TUM School of Computation,
Information and Technology

der Technischen Universität München

Dissertation in Informatik

Integrating System and Process Characteristics into Regression Test Optimization

Daniel Valentin Elsner

Technische Universität München
TUM School of Computation, Information and Technology

Integrating System and Process Characteristics into Regression Test Optimization

Daniel Valentin Elsner

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz: Prof. Dr. Thomas Neumann

Prüfer*innen der Dissertation:

1. Prof. Dr. Alexander Pretschner
2. Prof. Darko Marinov, Ph.D.,
   University of Illinois at Urbana-Champaign, USA
3. Prof. Dr. Yves Le Traon,
   Université du Luxembourg, Luxembourg

Die Dissertation wurde am 24.04.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 27.09.2023 angenommen.

Acknowledgments

First of all, I would like to thank my supervisor Alexander Pretschner. He taught me to think more critically and played a major role in shaping this work through valuable discussions. He also always encouraged my independence and showed me how to handle difficult situations with confidence and humor. Lastly, thanks to his persistent feedback, I will hopefully always remember to keep the *big picture* in mind and not (often) bore people with *core dumps* on technical details.

I would further like to thank Darko Marinov and Yves Le Traon for agreeing to be part of my thesis committee and especially thank Darko for the feedback and guidance on some of my work.

The choice of problems addressed by thesis has largely been influenced by industrial cooperations. My greatest thanks here go to the colleagues from CQSE and IVU who have helped me understand context-specific challenges in an always constructive exchange.

I am also grateful for the working environment at the chair with great students and colleagues that I have enjoyed working with and learning from. A special shout-out to my office mate Markus, it was my pleasure!

Finally, my utmost gratitude goes to my family and especially my wife, Lorena. I am not sure if I could have completed the doctorate without your unconditional support and encouragement.

Zusammenfassung

Problemdomäne Um sicherzustellen, dass Änderungen während der Entwicklung von Software nicht unbeabsichtigt Fehler in das bestehende Systemverhalten eingeführt haben, werden Regressionstests in Software durchgeführt. Die einfachste Strategie für das Regressionstesten besteht darin, nach jeder Änderung jeden Testfall in der Testsuite auszuführen. Bei großen Testsuiten, begrenzten Ressourcen oder kurzen Iterationsschleifen kann diese Strategie jedoch zu kostspielig werden. Obwohl im Rahmen zahlreicher Forschungsarbeiten Optimierungstechniken zur Verbesserung der Kosteneffektivität des Regressionstestens entwickelt wurden, kann moderne Software nur teilweise von diesen Fortschritten profitieren. Denn industrielle Softwaresysteme sind in der Regel in mehreren Programmiersprachen geschrieben, hochgradig konfigurierbar und verteilt auf mehrere Maschinen. Außerdem umfassen die Test-Prozesse sowohl automatisierte Tests, die kontinuierlich ausgeführt werden, als auch aufwändige manuelle Tests.

Forschungslücke Die meisten bestehenden Optimierungsverfahren sind angesichts dieser herausfordernden System- und Prozesseigenschaften oft nur eingeschränkt einsetzbar. Denn die traditionellen Verfahren konzentrieren sich auf kleine, einsprachige und monolithische Systeme, die in erster Linie mit automatisierten Unittests getestet werden. Deshalb ist es bisher unklar, wie Verfahren gestaltet sein müssen, um diese Einschränkungen zu überwinden. Dies erschwert den Einsatz und die Verbreitung der Verfahren in der Praxis.

Lösungsansatz Um diese Lücke zu schließen und Forschung und Praxis zusammenzuführen, integriert die vorliegende Dissertation Optimierungsverfahren mit System- und Prozesseigenschaften, die sich in der Praxis für verschiedene Industriepartner als relevant erwiesen haben. Insbesondere werden dafür bestehende und neue Priorisierungs- und Selektionsansätze für Regressionstests angepasst und entwickelt, um Herausforderungen im Zusammenhang mit diesen Eigenschaften adäquat zu adressieren. Dafür kommen sowohl Ansätze der Programmanalyse zum Einsatz, als auch Methoden, die Informationen nutzen, welche ohnehin im Test-Prozess anfallen und somit direkt verfügbar sind. Die Kosteneffektivität der entwickelten Techniken wird in groß angelegten empirischen Studien in Open-Source und industriellen Softwareprojekten evaluiert.

Beitrag Die Ergebnisse der Dissertation zeigen, dass die vorgestellten Techniken und Methoden den Aufwand für Regressionstests und die Feedbackzeit signifikant reduzieren und gleichzeitig das Risiko von nicht erkannten Fehlern minimieren. Je nach kontextspezifischen Anforderungen an die Fehlererkennung, können Techniken, die relevante System- und Prozesseigenschaften berücksichtigen, mittels leichtgewichtigen Heuristiken oder tiefgreifenderer dynamischer Programmanalyse erfolgreich implementiert werden. Aufgrund dieser ermutigenden Ergebnisse werden bereits einige der in dieser Arbeit entwickelten Techniken erfolgreich in einem industriellen Kontext eingesetzt.

