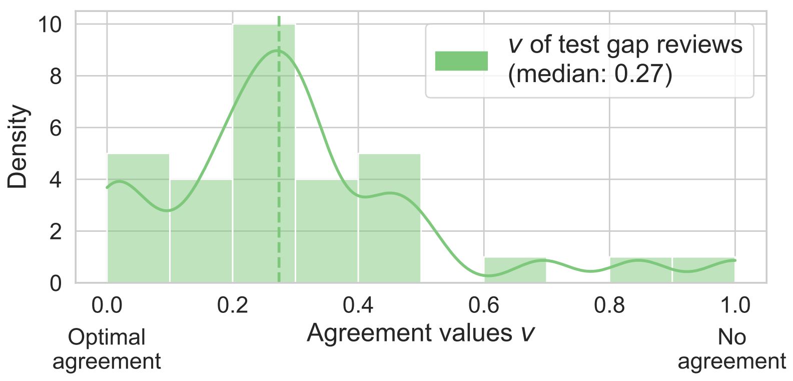


Figure 5.3: Kernel density plot for agreement values (v) from 31 test gap reviews of our eight industrial study subjects

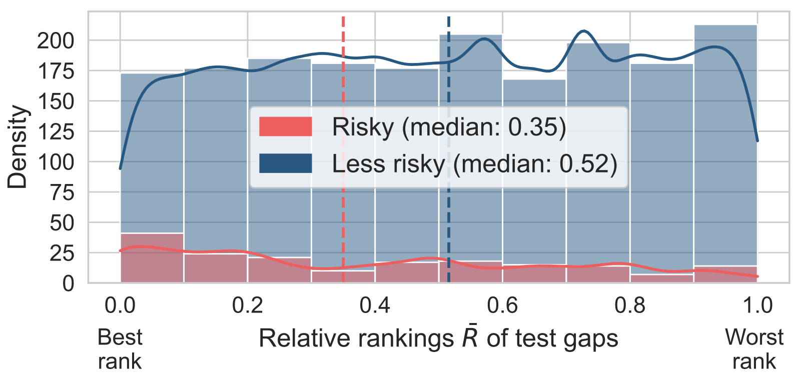


Figure 5.4: Kernel density plot for relative ranks $\bar{R}$ of 2,039 test gaps labelled risky (red) or less risky (blue); from 31 test gap reviews of our eight industrial study subjects

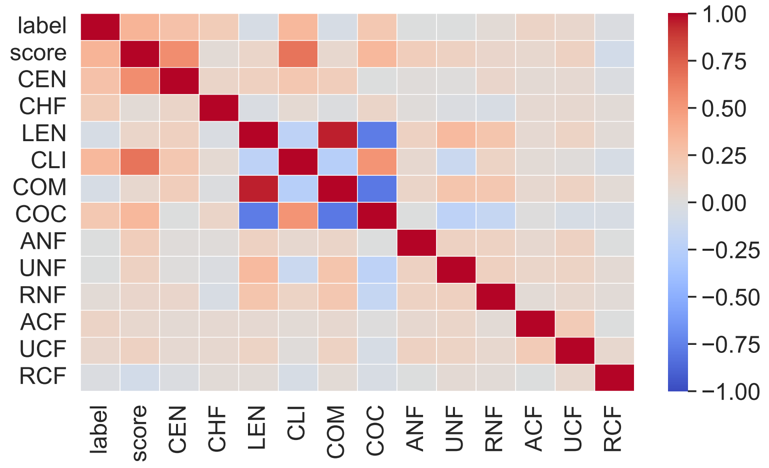


Figure 5.5: Correlation heatmap of the assessment label, the risk score, and the score metrics

test gaps $\tilde{R} = 0.35$, while less risky test gaps are ranked lower (median = 0.52); the average ranking is 0.5. The distributions in the two groups differed significantly (Mann–Whitney U = 127.5, $n_r = 181$, $n_{lr} = 1858$, $p < 0.05$, less).

Overall, the accuracy of the results of our application compared to the expert assessments appears rather good: An average ranking of risky gaps on the 35th percentile and a low false-low rate below 10% shows that risky test gaps are ranked higher by TestGapRadar than less risky test gaps. In the discussion of RQ_4, we explore reasons for deviations and their implications for the practical use of TestGapRadar.

Summary RQ_1. *Our approach achieves a good ranking performance: Risky test gaps are significantly more likely to be ranked higher by TestGapRadar than non-risky test gaps.*

5.3.5.2 *RQ_2: Metric Importance*

Figure 5.5 shows a correlation heatmap among the risk assessment labels ("risky" or "less risky" from the test gap reviews T), the risk score (determined by TestGapRadar), and associated metrics (see also Table 5.1). Noteworthy correlations include a moderate positive link between the risk score and CEN, CLI, and COC. LEN is moderately negatively correlated with COC but strongly correlated with COM. Additionally, COM has a moderate negative correlation with COC.

Table 5.4 shows the results of ANOVA with F statistics and p value for the independent variables (metrics). There are seven variables that have a strong relation ($p < 0.05$) with the assessment label: CEN, LEN, CLI, COM, COC, UNF, and ACF. For these, we ran a post-hoc analysis with linear regression models; Table 5.5 shows the results. The p value for all seven independent variables is well below the significance level $\alpha = 0.05$. This indicates a statistically significant relation between each of the variables and the manual risk assessment of test gap reviews. The largest coefficients have CLI, ACF, and CEN, so they appear to have a strong relationship with the manual risk assessment. Note that the R^2 values are relatively small across all models, suggesting limited explanatory power and the presence of other factors influencing the risk labels. This is expected in complex tasks such as test gap risk estima-

Table 5.4: ANOVA table with F statistics and p value per independent variable

Metric	Short	F	p
Code centrality	CEN	123	.000
Changed functions	CHF	1.95	.163
Length of reference function	LEN	39.7	.000
Changed lines of code	CLI	482	.000
Complexity of reference function	COM	12.0	.000
Complexity change	COC	65.6	.000
Added normal findings	ANF	.68	.408
Unresolved normal findings	UNF	15.0	.000
Removed normal findings	RNF	2.44	.118
Added critical findings	ACF	19.5	.000
Unresolved critical findings	UCF	.01	.936
Removed critical findings	RCF	.14	.706

tion for several reasons. Firstly, the subjectivity in risk label assignment by quality engineers is a factor. Secondly, background knowledge, domain expertise, and system familiarity influence the perceived risk of test gaps, elements not easily generalized or captured by our heuristics. Thirdly, there are diverse goals and risk factors in software testing, spanning functional, technical, economical, legal, and organizational aspects, which go beyond the scope of our work and exceed current software engineering methodologies.

There are three further noteworthy observations from the data. First, the correlation analysis suggests that longer function changes do not add as much complexity as new or small functions, particularly in the context of grown systems (which all of our study subjects are): Changes on existing—potentially grown—functions tend to be small compared to new metrics—added in new functions—that also add new, and on average, more complexity. Second, the complexity of reference function correlates strongly with function length and yields a modest F statistic in the ANOVA. Consequently, a simplification of our model might be to eliminate the COM metric. Third, from the variables that we found most important for a suitable risk score, that is, CLI, ACF and CEN, there is only a weak correlation between CLI and CEN. Hence, they all contribute significantly to the ranking performance of TestGapRadar.

Table 5.5: Post-hoc analysis with linear regression

Metric	Short	p	coef	R^2
Code centrality	CEN	.000	.465	.136
Length of reference function	LEN	.000	.180	.014
Changed lines of code	CLI	.000	.741	.187
Complexity of reference function	COM	.000	.207	.018
Complexity change	COC	.000	.380	.075
Unresolved normal findings	UNF	.000	.153	.004
Added critical findings	ACF	.000	.713	.015

The first-order sensitivity indices of COC and ACF are the highest ($S_{1_i} = 0.28$), which confirms their significant individual influence on the risk score. The five metrics LEN, COM, ANF, RNF, and CHF have the lowest indices ($S_{1_i} \leq 0.02$), indicating that, individually, they have minimal impact on the risk score when all other factors remain constant. Similarly, COC and ACF have the highest total-order sensitivity indices ($S_i^{\text{tot}} = 0.28$). The close similarity to their first-order indices suggests that these parameters have limited interaction effects with other parameters, reaffirming their role as primary contributors to output variance. The five metrics mentioned before also have the lowest total-order sensitivity indices ($S_i^{\text{tot}} \leq 0.02$), showing negligible differences between first-order and total-order indices. This supports the conclusion that their interactions with each other or with dominant metrics are minimal.