Abstract

Problem Domain Regression testing is regularly performed on software systems to ensure that changes have not inadvertently affected existing system behavior. The simplest regression testing strategy is to execute every test case in the test suite after each change. However, with large test suites, limited testing resources, and short software delivery life-cycles this strategy may become too costly to realize. Although significant research on Regression Test Optimization (RTO) has proposed numerous techniques to improve the cost-effectiveness of regression testing, modern software can only partially benefit from these advances: Industrial software systems are commonly written in multiple programming languages, highly configurable and distributed, and require testing processes that involve both continuous automated and effort-intense manual testing.

Research Gap However, most traditional RTO techniques are limited in presence of these challenging system and process characteristics. These techniques focus on small, monolingual, and monolithic systems that are primarily tested with automated unit level tests. It is unclear how RTO techniques can overcome these limitations which hinders the adoption of RTO in practice.

Solution To fill this gap and bring research closer to practice, this doctoral thesis integrates industry-relevant system and process characteristics that affect regression testing into RTO techniques. Particularly, we adapt existing and present novel Regression Test Prioritization (RTP) and Regression Test Selection (RTS) approaches to adequately address challenges related to these characteristics by harnessing program analysis approaches or utilizing information that is readily available from testing processes. We evaluate the cost-effectiveness of the developed techniques in large-scale empirical studies across years of development history in open-source and industrial software projects.

Contribution We find that the presented techniques and methods significantly reduce regression testing effort and developer feedback time while minimizing the risk of missing bugs. Our results show that, depending on context-specific requirements for failure detection, cost-effective RTO techniques that are aware of relevant system and process characteristics can be successfully implemented using either lightweight heuristics or sophisticated dynamic program analysis. Due to these encouraging results, some of the techniques developed in this thesis are used by developers on a daily basis in an industrial context.

Contents

Part I.

Introduction and Background

1. Introduction

This chapter motivates the topic of the doctoral thesis, outlines gaps in the literature, and lists the contributions this thesis makes to address the gaps. Parts of this chapter have appeared in peer-reviewed publications [33–37, 48, 58, 169], co-authored by the author of this thesis.

Software vastly influences wide areas of public and private life: Industry, governments, and entertainment equally rely on a continuously growing body of software products. In light of the immense importance of software, assessing the risk of software failures and thus the quality of software becomes inevitable. According to a recent report by the *Consortium for Information & Software Quality*, the cost of poor software quality in the US was estimated at 2.41 trillion USD in 2022 [76, p. 3]. The largest fraction of costs stems from operational software failures (1.81 trillion USD). Infamous examples from the past decades include the crash of the *Ariane 5* rocket [98], deadly accidents in the medical radiation therapy machine *Therac-25* [92], defective smart *Nest* thermostats [11], and leaking sensitive information due to the widespread *Heartbleed* vulnerability [26]. Since such operational failures can have severe consequences, detecting and fixing them is approximately ten times more expensive than catching them during software development [75, p. 25]. Therefore, early and comprehensive software testing is essential to reduce the risk of critical failures and ultimately enable confident delivery of software.

Software testing is performed to assess the quality of a piece of software and to verify the correctness of its implementation. Finding potential bugs using testing techniques is costly because test cases must be designed, executed, and maintained, and yet no testing technique can give any guarantees that the software is bug-free [136]. The *World Quality Report 2020-21* estimates costs for testing and quality assurance at 22% of the total IT budget in 2020 [126, p. 40].

Regression testing depicts an aspect of testing concerned with revealing bugs in evolving software systems. Specifically, when changes such as source code modifications are introduced to an existing System Under Test (SUT), regression testing is performed to ascertain that these changes did not inadvertently break existing system behavior [91]. These regular checks for software regressions, i.e., bugs affecting existing functionality, can help in detecting declining quality early and reacting accordingly [79]. Regression tests that are run against the SUT are typically collected in test suites, which co-evolve with the code base [144].

When performing regression testing, one simple strategy, namely *retest-all*, is to execute every test case in the test suite after each change. However, with large test suites, limited infrastructure or human resources, and shorter software (delivery) life-cycles as proposed by agile software development methodologies, *retest-all* may become too costly and thus difficult if not impossible to realize [172]. For instance, if *retest-all* takes hours or even days to complete, this can substantially slow down development and incur high costs for running potentially irrelevant tests. Since the late 1970s, these outlined challenges have been subject to research on Regression Test Optimization (RTO) [42]. Particu-

larly, techniques have been proposed for Test Suite Minimization (TSM), Regression Test Selection (RTS), and Regression Test Prioritization (RTP) which all aim to improve the cost-effectiveness of regression testing. The shared motivation behind these techniques is to optimally harness an existing pool of test cases with respect to certain cost-critical properties, e.g., early fault detection or low testing effort [172].

1.1. Industry-Relevant Context Factors Affecting Regression Testing

To support the adoption of RTO in practice, the consideration of industry-relevant context factors in the design of RTO techniques is becoming ever more important [1, 23]. On the one hand, testing processes have changed in recent years as the extent and frequency of regression testing has increased significantly due to rapidly evolving code bases and continuous testing using Continuous Integration (CI) practices [32]. Google reported on average one code commit to their code base per second, triggering more than 150 million test executions every day in 2019 [89]. On the other hand, modern software systems are becoming more complex as they are developed using multiple programming languages with cross-language links, are to a large degree configurable, and are often highly distributed [14, 107, 135, 178]. These system characteristics require elaborate and costly integration or (manual) system testing across language or system boundaries, which, in combination with the highly iterative development and testing processes, increases the relevance of cost-effective regression testing.