We shed light on the robustness and reliability of TestGapRadar by running a scenario analysis. First, we reduce the weights of the most influential metrics COC and ACF by a factor of 2. This results in a median relative rank of risky gaps $\tilde{R} = 0.38$, which indicates a deterioration in the ranking performance of TestGapRadar. Second, doubling the weight of the most influential metrics by a factor of 2 results in $\tilde{R} = 0.3$, which is a substantial improvement. Third, we investigate simplification opportunities for our model by setting the weights of the five least influential metrics (with $S_{1_i} \leq 0.02$) to 0. We consider different scenarios in this case: (1) all five metrics receive a weight of 0, and (2) five other models where each sets the weight another metric to 0. Our results show that the ranking performance remains the same as for the original model when leaving out all five metrics or each metric individually ($\tilde{R} = 0.35$). Only when LEN is left out, the ranking performance of the original model improves ($\tilde{R} = 0.32$). We evaluated further scenarios, for example, with doubled weights of the most influential metrics and leaving out some of the least influential metrics, but those models performed worse ($\tilde{R} > 0.3$) than the model with doubled weights of the most influential metrics. That is, in development contexts similar to our study subjects, a model refinement with slightly adjusted weights can even achieve better prioritization than the approach implemented and calibrated for this study.

Summary RQ2. *Changed lines, complexity change, added critical findings, and code centrality are key metrics in our model to predict test gap risk.*

5.3.5.3 *RQ$_3$: Comparison to Random Baseline*

Table 5.6 shows the median relative ranking of the risky test gaps $\tilde{R}$, the ranking variance var(R), and the results of a U test of our score-based approach and the random ranking strategy. Our null hypothesis H_0 can be rejected (marked with ✗ in the table). The ranking of TestGapRadar outperforms the baseline with regard to the median relative ranking $\tilde{R}$ and shows a lower ranking variance var(R). That is, the score-based approach clearly outperforms the random baseline.

Table 5.6: Baseline comparison with a random baseline

Ranking Strategy	$\tilde{R}$	var(R)	U stat.	p	H_0
TestGapRadar	.3	.05			
Random	.5	.06	235	.00	✗

Summary RQ$_3$. *TestGapRadar outperforms the baseline in ranking risky test gaps.*

5.3.5.4 *RQ$_4$: Expert Assessment*

In our semi-structured interviews, we observed that all six quality engineers (Q_{1-6}) found the metrics we used for test gap prioritization meaningful and representative of test gap risk. $Q_{1,2,5,6}$ saw a special value in the information about code centrality (CEN), since they usually do not have this information at hand when preparing test gap reviews, so TestGapRadar can provide valuable extra information to the experts in test gap reviews. Additionally, Q_{1-5} explicitly agreed on our choice of putting lower weight on test gaps that refer to simple refactorings which we identify by the number of changed functions (CHF). All experts Q_{1-6} underlined the importance of complexity indicators for risk assessments, since code complexity makes it harder for developers to implement code changes correctly, thus requiring thorough testing. Also, $Q_{1,2,5}$ emphasize their commitment on code quality by verifying static analysis results, putting special focus on critical findings.

In two out of eight interviews, the quality engineers deviated from their original sorting of the interviews after learning about the prioritization of TestGapRadar, so the tool prioritization outperformed the original expert sorting. In both cases, the information about code centrality was the decisive factor. For example, Q_1 stated "I have to agree with code centrality of [this method], which looks pretty important to me. In this case, I'd vote for ranking it higher because it is more important than the other gaps". That is, TestGapRadar was able to detect central test gaps that implied risky code changes, for which the experts retrospectively agreed that they would have considered code centrality if they had known about this factor beforehand. Conversely, when the quality engineers did not change their prioritization based on the reasoning of our approach, they justified their stance by citing several factors. These included the perceived higher risk associated with new functions compared to modified functions due to their lack of production history (2 cases). Additionally, they argued that the type of code (e. g., test code or generated code) could mitigate the risks associated with test gaps (2 cases). Furthermore, they expressed concerns about the extensive deletion of logic (1 case) and considered placeholder implementations (function stubs) to be less risky (1 case). In fact, they noted that an automated tool—while clearly helpful to them—can hardly capture all risk factors for test gaps, since risks can arise from other levels than source code, such as usage information, domain knowledge, or project context.

All quality engineers Q_{1-6} underline in the interviews that they see this tool as part of a semi-automated process, which still needs an expert in the loop. In this context, Q_4 praises that it can help to work in a "much more structured way and identify relevant, risky test gaps much more quickly", and as Q_3 articulates, "filtering out irrelevant gaps". Q_2 was quite enthusiastic and stated "overall, the results here were exactly in line with my assessment, especially for the riskier items, which is a very exciting result".

Summary RQ$_4$. *The experts consider TestGapRadar valuable, providing them with additional information such as the centrality of test gaps, enhancing their daily work. Identifying high-risk gaps and filtering out low-risk ones improves their efficiency.*

5.4 Guidelines for Practitioners

High code churn and limited testing resources are omnipresent circumstances in active industrial software development and contribute to large numbers of test gaps. TestGapRadar addresses the problem of identifying the riskiest test gaps among potentially large sets of test gaps by sorting them according to estimated risk. A primary design goal of TestGapRadar was to ease practical adoption, and we outline guidelines for practitioners in this section.

There are some prerequisites to implement our approach. First, the development process should require code changes to be successfully tested (e. g., in the definition of done). Second, test gap analysis needs to be established, that is, source code is under version control (e. g., using git) and all relevant testing environments are profiled.

To adopt our sorting and risk estimation approach, metrics and weights need to be chosen. For optimal ranking performance, we recommend to use all metrics and adapt the weights of ANF and ACF, as shown in Table 5.7 (which summarizes our results from the scenario analysis for RQ_2 in Sec. 5.3.5.2). A detailed configuration is depicted in Table 5.7. Optionally, the weights can be fine-tuned by means of context and domain knowledge.

Table 5.7: Overview of metric weights for which TestGapRadar obtained the best ranking performance in our multimethod study

Metric	Short	Weight
Code centrality	CEN	2
Changed functions	CHF	-1
Length of reference function	LEN	1
Changed lines of code	CLI	2
Complexity of reference function	COM	1
Complexity change	COC	4
Added normal findings	ANF	2
Unresolved normal findings	UNF	1
Removed normal findings	RNF	-1
Added critical findings	ACF	8
Unresolved critical findings	UCF	2
Removed critical findings	RCF	-2

For a successful implementation of the risk-based sorting of test gaps, the testing process needs to be enhanced. First, it is necessary to be able to differentiate between test gaps from finished development and work in progress. For example, a branching scheme could be implemented in the version control system such that stable code can be easily identified. Alternatively, code changes could be mapped to issues and the relevance of test gaps could then be inferred from the mapped ticket state. Second, the *test gap guard* role needs to be established. The test gap guard is responsible for checking test gaps of finished code changes, for example, in a regular interval or in the testing phase before a release. From our experience, this role is taken either by test management, a test lead, a tester, or developers. Our approach comes into play when the test gap guard checks for open test gaps: They sort all test gaps within the timespan $[b, t]$ of their interest by estimated risk. Every test gap from the sorted list is then manually reviewed and risky test gaps need to be closed, typically by adding new test cases. Our approach helps to identify the most risky gaps early, allowing for more time to close them, therefore increasing efficiency of the testing process. When there are too many test gaps to review all of them, effectivity is increased since review activities can be focused on more risky gaps.