However, existing traditional RTO techniques are unsuitable for the described modern software systems and testing processes as they are often limited to small, monolingual, and monolithic systems tested primarily with automated unit tests [14]. As Ali et al. [1] point out in their 2019 study *on the search for industry-relevant regression testing research*, several *context factors* affect regression testing that are critical to the applicability and relevance of RTO techniques. Specifically, the authors group "characteristics of an industrial environment that make regression testing challenging" [1, p. 2036] into *system-*, *process-* and *people-related* context factors. Following their work, this doctoral thesis addresses system- and process-related context factors by integrating *system* and *process characteristics* into regression testing research. By embedding these characteristics into cost-effective RTO techniques, the goal of this thesis is to advance the state-of-the-art, facilitate applicability, and ultimately bring RTO research closer to practice. People-related context factors include organizational and cognitive factors which have rarely been studied in RTO research [1]. As we focus on technical aspects around system and process characteristics, we do not explicitly study people-related context factors in this thesis, but still report on experiences we made when adopting RTO in practice and developer feedback we received from industry partners.

To structure system and process characteristics that affect and challenge regression testing, we provide a conceptual framework in Figure 1.1. Therein, we group system and process characteristics by the system- and process-related context factors provided in the taxonomy developed by Ali et al. [1]. To pick an example, the system characteristic *multilingual* belongs to the system-related context factor *complexity*. The framework does not aim to be exhaustive and can easily be extended by further characteristics that affect regression testing, which we show with several examples (some taken from Ali et al. [1]).

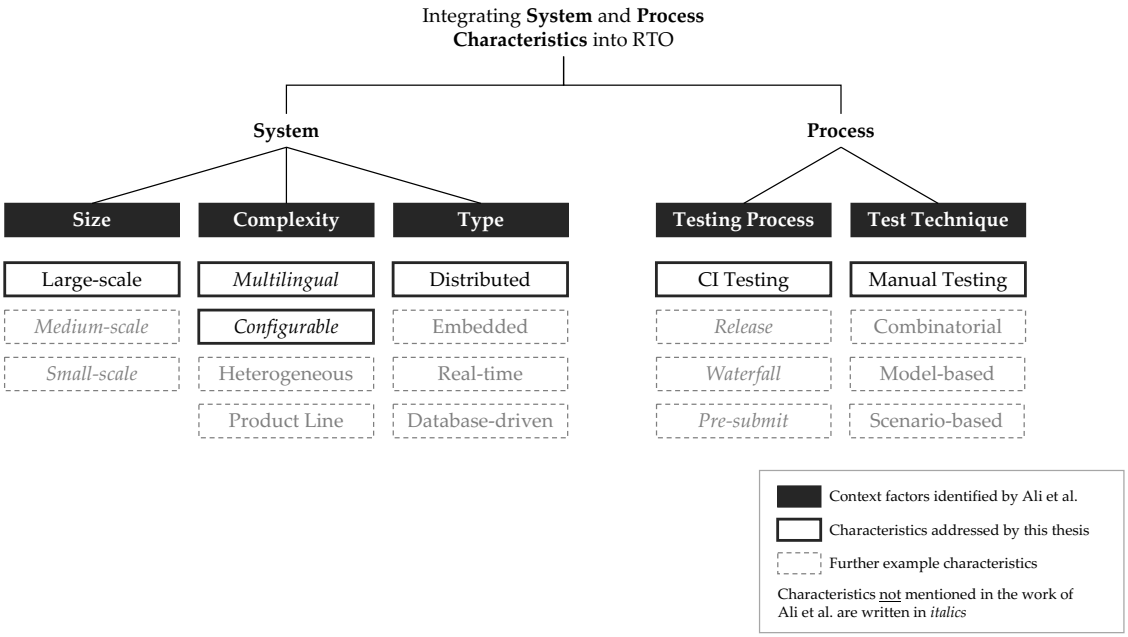


Figure 1.1.: Conceptual framework of system and process characteristics affecting regression testing. We structure the characteristics by the context factors identified by Ali et al. [1].

Within this thesis, we address system and process characteristics that have emerged as especially relevant from the close cooperation and discussion with our industry partners IVU Traffic Technologies[1] and CQSE[2]. In the following, we describe the characteristics tackled by this thesis in more detail. In Section 9.3, we discuss future work with regard to other characteristics.

1.1.1. System Characteristics Affecting Regression Testing

System characteristics concern the nature of the SUT. In alignment with Ali et al. [1], we group the system characteristics studied in this thesis into the three system-related context factors *size*, *complexity*, and *type*.

Size The main industry-relevant concern regarding the size of a system is that RTO techniques need to be applicable in *large-scale* software systems with millions of lines of code [1]. Although RTO is especially relevant in large-scale systems, practitioners report that existing RTO techniques may be prohibitively expensive to apply in these contexts [107, 118].

Complexity Various system characteristics contribute to the complexity of a system, with *multilingual* (or *multi-language*) and *configurable* software being subject of this thesis. Studies published in 2015 and 2017 [115, 116] found that software projects are on average

[1]IVU Traffic Technologies is one of the world's leading providers of public transport software solutions.
[2]CQSE develops a continuous software analysis platform and offers software quality consulting.

written in five to seven General-Purpose Languages (GPLs) and Domain-Specific Languages (DSLs). While cross-language links are useful to develop performance-critical components in native code (C or C++) or domain-specific tasks using DSLs such as SQL, they are reported to increase complexity as program comprehension and analysis in multi-language software is more difficult [97, 115, 117]. Higher complexity increases the risk of bugs, which in turn makes (regression) testing particularly important [73, 93, 94, 116]. Unfortunately, traditional RTO techniques typically rely on language-specific program analysis and are therefore incapable of collecting relevant information across language boundaries, which makes them impractical for multilingual software [16, 36, 107].