5.5 Threats to Validity

In this section, we discuss threats to internal and external validity and explain our mitigation strategies.

5.5.1 Internal Validity

The limited availability of data for the metrics at our study subjects represents a threat to internal validity because further metrics might result in better ranking performance. However, we have selected metrics that are related to well-known risk factors and, based on our experience, are readily collectible in highly regulated industrial projects employing test gap analysis. Consequently, prioritization can be readily incorporated and anchored in the development process. Our multimethod study results demonstrate that the ranking performance is sufficient for practical application.

The selection of parameters for our implementation poses a threat to internal validity, as the weights applied directly impact the risk score and subsequent ranking. We manually calibrated our selection using historical test gap reviews from one industrial partner (refer to Sec. 5.3.4). Our focus on evaluating a straightforward prioritization approach within an industrial setting forced us to conduct a multimethod case study under limited training data availability. Further refinements, particularly weight adjustments, are deferred, offering the potential for improved ranking outcomes.

Imbalance in data threatens internal validity. The data need to contain an appropriate balance between safe and risky test gaps. This is especially important as our approach estimates the relative risk in the respective set. As discussed in Section 5.3.5.2, there is no universal definition of test gap risk. Therefore, there is no objective way to assess the validity of the set in this regard. However, a manual analysis showed that the study subjects contain a wide variety of test gaps, including complex and trivial ones.

5.5.2 External Validity

As true for most software engineering research, the huge diversity of software systems, processes, and teams, threatens the generalizability of our work [144]. All study subjects used in our evaluation are industrial, closed-source systems (which is not the case for most related work). While they implement sophisticated testing processes, there is a tremendous variety of testing in practice. For example, most open-source software projects often implement substantially different testing processes, for which TestGapRadar might produce different results. Nevertheless, with a technologically and process-related diverse set of study subjects from different industry partners, we share meaningful insights into the benefits and limitations of our work in practical use.

5.6 Conclusion

The prevalence of test gaps introducing new defects presents a significant challenge in modern software development projects, for example, for test management and quality assurance, which need to review a large amount of test gaps to allocate testing efforts and assess test completion. In this chapter, we proposed TestGapRadar, an automated approach for prioritizing test gaps based on their individual risk. For the risk estimation, we incorporated fourteen metrics reflecting three major risk factors, that is, code criticality, complexity, and static code analysis results. In a multimethod study, we validated our approach across eight large-scale software systems from two industry partners. Our study is based on an analysis of 31 historical test gap reviews for their systems and semi-structured interviews with the quality engineers who wrote those reviews. Our study showcased the effectiveness of TestGapRadar in ranking risky test gaps significantly higher than less risky test gaps, on average, at the 30th percentile. In a quality assurance expert survey, the external quality engineers of our industry partners underlined the meaningful representation and potential superiority of the automated risk assessment of TestGapRadar over the expert judgments in certain scenarios. Our study's results underscore the significance of test gap risk estimation for facilitating risk-driven prioritization, empowering test management and quality assurance teams to efficiently pinpoint and manage critical test gaps. Our work enables practitioners to implement a risk-focused safety net into their testing process to ensure that no potentially risky code change is released untested. The quality engineers at our industry partners are definitely planning to implement our approach to prioritizing test gaps as part of their quality assurance processes.

5.7 Avenues of Future Research

There are different avenues of future research: Enhancements of our work and practical needs to reduce risk in software testing and development in general. The risk estimation could be enriched by production usage data to filter test gaps which are not used in production and highlight test gaps in heavily used core features. Additionally, socio-technical analyses and metrics [74, 75] could be considered to reflect additional dimensions of risk. Also, natural-language processing of the commit messages that resulted in a test gap could help to estimate the associated risk. Future test gap risk estimation methods could cluster test gaps and aggregate the risk of a set of related test gaps. More sophisticated risk estimation methods, possibly including line-level defect prediction, may help to identify risky test gaps without needing to compare them with other test gaps. Key factors for practical adoption for any sophisticated approach include to make them approachable and understandable. For the adoption of artificial intelligence in the field of defect prediction, explainability is crucial to convince developers of potential problems, motivating them to fix the underlying defect. Furthermore, practitioners could be guided how to close risky test gaps, for example, by generating test cases that close the gaps and finally mitigate their risk.

6 Conclusion and Future Work

Human-in-the-loop software testing processes in industrial contexts frequently suffer from slow test feedback and risks arising from untested code changes. Feedback to developers is especially delayed in the context of manual testing because it can take weeks or even months to execute extensive manual test suites. Risks from untested changes need to be assessed by humans, for instance, to decide whether testing has completed since all relevant risks have been mitigated—while the risk varies greatly between changes. This thesis sought to resolve the aforementioned issues by means of enhancing the feedback provided by human-in-the-loop testing processes in industrial contexts, thereby increasing their efficiency and effectiveness.

In this dissertation, we conducted three different empirical studies on optimization strategies for human-in-the-loop testing processes. Our research methodology incorporates a combination of quantitative and qualitative research techniques, employing a multifaceted approach to analyze human-in-the-loop testing processes from various perspectives. This multi-methodological approach is designed to develop a nuanced understanding of optimization levers and their impact when applied effectively. This research opens different avenues for future research, and guides practitioners in transferring our scientific results into industrial practice.

In what follows, we briefly summarize our contributions. We conclude this thesis by offering insights into future research opportunities to further advance the optimization of software testing.

6.1 Summary of the Contributions

In essence, based on the three empirical studies that we have presented in this thesis, our contribution to optimization of human-in-the-loop testing processes is threefold:

1. **Exploration of Optimization Technique Transferability from Automated to Manual Testing:** Our first contribution explored what optimization techniques from automated testing are applicable in manual testing, how to integrate them in existing processes and infrastructure, and which limitations need to be accepted. We discovered and systematized characteristics of manual testing processes that deviate from automated testing and that hinder or enable optimization of manual testing. For this purpose, we conducted two surveys and queried corresponding test suites, involving, in total, 43 testing professionals and their testing processes from 20 companies. Our results gathered evidence that manual testing is employed extensively, and in many cases without

the intention of full automation, underscoring the need for optimizing manual testing. We identified nine optimization techniques applicable to manual testing, such as intentional under-specification, test case selection and prioritization, and test gap analysis. Our discussion centered around the question under which circumstances they can be implemented in practice, making prerequisites and caveats of the individual optimization techniques transparent. To facilitate the adoption of manual testing optimization in other contexts, we synthesized our findings in an annotated model of manual software testing processes, accompanied by two sets of guidelines for practitioners on selecting suitable optimization techniques (see also Chapters 3 and 4).

2. **Evidence on Optimization Effectiveness for Manual Testing:** Our second contribution builds up on the previous one: having shown that optimization techniques from automated testing can in principle be transferred to manual testing, we investigated the question to which extent such techniques can be applied to solve the issue of long-running manual test suites in practice, and which limitations need to be accepted in practical use. Two industrial case studies on test suites from Munich Re and IVU Traffic Technologies, demonstrated improvements in fault detection probability, test feedback time and test creation efforts by following our guidelines mentioned in the first contribution (see also Chapter 3). We extended the insights from these two case studies substantially in another empirical study with five subjects from, inter alia, Bayerische Versorgungskammer, Dolby, ILP, and Carl Zeiss Microscopy, in which we implemented two optimization techniques—test impact analysis and Pareto testing—for their automated and manual testing processes. The former, test impact analysis, is a common optimization approach combining test case selection and prioritization. The latter, Pareto testing, is an optimization technique which collects test cases up to a certain cost limit based on existing techniques from test case prioritization and minimization. In our second empirical study, we extracted relevant differences between automated and manual testing processes for test optimization, such as differing test activities, bottlenecks, and presence of flaky tests. For the field experiment, we analyzed industry data from more than 43,300 test cases and corresponding test-wise coverage and, in total, 2,622 test failures from the study subjects' test histories. Our results showed that optimized automated test suites detect, on average, 80% of failures while saving 66% of execution time, compared to 81% failure detection and 43% time savings for manual tests. Despite inherent limitations of manual testing, such as less frequent test execution and fewer historical data, we provide evidence for the effectiveness of these techniques in industrial settings (Chapter 4).