Furthermore, modern software is often configurable through non-code artifacts which can impact regression testing [157, 179]. These non-code artifacts may be used for configuration of the system or the test environment, or to define test input and expected output. The impact can be significant, given that, on average, up to 50% of files in open-source software are non-code artifacts and one third of all commits in the Version Control System (VCS) includes changes to these artifacts [10]. Existing RTO techniques often rely on the assumption that testing conditions remain the same except for source code changes [142, 144, 145, 147]. However, this assumption does not hold in the presence of non-code changes [16, 125, 179].

Type The type of the system may limit the applicability of RTO approaches. One type of system that challenges RTO are *distributed* systems, where multiple processes may run on different physical machines [14, 178]. Corresponding regression test suites will typically include integration- or system-level tests to check for more complex interaction and integration bugs between processes or machines [14, 90]. As most RTO techniques focus on single-process, monolithic systems, they are limited in distributed systems [101, 124, 178].

1.1.2. Process Characteristics Affecting Regression Testing

Process characteristics that affect regression testing include aspects of the testing or development process [1]. Ali et al. [1] identify *testing process* and *test technique* as process-related context factors.

Testing Process For the testing process, this thesis addresses continuous regression testing through CI practices as one industry-relevant process characteristic [1]. CI processes have been popularized by agile development methodologies such as *Extreme Programming* [6] to differentiate from traditional testing processes (e.g., *Waterfall*) that test sequentially at different levels and thus have a long feedback cycle and risk late integration problems [65]. According to various studies on RTO in *CI testing*, with large code bases and test suites, CI pipeline execution time is driven by compilation, (static) analysis, and testing efforts, which can slow down iterative development [32, 37, 107, 157]. Therefore, it is essential for adequate RTO techniques to consider how regression testing is embedded into the CI process.

Test Technique Regarding the applied test technique, one widely used technique is *manual testing* which is particularly costly and involves significant human effort [48]. Reasons why test automation is undesired, non-trivial, or impossible are manifold: human testers may detect other software faults than automated tests [21], test automation might be too costly [166] or too complex [13], or manual testing may be required due to legal regulations [48]. Despite the fact that manual testing is widespread and is unlikely to be entirely replaced in near future, regression testing research focuses on automated unit-level testing while fewer works exist for manual (system) testing [14, 48]. Hence, regression test suites largely consisting of such manual (system) tests cannot yet fully benefit from RTO advancements. Paradoxically, due to involved high manual testing effort, it is precisely these tests that impede the development process and must therefore be optimized [48].

1.2. Problem Statement and Research Gaps

The applicability and adoption of RTO is challenged by system and process characteristics in industrial software systems. Related to this practical problem, this thesis addresses the following research gaps (denoted by G):

- *System Characteristics*
  - **G1: Multi-language RTO.** Modern software is multilingual and regression testing is performed across language boundaries. However, most existing RTO techniques employ language-specific program analysis which restricts them to a single language. This is especially relevant for RTS, if no failures should be missed by the test selection algorithm. Yet, with incomplete information on which test depends on what parts of the multilingual code base, RTS safety guarantees are limited [37]. To our knowledge, there exists one RTS technique that goes beyond language boundaries [16], but is limited to Linux systems.
  - **G2: Language-agnostic RTO.** An alternative RTO approach for multi-language software is to rely on purely language-agnostic information. Various studies demonstrate the effectiveness of respective RTP and unsafe RTS techniques that rely on rule-based heuristics or Machine Learning (ML). However, these existing techniques often include project- or organization-specific information that is not available in other contexts, making them difficult to transfer and compare. In addition, results in the literature are equivocal about the cost-effectiveness trade-offs and their sensitivity, and lack empirical guidance on how practitioners should calibrate these techniques [34].
  - **G3: Non-code Artifacts.** Non-source-code files are commonly used to configure systems or to reflect test input, expected test output, or test configuration and may thus affect test results [36, 157, 179]. Existing RTS techniques that track non-code changes are yet limited to projects running on the Java Virtual Machine (JVM) [46, 125] or on Linux [16], while approaches for other environments (e.g., for closed-source operating systems or native binaries) are missing.

  - **G4: End-to-End Testing Microservices.** In distributed systems, e.g., software with microservice architectures, consideration of interface and interaction bugs is essential. Hence, regression testing requires system or end-to-end tests that operate across service boundaries. However, the few existing RTS techniques for web applications or services are limited [101, 124, 178], as they (1) instrument server code, which is incomplete for end-to-end testing of microservices where clients may contain extensive business logic and orchestrate requests to multiple services, (2) target one specific network communication protocol, and (3) lack analysis of instrumentation overhead and effectiveness of collection strategies for per-test dependencies.