3. **Risk-based Prioritization of Test Gaps:** Our third contribution proposed a risk-based prioritization approach for test gaps, which we evaluated using a multimethod study. This approach targets the problem of the large number of test gaps with varying risk which need to be inspected manually during (1) test planning or (2) assessing test-end criteria. To solve this problem, we proposed TestGapRadar, an automated, score-based approach that prioritizes test gaps based on estimated risk, which considers the magnitude of impact and defect probability. We validated the risk criteria's transferability through a multimethod study which involved historical quality assurance reports from eight industrial software systems of Munich Re and LV 1871, and semi-structured in-

terviews with six quality engineers that authored the reports. Our results showed the effectiveness of our approach from two perspectives: First, risky test gaps are ranked significantly higher than less risky test gaps, on average, at the 30th percentile. Second, TestGapRadar is on par with expert rankings, and in some cases, even outperforms the expert rating. Our approach is suitable to empower test management and quality assurance teams to efficiently recognize and manage risky test gaps, ensuring that no potentially risky code change is released untested. To facilitate adoption, we presented guidelines for practitioners to implement a risk-based prioritization of test gaps in their testing process (Chapter 5).

Overall, this thesis contributes toward effective and efficient human-in-the-loop testing processes in industry, including enhanced feedback, shorter test runtimes, and less remaining risks when testing has completed. It presents various optimization levers to conquer slow test feedback in manual testing and suggests specific optimization techniques to leverage these optimization potentials. To overcome risks from untested changes, we suggest an approach to prioritize test gaps, enabling humans-in-the-loop to mitigate the largest risks from untested changes as early as possible. Our studies have shown the suitability and effectiveness of suggested solutions in practice and, after our studies, many study subjects decided to anchor our solutions in their testing process. So, our results are of scientific and practical value: Researchers can build upon our studies and methodologies to address further open questions in the field of optimizing human-in-the-loop testing processes. Practitioners, such as developers, testers, management, and quality engineers, can learn from the insights of our empirical studies in industry contexts and adopt our work—following our practice-oriented guidelines—to optimize human-in-the-loop testing processes in their contexts.

6.2 Future Work

While conducting the research presented in this dissertation, we have identified four promising avenues of further research.

Further Enhancements of Manual Testing Processes In practice, manual testing processes show great variance. Our studies on optimization of manual testing presented in Chapter 3 and Chapter 4 have covered only a fraction of possible testing process parameters. Further research is needed to gather a more complete understanding of manual testing process optimization. Collecting coverage of manual tests can be challenging, since end-to-end tests typically involve multiple languages and frameworks, complicating the optimization effort. Instead, future work should include further optimization techniques, possibly leveraging other data than code coverage information. In the era of AI, approaches building up on information retrieval or large language models might facilitate manual test optimization. For example, test cases might be selected on similarity measures between test cases and source code: text/code embeddings or LLM-generated summaries of test cases and sources could be used as similarity measure, so that the most similar test cases to changed source code can be selected for test execution.

Cross-Industry Analysis, Benchmarking, and Longitudinal Studies on Optimization Impact In our research, we covered several manual testing processes from various domains. Our data do not suffice to provide a cross-industry overview of manual testing processes. For this purpose, comparative studies across different industries should be conducted to identify domain-specific challenges and commonalities in test optimization. This data could also be leveraged to benchmark the effectiveness of optimization techniques in various industrial contexts. We see additional value in longitudinal studies to assess the long-term benefits and evolution of manual testing process optimization. Such studies could evaluate improvements in software quality, defect rates, and coding throughput.

Enhancements of Test Gap Risk Estimation and Mitigation Future work could enhance our approach to estimate the risk of test gaps by taking additional risk factors into account, such as production usage data, natural-language processing of commit messages, as well as socio-technical analyses and metrics. Alternatively, approaches from the defect prediction field applying AI could be transferred to assess the risk of individual test gaps. Additionally, the accumulation of test gap risk opens an interesting field of research: by now, it remains unclear how the information on test gap risk for code changes within different work items should be aggregated and how to compare a set of work items with regards to their accumulated risk. Also, there is need for guidance in closing test gaps, that is, collaboration strategies between testers and developers. In manual testing, old test cases that formerly executed changed but currently untested code could be suggested for execution by implementing coverage-based optimization approaches. Future studies could apply test case generation techniques from the literature to generate test cases from test gap information.

Real-Time Suggestions to Improve Manual Testing Effectiveness The research fields discussed in this thesis, manual test optimization and risk mitigation of untested changes, could be combined to mitigate test gap risks on-the-fly during manual test execution. For this purpose, future work shall focus on the question of how to guide testers during testing to close test gaps on-the-fly. This could involve an interaction-distance measure which expresses the amount of necessary user interaction to trigger execution of a specific piece of code, for example, a test gap, from the current system state. Thus, nearby-test gaps could be suggested to manual testers, possibly also indicating which system-interaction is required to close the gap. This kind of guided exploratory testing requires domain knowledge so that testers can actually detect deviations between the system under test's intended and actual behavior.

In conclusion, this dissertation contributed on knowledge about human-in-the-loop testing processes in industry and their optimization levers. There are many further directions for research that can build upon our work, and opportunities to leverage untapped optimization potential in practice, both shall serve toward better software. We are convinced that humans will continue to play a decisive role in the future of software testing, and the question how to make optimal use of the human skills will continue to drive us in the future.

A

Appendix

In this dissertation, we have presented three empirical studies on optimization of human-in-the-loop testing processes to overcome slow feedback from manual testing and eliminate risks of relevant test gaps for test completion assessment. For all studies, we share supplemental material on Web sites, that allow in-depth navigation through our study data, obtain analysis and plotting scripts for traceability and reproducibility of our results. In some cases, the raw data can not be shared because of restrictions by confidentiality agreements with our research partners. Then, we published aggregated data to allow for traceability of our conclusions. Some repositories share additional details on study setups, results and discussions. In what follows, we briefly summarize the supplemental material of each study and point to the corresponding Web sites.

A.1 Supplemental Material for Chapter 3

The survey results, analyses, and the optimization guidelines of our study [53] are publicly available in our supplemental repository:
`https://github.com/manual-testing-study/manual-testing-esec-fse-21/`.

A.2 Supplemental Material for Chapter 4

The raw data obtained in our empirical study [56] cannot be shared because of confidentiality agreements. For reproducibility, we published aggregated data and the analysis scripts, along with additional details on our subjects, our questionnaire, and a subject specific discussion of results on a supplementary Web site: `https://zenodo.org/records/11502386`.

A.3 Supplemental Material for Chapter 5

The raw data obtained in our study [57] cannot be shared because of confidentiality agreements. For reproducibility, we published aggregated data and the analysis scripts, along with additional details on our studies on a supplementary Web site:
`https://github.com/se-sic/test-gap-risk-study`.

Bibliography

[1] n. A. "International Standard—Software and systems engineering –Software testing –Part 1:General concepts." In: *ISO/IEC/IEEE 29119-1:2022(E)* (2022), pp. 1–60.

[2] n. A. *Next-generation quality intelligence*. Ed. by Tricentis Sealights. 2025. URL: `https://www.tricentis.com/products/quality-intelligence-sealights` (visited on 02/25/2025).

[3] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce. "Evaluating non-adequate test-case reduction." In: *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2016, pp. 16–26.

[4] D. Amalfitano, S. Faralli, J. C. R. Hauck, S. Matalonga, and D. Distante. "Artificial Intelligence Applied to Software Testing: A Tertiary Study." In: *ACM Computing Surveys* 56.3, 58 (2023).

[5] M. Bagherzadeh, N. Kahani, and L. C. Briand. "Reinforcement Learning for Test Case Prioritization." In: *Transactions on Software Engineering* 48.8 (2021), pp. 2836–2856.