- *Process Characteristics*
  - **G5: Build System Integration.** Through CI practices, regression testing is tightly coupled to building the software, as CI pipelines typically first compile and link the code before executing tests. Therefore, RTO techniques ought to consider that (1) the configuration of the build system may impact the test results (e.g., replacing run-time dependencies), and (2) if only parts of the test suite need to be executed, valuable time can be saved by instructing the build system to only build relevant code parts [37]. One existing RTS technique by Shi et al. [157] considers these two aspects, but is rather imprecise in the case of build system configuration changes (1).
  - **G6: Transfer to Manual Testing.** Manual regression testing is widely used and because of the costly human effort involved, there is a great need and potential for optimization. However, unlike automated regression testing, the process for manual testing is often decoupled from the VCS or CI system and performed on production builds without instrumentation [48]. It is unclear how available data can be leveraged to transfer and implement RTO techniques and how to integrate them into manual testing processes.

1.3. Solution

This thesis aims to bring state-of-the-art research closer to practice by integrating system and process characteristics that affect regression testing into RTO approaches. The goal is to develop suitable, cost-effective RTO techniques that reduce developer feedback time and testing effort while still reliably detecting bugs.

The solution proposed in this thesis consists of *methodological* and *technical* parts that tackle the shortcomings of contemporary RTO techniques identified in the research gaps. Our solution comprises methodological guidance on how to (1) harness readily available VCS and CI metadata to construct language-agnostic RTP and unsafe RTS approaches [34], (2) transfer RTO from automated to manual regression testing [48], and (3) overcome challenges when implementing RTS for end-to-end testing in distributed microservice systems [33]. From a technical perspective, this thesis presents novel RTS techniques which rely on static and dynamic program analysis [33, 35, 37]. These techniques advance the state-of-the-art by providing practical approaches to consider (1) multi-language software and non-code artifacts using dynamic binary instrumentation [35] and probe-based

system call tracing [37], and (2) build system configuration through module-level static dependency analysis [37].

To ensure industry-relevance and applicability of the developed solution, we evaluate its cost-effectiveness in large-scale empirical studies and case studies on years of version control history in open-source and industrial projects. Throughout the course of this thesis, we have been involved with engineers, testers, and managers from 18 companies to discuss the research ideas embodied in this thesis [33, 48, 169]. As of today, some of the developed RTS techniques have been deployed in a CI context where they are used by developers on a day-to-day basis for more than a year at IVU, one of our closest industry collaborators [35, 37].

1.4. Contributions

The contributions of this dissertation are constituted by the extent to which the research gaps are filled. Below, we describe the contributions this dissertation makes to fill the gaps in detail, whereas the limitations and threats to validity of the achieved results of this thesis are discussed in Section 9.2:

- To fill **G1**, we contribute two RTS techniques that are aware of cross-language links and thereby suitable for multi-language software. These techniques harness dynamic program analysis to collect per-test execution traces across language boundaries at file or function granularity, depending on the required selection precision. The underlying analysis supports various operating systems which eases transferability. Using these techniques, we were able to effectively reduce the test duration on average by 42%–72% compared to *retest-all* for pull requests in CI environments at IVU without failing to select any real test failures [35, 37].

- To fill **G2**, we contribute a methodology to build and evaluate language-agnostic approaches for RTP and unsafe RTS exclusively using readily available CI and VCS metadata. We empirically evaluate these approaches' cost-effectiveness sensitivity across 23 open-source and industrial software projects and find that (1) sometimes limiting training data is beneficial, (2) test history is a particularly good predictor for future test failures, and (3) complex ML models are often outperformed by simple and inexpensive heuristics. Across all studied software projects, the best approaches found using our methodology significantly outperform RTP baselines and save on average 84% of testing time whilst detecting 90% of test failures for unsafe RTS [34].

- To fill **G3**, we contribute two RTS techniques that use dynamic program analysis to track which test accesses which non-code file. In particular, we harness (1) lightweight probe-based system call tracing and (2) binary instrumentation to monitor which files are opened by processes executing test cases. Again the techniques support various operating systems and we can use the resulting file-level per-test execution traces to select tests in case a non-code file is modified [35, 37].

- To fill **G4**, we contribute a dynamic RTS technique targeting end-to-end testing in distributed microservice-based software systems. Therefore, we combine code instrumentation with polyglot distributed tracing frameworks for protocol-agnostic

tracing of test execution across microservices and clients. We evaluate the instrumentation overhead on a microservice benchmark and find that most overhead occurs at service startup and is mainly caused by the distributed tracing framework. We further describe challenges in applying our technique in an initial case study on an open-source microservice system. The results show that in contrast with automated unit tests, class-level granularity of test traces is too coarse grained for RTS of end-to-end tests. Across 12 studied software versions, method-level RTS can exclude 10%–50% of the end-to-end tests, but only if test cases are not underspecified [33].

- To fill **G5**, we contribute a build system aware RTS technique. Contrary to existing RTS techniques, our technique does not assume a fully compiled workspace to be readily available for test selection. Instead, it operates directly on the source code and determines a minimum set of code modules that are (transitively) affected by changes or contain affected regression tests and must therefore be compiled for testing. Additionally, in the case of build system configuration changes, e.g., changes to run-time dependencies, our technique selects test cases affected by these changes. We were able to reduce end-to-end execution time for Java CI pipelines (including building, analyzing, and testing pull requests) on average by 50%–63% at IVU [37].
- To fill **G6**, we conduct a developer survey among 38 testing professionals from 16 companies and derive methodological guidelines on how to transfer optimization techniques from automated testing to manual testing [48]. We further demonstrate the usefulness of these optimization guidelines in two industrial case studies where we identify levers to reduce test feedback time and test creation efforts. Additionally, we perform a small case study on RTS for manual end-to-end tests (see contribution for **G4**), where we find that if manual tests are imprecisely specified, this effectively leads to all tests being selected even for unrelated changes [33].

The contributions made by this publication-based dissertation have previously appeared in peer-reviewed publications. These are listed below and visualized in Figure 1.2, where we associate each publication to the research gaps it addresses, again grouped into system and process characteristics:

P1 **Daniel Elsner**, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. *Build System Aware Multi-language Regression Test Selection in Continuous Integration.* Proceedings of the International Conference on Software Engineering: Software Engineering in Practice, pages 87–96, 2022

P2 **Daniel Elsner**, Severin Kacianka, Stephan Lipp, Alexander Pretschner, Axel Habermann, Maria Graber, and Silke Reimer. *BinaryRTS: Cross-language Regression Test Selection for C++ Binaries in CI.* Proceedings of the International Conference on Software Testing, Verification and Validation, pages 327–338, 2023

P3 **Daniel Elsner**, Florian Hauer, Alexander Pretschner, and Silke Reimer. *Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration.* Proceedings of the International Symposium on Software Testing and Analysis, pages 491–504, 2021

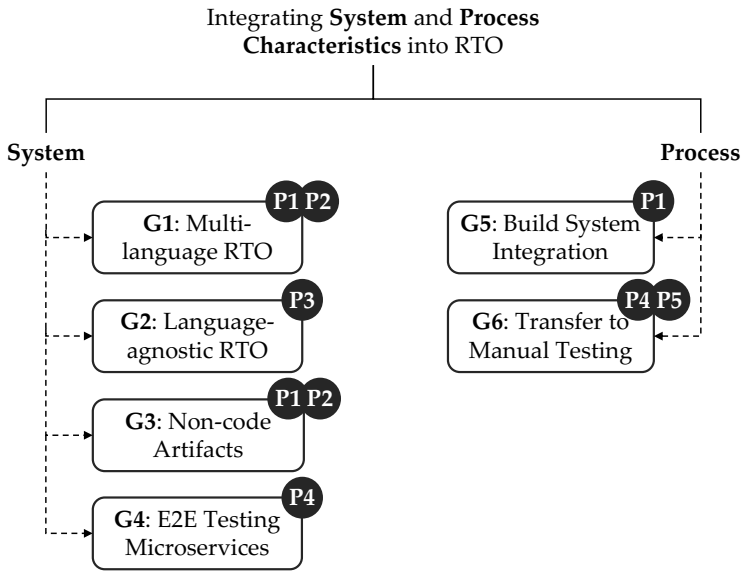


Figure 1.2.: Research gaps and associated publications that constitute the contributions of this doctoral dissertation

P4 **Daniel Elsner**, Daniel Bertagnolli, Alexander Pretschner, and Rudi Klaus. *Challenges in Regression Test Selection for End-to-End Testing of Microservice-based Software Systems.* Proceedings of the International Conference on Automation of Software Test, pages 1–5, 2022‡

P5 Roman Haas*, **Daniel Elsner***, Elmar Juergens, Alexander Pretschner, and Sven Apel. *How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies.* Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1281–1291, 2021

In addition to these publications, the author of this thesis has co-authored further papers which tackle related problems and have thus contributed to the idea of this thesis, but are not explicitly included:

- **Daniel Elsner**, Roland Wuersching, Markus Schnappinger, Alexander Pretschner. *Probe-based Syscall Tracing for Efficient and Practical File-level Test Traces.* Proceedings of the International Conference on Automation of Software Test, pages 126–137, 2022
- Roland Wuersching*, **Daniel Elsner***, Fabian Leinen, Alexander Pretschner, Georg Grueneissl, Thomas Neumeyr, and Tobias Vosseler. *Severity-Aware Prioritization of System-Level Regression Tests in Automotive Software.* Proceedings of the International Conference on Software Testing, Verification and Validation, pages 398–409, 2023

‡Note that publication P4 is not a *core* publication in this publication-based thesis.
*Both authors contributed equally

- Simon Hundsdorfer*, **Daniel Elsner***, Alexander Pretschner. *DIRTS: Dependency Injection Aware Regression Test Selection.* Proceedings of the International Conference on Software Testing, Verification and Validation, pages 422–432, 2023

1.5. Outline

The outline for the remainder of this thesis is as follows: Chapter 2 provides relevant definitions and background on regression testing research and describes state-of-the-art RTO techniques. Chapters 3–7 present the methodological and technical solutions to fill the aforementioned research gaps. Existing work that is related to the contributions made by this thesis is discussed in Chapter 8. Finally, in Chapter 9, limitations of this thesis and future directions of research are discussed, and conclusions are drawn.

2. Background

This chapter establishes an understanding of the background of this thesis. It introduces fundamentals, definitions, and quality criteria related to regression testing and provides an overview over relevant techniques for optimizing regression testing from the literature. Parts of this chapter have appeared in peer-reviewed publications [33–37, 48, 58, 169], co-authored by the author of this thesis.