[6] K. Barbosa, R. Ferreira, G. Pinto, M. d'Amorim, and B. Miranda. "Test Flakiness Across Programming Languages." In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 2039–2052.

[7] Y. Baron and K. Yitmen. *ISTQB® Worldwide Software Testing Practices*. ISTQB, 2018. URL: `https://www.turkishtestingboard.org/files/ISTQB-Worldwide-Software-Testing-Practices-Report-2017-18.pdf` (visited on 02/25/2025).

[8] E. Bernard, J. Botella, F. Ambert, B. Legeard, and M. Utting. "Tool Support for Refactoring Manual Tests." In: *Proceedings of the International Conference on Software Testing, Validation and Verification*. IEEE, 2020, pp. 332–342.

[9] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Lango, and T. Gyimóthy. "Code Coverage-based Regression Test Selection and Prioritization in WebKit." In: *Proceedings of the International Conference on Software Maintenance*. IEEE, 2012, pp. 46–55.

[10] B. W. Boehm. "Verifying and Validating Software Requirements and Design Specifications." In: *IEEE Software* 1.1 (1984), pp. 75–88.

[11] M. Boehme, C. Cadar, and A. Roychoudhury. "Fuzzing: Challenges and Reflections." In: *IEEE Software* 38.3 (2021), pp. 79–86.

[12] E. Borgonovo and E. Plischke. "Sensitivity analysis: A review of recent advances." In: *European Journal of Operational Research* 248.3 (2016), pp. 869–887.

[13] C. Brandt, M. Castelluccio, C. Holler, J. Kratzer, A. Zaidman, and A. Bacchelli. "Mind the Gap: What Working With Developers on Fuzz Tests Taught Us About Coverage Gaps." In: *Proceedings of the International Conference on Software Engineering (Software Engineering in Practice)*. ACM, 2024, pp. 157–167.

[14] L. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh. "The Case for Context-Driven Software Engineering Research: Generalizability Is Overrated." In: *IEEE Software* 34.5 (2017), pp. 72–75.

[15] G. Buchgeher, C. Ernstbrunner, R. Ramler, and M. Lusser. "Towards Tool-Support for Test Case Selection in Manual Regression Testing." In: *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013, pp. 74–79.

[16] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric. "Regression Test Selection Across JVM Boundaries." In: *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2017, pp. 809–820.

[17] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. "TestTube: A System for Selective Regression Testing." In: *Proceedings of the International Conference on Software Engineering*. IEEE, 1994, pp. 211–220.

[18] R. Cheng, L. Zhang, D. Marinov, and T. Xu. "Test-Case Prioritization for Configuration Testing." In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2021, pp. 452–465.

[19] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner. "Finding Faults: Manual Testing vs. Random+ Testing vs. User Reports." In: *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE, 2008, pp. 157–166.

[20] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. 2nd ed. Lawrence Erlbaum Associates, 1988.

[21] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri. "Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics." In: *IEEE Transactions on Software Engineering* 48.6 (2021), pp. 2086–2104.

[22] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." In: *Computer* 11.4 (1978), pp. 34–41.

[23] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. "Refactoring Test Code." In: *Proceedings of the International Conference on Extreme Programming and Flexible Processes in Software Engineering*. CWI. 2001, pp. 92–95.

[24] D. Di Nardo, N. Alshahwan, L. C. Briand, and Y. Labiche. "Coverage-Based Regression Test Case Selection, Minimization and Prioritization: A Case Study on an Industrial System." In: *Software Testing, Verification and Reliability* 25.4 (2015), pp. 371–396.

[25] C. Di Nucci, F. Palomba, G. de Rosa, G. Bavota, R. Oliveto, and A. de Lucia. "A Developer Centered Bug Prediction Model." In: *IEEE Transactions on Software Engineering* 44.1 (2018), pp. 5–24.

[26] F. Dreier. “Obtaining Coverage per Test Case.” Master’s thesis, Technical University of Munich. Technical University of Munich, 2017. URL: https://teamscale.com/hubfs/Publications/2017-obtaining-coverage-per-test-case.pdf (visited on 02/25/2025).

[27] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K.-H. Prommer. “Did We Test Our Changes? Assessing Alignment Between Tests and Development in Practice.” In: *Proceedings of the International Workshop on Automation of Software Test*. IEEE, 2013, pp. 107–110.

[28] S. Eder, B. Hauptmann, M. Junker, R. Vaas, and K.-H. Prommer. “Selecting Manual Regression Test Cases Automatically using Trace Link Recovery and Change Coverage.” In: *Proceedings of the International Workshop on Automation of Software Test*. ACM, 2014, pp. 29–35.

[29] S. Elbaum, A. Malishevsky, and G. Rothermel. “Prioritizing Test Cases for Regression Testing.” In: *Proceedings of the International Symposium on Software Testing and Analysis*. IEEE, 2000, pp. 101–112.

[30] S. Elbaum, A. Malishevsky, and G. Rothermel. “Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization.” In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2001, pp. 329–338.

[31] S. Elbaum, G. Rothermel, and J. Penix. “Techniques for Improving Regression Testing in Continuous Integration Development Environments.” In: *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM, 2014, pp. 235–245.

[32] S. Eldh. “On Technical Debt in Software Testing—Observations from Industry.” In: *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2022, pp. 301–323.

[33] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer. “Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration.” In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2021, pp. 491–504.

[34] D. Elsner, S. Kacianka, S. Lipp, A. Pretschner, A. Habermann, M. Graber, and S. Reimer. “BinaryRTS: Cross-language Regression Test Selection for C++ Binaries in CI.” In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2023.

[35] E. Engström, P. Runeson, and A. Ljung. “Improving Regression Testing Transparency and Efficiency with History-Based Prioritization—An Industrial Case Study.” In: *Proceedings of the International Conference on Software Testing, Verification, and Validation*. IEEE, 2011, pp. 367–376.

[36] E. Engström, P. Runeson, and M. Skoglund. “A Systematic Review on Regression Test Selection Techniques.” In: *Information and Software Technology* 52.1 (2010), pp. 14–30.

[37] E. Enoiu, D. Sundmark, A. Čaušević, and P. Pettersson. "A Comparative Study of Manual and Automated Testing for Industrial Control Software." In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2017, pp. 412–417.

[38] European Parliament and of the Council. *Directive 2008/104/EC of the European Parliament and of the Council of 19 November 2008 on temporary agency work*. Official Journal of the European Union. 2022. URL: https://eur-lex.europa.eu/eli/dir/2008/104/oj (visited on 02/25/2025).

[39] European Parliament and of the Council. *Regulation on digital operational resilience for the financial sector and amending Regulations (EC) No 1060/2009, (EU) No 648/2012, (EU) No 600/2014, (EU) No 909/2014 and (EU) 2016/1011*. Official Journal of the European Union. 2022. URL: https://eur-lex.europa.eu/eli/reg/2022/2554 (visited on 02/25/2025).

[40] D. Falessi, S. M. Laureani, J. Çarka, M. Esposito, and D. A. da Costa. "Enhancing the defectiveness prediction of methods and classes via JIT." In: *Empirical Software Engineering* 28, 37 (Jan. 2023).

[41] S. Fatima, T. A. Ghaleb, and L. Briand. "Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests." In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 1912–1927.

[42] M. Fowler. *TestPyramid*. 2012. URL: https://martinfowler.com/bliki/TestPyramid.html (visited on 02/25/2025).

[43] P. G. Frankl, G. Rothermel, K. Sayre, and F. I. Vokolos. "An Empirical Comparison of Two Safe Regression Test Selection Techniques." In: *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE, 2003, pp. 195–204.

[44] G. Fraser and J. M. Rojas. "Software Testing." In: *Handbook of Software Engineering* (2019), pp. 123–192.

[45] V. Garousi and J. Zhi. "A Survey of Software Testing Practices in Canada." In: *Journal of Systems and Software* 86.5 (2013), pp. 1354–1376.