2.1. Foundations and Definitions

2.1.1. Software Testing

Software testing is a software development activity that intends to assess the quality of a SUT by checking if the code correctly "does what it was designed to do" [122, p. 2] and to detect failures[4]. Therefore, test cases are run against the SUT, where a set of test cases is called a *test suite*. Abstractly, a *test case* consists of a set of program inputs, certain execution conditions, and expected program outputs [59, 65]. After the test is executed, a *test report* is typically created that contains the *test verdicts* which reflects if a test case failed or passed [57, 59].

Software testing can be done at different levels of abstraction, namely the unit, integration, and system level [65]. In the thesis, we stick to the IEEE definitions [59], but definition boundaries between testing levels are not unequivocal and sometimes blurry in the literature [162]: *Unit testing* refers to testing of "software units or groups of related units" [59, p. 79]. Depending on the context of the system, a unit could be a function, class, component, or subsystem. *Integration testing* refers to testing combinations of units or components "to evaluate the interaction between them" [59, p. 41]. Testing at the integration level helps detecting defects in interfaces between potentially independently developed components. *System testing* refers to testing a "complete, integrated system to evaluate the system's compliance with its specified requirements" [59, p. 74]. Sometimes system testing is also referred to as *end-to-end testing*, as entire user scenarios can be tested in an end-to-end manner.

2.1.2. Defects, Faults, Errors, and Failures

Following Pretschner's [136] and Holling's [57] terminology, we use software *defect* as an umbrella term for faults, errors, or failures in this thesis. Accordingly, a *fault* is a semantically incorrect piece of code (commonly also referred to as *bug*). An *error* is provoked by a fault and depicts an incorrect internal program state. An error may in turn lead to a *failure* which reflects an observable difference between the expected and actual program

[4] While this description applies to dynamic, functional software testing, which is the subject of this thesis, there are other forms of testing that statically or dynamically examine (non-)functional quality aspects.

behavior (cf. [85]). A failing test case reveals a failure, hence, we will refer to them as *test failures*.

2.1.3. Regression Testing

In evolving software systems, *regression testing* involves testing the program after it has been modified to "re-establish our confidence that the program will perform according to the (possibly modified) specification" [91, p. 60]. The simplest regression testing strategy is to execute every test case in the regression test suite after each change, commonly referred to as *retest-all*. However, with large test suites, high testing frequency, or limited testing resources, *retest-all* is often too costly [172].

These outlined challenges have been subject to research on RTO at least since the late 1970s [42]. RTO can be categorized into three major research branches: TSM, RTS, and RTP. These share not only the objective to optimize the cost-effectiveness of regression testing, but also the proposed solution approaches exhibit overlap [172]. As this thesis contributes to RTS and RTP (and not TSM) research, we provide definitions of the underlying problems for RTS and RTP in the next two subsections. Within this document, we collectively refer to techniques addressing the defined problems as *RTO techniques* or *regression testing techniques*, which aligns with the terminology used by Yoo and Harman [172].

2.1.4. Regression Test Selection

The goal of RTS is to reduce testing effort by executing only a subset of the regression test suite during software evolution, while still selecting (ideally) all fault-exposing tests (see Section 2.2.1 for background on RTS quality criteria). More formally, given a program P, the modified version of P, P', and a test suite T, the test case selection problem is defined as follows [146, 172]:

> **Definition 2.1.1** (Test Case Selection Problem)**.** Select a subset of tests $T' \subseteq T$ to execute on P'.

T can be a test suite consisting of (1) existing test cases developed for testing P (*corrective* regression testing), as well as (2) new test cases to check the correctness of P' (*progressive* regression testing) [91, 172].

2.1.5. Regression Test Prioritization

The goal of RTP is to find the best permutation of the execution order of a test suite with respect to relevant properties, e.g., earliness of fault detection [172]. This way, RTP can increase the testing effectiveness, if regression testing is prematurely halted [148]. More formally, given a test suite T, the set of all permutations of T, PT, and a function f from PT to real numbers, $f : PT \to \mathbb{R}$, the test case prioritization problem is defined as follows [105, 148, 172]:

> **Definition 2.1.2** (Test Case Prioritization Problem)**.** Find a test suite ordering $T' \in PT$ such that $(\forall T'')(T'' \in PT \wedge T'' \neq T') \Rightarrow f(T') \geq f(T'')$.

The definition of f depends on context-specific goals of prioritization, such as early fault detection or meeting some code coverage criterion earlier [148]. In contrast with RTS, RTP is not necessarily performed after modifications have been introduced between two program versions, but can be performed at arbitrary points in time.

2.2. Quality Criteria

To assess the quality of an RTO technique, different criteria and evaluation metrics are used across research. We introduce the most relevant ones for RTS and RTP in the following.

2.2.1. Regression Test Selection

Rothermel and Harrold [143] propose a framework to evaluate RTS techniques consisting of *inclusiveness*, *precision*, *efficiency*, *generality*, and *accountability* [143]:

- *Inclusiveness* measures to what degree modification-revealing test cases, i.e., tests which produce a different output on the modified program, have been selected by the RTS technique. If all modification-revealing tests are selected, the technique is considered to be *safe*, as it implies that all tests that potentially expose a regression fault are safely selected.
- *Precision* measures how well a technique manages to exclude irrelevant test cases from the set of selected tests. If an RTS technique does not select any non-modification-revealing test cases, it is considered to be *precise*.
- *Efficiency* measures the involved space and time for applying the RTS technique. Thus, efficiency depends on the computational (analysis) effort required to calculate the test selection for the changes introduced between two program versions.
- *Generality* measures how well an RTS technique works on arbitrarily complex programming language constructs (e.g., procedures, object-oriented structures), and in different modification scenarios (e.g., change, deletion, addition).
- *Accountability* measures to what degree an RTS technique helps to maximize structural coverage[5] with selective testing, which "can aid in the evaluation of test suite adequacy" [143, p. 201].