[46] M. Gligoric, L. Eloussi, and D. Marinov. "Ekstazi: Lightweight Test Selection." In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2015, pp. 713–716.

[47] M. Golagha, A. Pretschner, and L. C. Briand. "Can We Predict the Quality of Spectrum-based Fault Localization?" In: *Proceedings of the International Conference on Software Testing, Validation and Verification*. IEEE, 2020, pp. 4–15.

[48] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. "Predicting Fault Incidence Using Software Change History." In: *IEEE Transactions on Software Engineering* 26.7 (2000), pp. 653–661.

[49] R. Greca, B. Miranda, M. Gligoric, and A. Bertolino. "Comparing and Combining File-based Selection and Similarity-based Prioritization towards Regression Test Orchestration." In: *Proceedings of the International Conference on Automation of Software Test*. IEEE, 2022, pp. 115–125.

[50] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. "An Empirical Study of Flaky Tests in Python." In: *Proceedings of the Conference on Software Testing, Verification and Validation*. IEEE, 2021, pp. 148–158.

[51] B. Gruner, C.-A. Brust, and A. Zeller. "Finding Information Leaks with Information Flow Fuzzing." In: *ACM Transactions on Software Engineering and Methodology* (Jan. 2025). accepted for publication 2024-05-03, to appear.

[52] T. Gyimothy, R. Ferenc, and I. Siket. "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction." In: *IEEE Transactions on Software Engineering* 31.10 (2005), pp. 897–910.

[53] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel. "How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies." In: *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021, pp. 1281–1291.

[54] R. Haas, R. Niedermayr, and E. Juergens. "Teamscale: Tackle Technical Debt and Control the Quality of Your Software." In: *Proceedings of the International Conference on Technical Debt*. IEEE, 2019, pp. 55–56.

[55] R. Haas, R. Niedermayr, T. Roehm, and S. Apel. "Is Static Analysis Able to Identify Unnecessary Source Code?" In: *ACM Transactions on Software Engineering and Methodology* 29.1 (2020), pp. 1–23.

[56] R. Haas, R. Nömmer, E. Juergens, and S. Apel. "Optimization of Automated and Manual Software Tests in Industrial Practice: A Survey and Historical Analysis." In: *IEEE Transactions on Software Engineering* 50.8 (2024), pp. 2005–2020.

[57] R. Haas, M. Sailer, M. Joblin, E. Juergens, and S. Apel. "Prioritizing Test Gaps by Risk in Industrial Practice: An Automated Approach and Multimethod Study." In: *IEEE Transactions on Software Engineering* 51.5 (2025), pp. 1554–1568.

[58] A. Habib and M. Pradel. "How Many of All Bugs Do We Find? A Study of Static Bug Detectors." In: *Proceedings of the International Conference on Automated Software Engineering*. ACM, 2018, pp. 317–328.

[59] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering." In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1276–1304.

[60] R. G. Hamlet. "Testing Programs with the Aid of a Compiler." In: *IEEE Transactions on Software Engineering* 3.4 (1977), pp. 279–290.

[61] M. Harrold, R. Gupta, and M. Soffa. "A Methodology for Controlling the Size of a Test Suite." In: *ACM Transactions on Software Engineering and Methodology* 2.3 (1993), pp. 270–285.

[62] M. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. "Empirical Studies of a Prediction Model for Regression Test Selection." In: *Transactions on Software Engineering* 27.3 (2001), pp. 248–263.

[63] N. Hashemi, A. Tahir, and S. Rasheed. "An Empirical Study of Flaky Tests in JavaScript." In: *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2022, pp. 24–34.

[64] A. E. Hassan. "Predicting Faults Using the Complexity of Code Changes." In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2009, pp. 78–88.

[65] B. Hauptmann, L. Heinemann, R. Vaas, and P. Braun. "Hunting for Smells in Natural Language Tests." In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2013, pp. 1217–1220.

[66] H. Hemmati, Z. Fang, and M. Mäntylä. "Prioritizing Manual Test Cases in Traditional and Rapid Release Environments." In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2015, pp. 1–10.

[67] H. Hemmati and F. Sharifi. "Investigating NLP-Based Approaches for Predicting Manual Test Case Failure." In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2018, pp. 309–319.

[68] J. Herman and W. Usher. "SALib: An open-source Python library for sensitivity analysis." In: *Journal of Open Source Software* 2.9 (2017), p. 97.

[69] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi. "DeepJIT: an End-to-End Deep Learning Framework for Just-in-Time Defect Prediction." In: *Proceedings of the International Conference on Mining Software Repositories*. IEEE. 2019, pp. 34–45.

[70] S. Hosseini, B. Turhan, and D. Gunarathna. "A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction." In: *IEEE Transactions on Software Engineering* 45.2 (2017), pp. 111–147.

[71] W. Hudson. "Card Sorting." In: *Encyclopedia of Human-Computer Interaction*. Interaction Design Foundation, 2012.

[72] M. Ivanković, G. Petrović, Y. Kulizhskaya, M. Lewko, L. Kalinovcic, R. Just, and G. Fraser. "Productive Coverage: Improving the Actionability of Code Coverage." In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2024, pp. 58–68.

[73] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. "Reporting Experiments in Software Engineering." In: *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 201–228.

[74] M. Joblin and S. Apel. "How Do Successful and Failed Projects Differ? A Socio-Technical Analysis." In: *ACM Transactions on Software Engineering and Methodology* 31.4 (2022), pp. 1–24.

[75] M. Joblin, S. Apel, and W. Mauerer. "Evolutionary trends of developer coordination: A network approach." In: *Empirical Software Engineering* 22 (Aug. 2017), pp. 2050–2094.

[76] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlögel, and A. Wübbeke. "Regression Test Selection of Manual System Tests in Practice." In: *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 309–312.

[77] E. Juergens and D. Pagano. *Did We Test the Right Thing? Experience with Test Gap Analysis in Practice*. White Paper. CQSE GmbH, 2018.

[78] E. Juergens, D. Pagano, and A. Goeb. *Test Impact Analysis: Detecting Errors Early Despite Large, Long-Running Test Suites*. White Paper. CQSE GmbH, 2018.

[79] Y. Kamei and E. Shihab. "Defect Prediction: Accomplishments and Future Challenges." In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE. 2016, pp. 33–45.

[80] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. "A Large-Scale Empirical Study of Just-in-Time Quality Assurance." In: *IEEE Transactions on Software Engineering* 39.6 (2012), pp. 757–773.

[81] R. Kazmi, D. Jawawi, R. Mohamad, and I. Ghani. "Effective Regression Test Case Selection: A Systematic Literature Review." In: *Computing Surveys* 50.2 (2017), pp. 1–32.

[82] M. Kendall. "The Treatment of Ties in Ranking Problems." In: *Biometrika* 33.3 (1945), pp. 239–251.

[83] M. Khatibsyarbini, M. A. Isa, D. N. A. Jawawi, and R. Tumeng. "Test Case Prioritization Approaches in Regression Testing: A Systematic Literature Review." In: *Information and Software Technology* 93 (Jan. 2018), pp. 74–93.

[84] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. "Early Quality Prediction: A Case Study in Telecommunications." In: *IEEE Software* 13.1 (1996), pp. 65–71.

[85] S. Kim, E. J. Whitehead, and Y. Zhang. "Classifying Software Changes: Clean or Buggy?" In: *IEEE Transactions on Software Engineering* 34.2 (2008), pp. 181–196.

[86] H. Kirinuki and H. Tanno. "Automating End-to-End Web Testing via Manual Testing." In: *Journal of Information Processing* 30 (Apr. 2022), pp. 294–306.

[87] R. Lachmann, M. Nieke, C. Seidl, I. Schaefer, and S. Schulze. "System-Level Test Case Prioritization using Machine Learning." In: *Proceedings of the International Conference on Machine Learning and Applications*. IEEE, 2016, pp. 361–368.

[88] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. "Root Causing Flaky Tests in a Large-Scale Industrial Setting." In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 101–111.

[89] W. Lam, K. Muslu, H. Sajnani, and S. Thummalapenta. "A Study on the Lifecycle of Flaky Tests." In: *Proceedings of the International Conference of Software Engineering*. ACM, 2020, pp. 1471–1482.

[90] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. "An Extensive Study of Static Regression Test Selection in Modern Software Evolution." In: *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 583–594.

[91] A. Leitner, H. Ciupa, B. Meyer, and M. Howard. "Reconciling Manual and Automated Testing: The AutoTest Experience." In: *Proceedings of the Annual Hawaii International Conference on System Sciences*. IEEE, 2007, pp. 261–270.

[92] Z. Li, X-Y. Jing, and X. Zhu. "Progress on approaches to software defect prediction." In: *IET Software* 12.3 (2018), pp. 161–175.

[93] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. "Fuzzing: State of the Art." In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 1199–1218.

[94] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. "An Empirical Analysis of Flaky Tests." In: *Proceedings of the Symposium on the Foundations of Software Engineering*. ACM, 2014, pp. 643–653.

[95] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta. "Assessing Test Case Prioritization on Real Faults and Mutants." In: *Proceedings of the International Conference on Software Maintenance*. IEEE, 2018, pp. 240–251.

[96] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. "Predictive Test Selection." In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2019, pp. 91–100.

[97] D. Marijan and M. Liaaen. "Practical Selective Regression Testing with Effective Redundancy in Interleaved Tests." In: *Proceedings of the International Conference on Software Engineering*. ACM, 2018, pp. 153–162.

[98] F. Matloob, T. M. Ghazal, N. Taleb, S. Aftab, M. Ahmad, M. A. Khan, S. Abbas, and T. R. Soomro. "Software Defect Prediction Using Ensemble Learning: A Systematic Literature Review." In: *IEEE Access* 9 (July 2021), pp. 98754–98771.

[99] P. Mayring. *Qualitative Content Analysis: A Step-by-Step Guide*. SAGE, 2021.

[100] T. J. McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* 2.4 (1976), pp. 308–320.

[101] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. "Taming Google-Scale Continuous Testing." In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 2017, pp. 233–242.

[102] T. Menzies and A. Marcus. "Automated Severity Assessment of Software Defect Reports." In: *Proceedings of the International Conference on Software Maintenance*. IEEE, 2008, pp. 346–355.

[103] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino. "FAST Approaches to Scalable Similarity-Based Test Case Prioritization." In: *Proceedings of the International Conference on Software Engineering*. ACM, 2018, pp. 222–232.

[104] A. Mockus and D. M. Weiss. "Predicting Risk of Software Changes." In: *Bell Labs Technical Journal* 5.2 (2000), pp. 169–180.

[105] R. Moser, W. Pedrycz, and G. Succi. "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction." In: *Proceedings of the International Conference on Software Engineering*. ACM, 2008, pp. 181–190.

[106] M. Mossige, A. Gotlieb, H. Spieker, H. Meling, and M. Carlsson. "Time-Aware Test Case Execution Scheduling for Cyber-Physical Systems." In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming*. Springer, 2017, pp. 387–404.

[107] *Microsoft Azure DevOps Server*. 2025. URL: https://azure.microsoft.com/en-us/products/devops/server/ (visited on 02/25/2025).

[108] J. C. Munson and T. M. Khoshgoftaar. "The Detection of Fault-Prone Programs." In: *IEEE Transactions on Software Engineering* 18.5 (1992), pp. 423–433.

[109] M. Nachtigall, M. Schlichtig, and E. Bodden. “A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools.” In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2022, pp. 532–543.

[110] N. Nagappan, B. Murphy, and V. Basili. “The Influence of Organizational Structure on Software Quality.” In: *Proceedings of the International Conference on Software Engineering*. ACM, 2008, pp. 521–530.

[111] T. Nakagawa, K. Munakata, and K. Yamamoto. “Applying Modified Code Entity-Based Regression Test Selection for Manual End-To-End Testing of Commercial Web Applications.” In: *Proceedings of the International Symposium on Software Reliability Engineering Workshops*. IEEE, 2019, pp. 1–6.

[112] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. “Regression Testing in the Presence of Non-Code Changes.” In: *Proceedings of the International Conference on Software Testing, Verification, and Validation*. IEEE, 2011, pp. 21–30.

[113] R. Niedermayr, T. Röhm, and S. Wagner. “Too trivial to test? An inverse view on defect prediction to identify methods with low fault risk.” In: *PeerJ Computer Science* 5.2, e187 (2019).

[114] R. Nömmer. “Conception and Evaluation of Test Suite Minimization Techniques for Regression Testing in Practice.” Master’s Thesis. Technical University of Munich, 2019. URL: https://teamscale.com/publication-detail/261 (visited on 02/25/2025).

[115] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. “Predicting the Location and Number of Faults in Large Software Systems.” In: *IEEE Transactions on Software Engineering* 31.4 (2005), pp. 340–355.

[116] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report. Stanford InfoLab, 1999.

[117] F. Palomba, M. Zanoni, F. A. Fontana, A. de Lucia, and R. Oliveto. “Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells.” In: *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 244–255.

[118] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. C. Briand. “Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review.” In: *Empirical Software Engineering* 27.2 (2022), pp. 1–43.

[119] R. Pan, T. A. Ghaleb, and L. C. Briand. “LTM: Scalable and Black-Box Similarity-Based Test Suite Minimization Based on Language Models.” In: *IEEE Transactions on Software Engineering* 50.11 (2024), pp. 3053–3070.

[120] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn. “Test smells 20 years later: detectability, validity, and reliability.” In: *Empirical Software Engineering* 27.7, 170 (2022).

[121] L. Pascarella, F. Palomba, and A. Bacchelli. “Re-evaluating Method-Level Bug Prediction.” In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2018, pp. 592–601.

[122] L. Pascarella, F. Palomba, and A. Bacchelli. “On the performance of method-level bug prediction: A negative result.” In: *Journal of Systems and Software* 161, 110493 (Mar. 2020).

[123] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. “Evaluating and Improving Fault Localization.” In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2017, pp. 609–620.

[124] Q. Peng, A. Shi, and L. Zhang. “Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization.” In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2020, pp. 324–336.

[125] G. Petrović, M. Ivanković, G. Fraser, and R. Just. “Does mutation testing improve testing practices?” In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2021, pp. 910–921.

[126] G. Petrović, M. Ivanković, G. Fraser, and R. Just. “Practical Mutation Testing at Scale: A view from Google.” In: *IEEE Transactions on Software Engineering* 48.10 (2022), pp. 3900–3912.

[127] A. Philip, R. Bhagwan, R. Kumar, C. Maddila, and N. Nagppan. “FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services.” In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2019, pp. 408–418.

[128] L. S. Pinto, S. Sinha, and A. Orso. “Understanding Myths and Realities of Test-Suite Evolution.” In: *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM, 2012, pp. 1–11.

[129] C. Popeea-Simeth. *Monthly assessments: How often to inspect the code quality in a code quality control process?* 2018. URL: https://teamscale.com/blog/en/news/blog/monthly-assessments (visited on 02/25/2025).

[130] C. Pornprasit and C. K. Tantithamthavorn. “DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction.” In: *IEEE Transactions on Software Engineering* 49.1 (2023), pp. 84–98.

[131] R. S. Pressman. *Software Engineering: a practitioner's approach*. Vol. 7. McGraw-Hill Education, 2009.

[132] A. Pretschner. “Defect-Based Testing.” In: *Dependable Software Systems Engineering*. IOS Press, 2015, pp. 224–245.

[133] S. U. Rehman Khan, S. P. Lee, N. Javaid, and W. Abdul. “A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines.” In: *IEEE Access* 6 (2018), pp. 11816–11841.

[134] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona. “How bugs are born: a model to identify how bugs are introduced in software components.” In: *Empirical Software Engineering* 25 (2020), pp. 1294–1340.

[135] R. Rosenthal and K. Fode. “The Effect of Experimenter Bias on the Performance of the Albino Rat.” In: *Behavioral Science* 8.3 (1963), pp. 183–189.

[136] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. "Test Case Prioritization: an Empirical Study." In: *Proceedings of the International Conference on Software Maintenance*. IEEE, 1999, pp. 179–188.

[137] J. Rott. "Test Intelligence: How Modern Analyses and Visualizations in Teamscale Support Software Testing." In: *Proceedings of the International Workshop on Visualization in Testing of Hardware, Software, and Manufacturing*. IEEE, 2022, pp. 15–21.

[138] P. Runeson, M. Host, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st. Wiley, 2012.

[139] M. Sailer. "Grouping and Prioritization of Test Gaps." Master's Thesis. Technical University of Munich, 2019. URL: `https://teamscale.com/2019-grouping-and-prioritization-of-test-gaps` (visited on 02/25/2025).

[140] V. Y. Shen, Tze-jie Yu, S. M. Thebaut, and L. R. Paulsen. "Identifying Error-Prone Software—An Empirical Study." In: *IEEE Transactions on Software Engineering* 11.4 (1985), pp. 317–324.

[141] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov. "Evaluating Test-Suite Reduction in Real Software Evolution." In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 84–94.

[142] A. Shi, T. Yung, A. Gyori, and D. Marinov. "Comparing and Combining Test-Suite Reduction and Regression Test Selection." In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2015, pp. 237–247.

[143] S. Y. Shin, S. Nejati, M. Sabetzadeh, L. C. Briand, and F. Zimmer. "Test Case Prioritization for Acceptance Testing of Cyber Physical Systems: A Multi-Objective Search-Based Approach." In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 49–60.

[144] J. Siegmund, N. Siegmund, and S. Apel. "Views on Internal and External Validity in Empirical Software Engineering." In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2015, pp. 9–19.

[145] E. Soares, M. Aranda, N. Oliveira, M. Ribeiro, R. Gheyi, E. Souza, I. Machado, A. Santos, B. Fonseca, and R. Bonifácio. "Manual Tests Do Smell! Cataloging and Identifying Natural Language Test Smells." In: *International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2023, pp. 1–11.

[146] I. M. SoboÍ. "Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates." In: *Mathematics and Computers in Simulation* 55.1 (2001), pp. 271–280.

[147] I. Şora. "A PageRank Based Recommender System for Identifying Key Classes in Software Systems." In: *Proceedings of the International Symposium on Applied Computational Intelligence and Informatics*. IEEE, 2015, pp. 495–500.

[148] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler. "Continuous Software Quality Control in Practice." In: *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 561–564.

[149] D. Steidl, B. Hummel, and E. Juergens. "Using Network Analysis for Recommendation of Central Software Classes." In: *Proceedings of the Working Conference on Reverse Engineering*. IEEE, 2012, pp. 93–102.

[150] K. Stol and B. Fitzgerald. "The ABC of Software Engineering Research." In: *ACM Transactions on Software Engineering and Methodology* 27.3, 11 (2018).

[151] S. Stradowski and L. Madeyski. "Industrial applications of software defect prediction using machine learning: A business-driven systematic literature review." In: *Information and Software Technology* 159, 107192 (July 2023).

[152] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander. "Trade-Off Between Automated and Manual Software Testing." In: *International Journal of Systems Assurance Engineering and Management* 2.2 (2011), pp. 114–125.

[153] H. D. Tessema and S. L. Abebe. "Enhancing Just-in-Time Defect Prediction Using Change Request-Based Metrics." In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. IEEE. 2021, pp. 511–515.

[154] P. Tourani and B. Adams. "The Impact of Human Discussions on Just-in-Time Quality Assurance: An Empirical Study on OpenStack and eclipse." In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. Vol. 1. IEEE. 2016, pp. 189–200.

[155] A. Trautsch, J. Erbel, S. Herbold, and J. Grabowski. "What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes." In: *Empirical Software Engineering* 28, 30 (Jan. 2023).

[156] A. Trautsch, S. Herbold, and J. Grabowski. "Static Source Code Metrics and Static Analysis Warnings for Fine-Grained Just-in-Time Defect Prediction." In: *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE. 2020, pp. 127–138.

[157] S. van der Burg and E. Dolstra. "Automating System Tests using Declarative Virtual Machines." In: *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 181–190.

[158] P. Virtanen, R. Gommers, and T. et al. Oliphant. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17 (Mar. 2020), pp. 261–272.

[159] S. Wagner, D. Mendez, M. Felderer, D. Graziotin, and M. Kalinowski. "Challenges in Survey Research." In: *Contemporary Empirical Methods in Software Engineering*. Ed. by M. Felderer and G. H. Travassos. Springer, 2020, pp. 93–125.

[160] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang. "Perceptions, Expectations, and Challenges in Defect Prediction." In: *Transactions on Software Engineering* 46.11 (2018), pp. 1241–1266.

[161] S. Wang, Y. Lian, D. Marinov, and T. Xu. "Test Selection for Unified Regression Testing." In: *Proceedings of the International Conference on Software Engineering*. IEEE, 2023, pp. 1687–1699.

[162] A. H. Watson, D. R. Wallace, and T. J. McCabe. *Structured Testing: A Testing Methodology using the Cyclomatic Complexity Metric*. Vol. 500. 235. US National Institute of Standards and Technology, 1996.

[163] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist. "Impediments for Software Test Automation: A Systematic Literature Review." In: *Software Testing Verification and Reliability* 27.8, e1639 (2017).

[164] R. Wuersching, D. Elsner, F. Leinen, A. Pretschner, G. Grueneissl, T. Neumeyr, and T. Vosseler. "Severity-Aware Prioritization of System-Level Regression Tests in Automotive Software." In: *Proceedings of the Conference on Software Testing, Verification and Validation*. IEEE, 2023, pp. 398–409.

[165] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand. "Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts." In: *Transactions on Software Engineering* 49.4 (2023), pp. 1615–1639.

[166] P. Yi, H. Wang, T. Xie, D. Marinov, and W. Lam. "A Theoretical Analysis of Random Regression Test Prioritization." In: *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, pp. 217–235.

[167] S. Yin, S. Guo, H. Li, C. Li, R. Chen, X. Li, and H. Jiang. "Line-Level Defect Prediction by Capturing Code Contexts with Graph Convolutional Networks." In: *IEEE Transactions on Software Engineering* 51.1 (2024), pp. 172–191.

[168] S. Yoo and M. Harman. "Regression Testing Minimization, Selection and Prioritization: A Survey." In: *Software Testing Verification and Reliability* 22.2 (2012), pp. 67–120.

[169] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen. "Mining Software Repositories to Study Co-Evolution of Production & Test Code." In: *Proceedings of the International Conference on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 220–229.

[170] A. Zeller and R. Hildebrandt. "Simplifying and isolating failure-inducing input." In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200.

[171] J. Zhang, Y. Liu, M. Gligoric, O. Legunsen, and A. Shi. "Comparing and Combining Analysis-Based and Learning-Based Regression Test Selection." In: *Proceedings of the International Conference on Automation of Software Test*. IEEE, 2022, pp. 17–28.

[172] Y. Zhao, K. Damevski, and H. Chen. "A Systematic Survey of Just-in-Time Software Defect Prediction." In: *ACM Computing Surveys* 55.10 (2023), pp. 1–35.

[173] W. Zheng, G. Liu, M. Zhang, X. Chen, and W. Zhao. "Research Progress of Flaky Tests." In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2021, pp. 639–646.

[174] H. Zhong, L. Zhang, and S. Khurshid. "TestSage: Regression Test Selection for Large-Scale Web Service Testing." In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 2019, pp. 430–440.

[175] Q. Zhu, A. Panichella, and A. Zaidman. "A systematic literature review of how mutation testing supports quality assurance processes." In: *Software Testing, Verification and Reliability* 28.6, e1675 (2018).