RTS techniques typically consist of three phases: (1) the *analysis phase*, where the tests are selected, (2) the *execution phase*, which executes the selected tests, and (3) the *collection phase*, where relevant information for the next analysis phase is collected [46]. Since all phases may impact the efficiency of an RTS technique, Gligoric et al. [46] suggest to measure the *end-to-end time* comprising the execution time for all three phases. Since this reflects developers' perceived feedback time more realistically, several studies have reported end-to-end time [43, 45, 46, 87, 88, 179]. Notably, if RTS is used for pull requests[6]

[5] *Structural (code) coverage* refers to the percentage of the program's statements or expressions, i.e., structural elements, that were executed by a test suite [5].

[6] A *pull request* (or *merge request*) is a term often used in source code version control that refers to a set of changes that a developer requests to integrate or merge into the code base [37].

in a distributed VCS setting, the collection phase is sometimes excluded from the end-to-end time, as information for RTS is independently collected (i.e., *off-line* [46, 174], for example every night) and centrally provided to all pull requests, and thus does not affect feedback time [35, 37].

2.2.2. Regression Test Prioritization

The most common way to assess the quality of RTP techniques is the Average Percentage of Faults Detected (APFD) metric, where Rothermel et al. [149] and Elbaum et al. [30] were among the first to use it. Intuitively, the APFD metric measures how early in the testing process faults are detected by a prioritized test suite. Early detection of faults leads to shorter feedback time for developers who can then start fixing problems earlier. More formally, for a prioritized test suite T' of n test cases that reveals m faults, the APFD is defined as

$$\text{APFD} = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n} \tag{2.1}$$

where TF_i denotes the priority rank of the first test case that reveals the i-th fault [31, 149]. An APFD value close to 1 means that T' detects most faults rather early, whereas a value close to 0 means that T' detects most faults rather late. One caveat when applying the APFD metric is that information about detected *faults* is required. This is trivial if faults are seeded. However, in practice often only observed test *failures* are reported and a mapping from failures to faults is missing. Therefore, prior RTP research commonly assumes a one-to-one failure-to-fault mapping which essentially translates the APFD to the average percentage of detected *failures* [34, 112, 134, 155]. As the same fault can trigger multiple failures, this assumption can potentially distort evaluation results [134].

Since the severity of faults and test execution costs may significantly differ, Elbaum et al. [29] extended the APFD metric to incorporate varying test case and fault severity costs. The resulting Average Percentage of Faults Detected per Cost (APFD_C) metric is defined as

$$\text{APFD}_C = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \tag{2.2}$$

where $t_1, t_2, ..., t_n$ reflect the test execution costs and $f_1, f_2, ..., f_m$ the fault severities [29]. In general, the *cost-aware* APFD_C metric is considered to be more realistic than the *cost-unaware* APFD metric and thus preferred in more recent research, albeit often with equal fault severities [18, 134]. The intuition behind the APFD and APFD_C metric is best illustrated by drawing the gain curve as shown in Figure 2.1, where the APFD represents the area under the gain curve. Consequently, for the APFD_C, the x-axis would be the *percentage of total test cost*, whereas the y-axis would be the *percentage of total fault severity*.

Notably, in capacity-constrained scenarios, where not all n test cases can be executed, a variant of the APFD called Normalized APFD can be used [9, 139, 158]. Furthermore, in addition to the APFD and APFD_C metric, prior research has assessed the quality of RTP techniques by the time required until detecting the first or last regression fault [18].

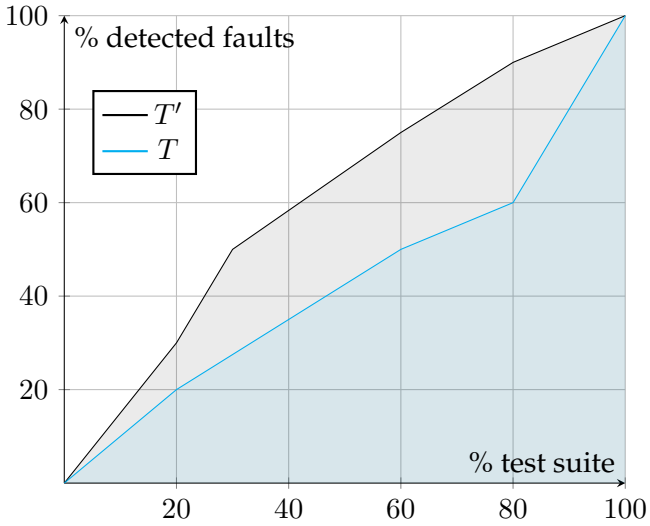


Figure 2.1.: Illustration of the intuition underlying the APFD evaluation metric; it reflects the area under the gain curve

2.3. Overview of Existing Techniques

Since the 1970s, RTO has been the subject of a large body of literature. Several literature reviews on RTO research exist, providing comprehensive and detailed overviews [1, 38, 68, 70, 102, 132, 141, 172]. The goal of this section is to establish an understanding of existing RTS and RTP techniques with a focus on more recent approaches, whereas we discuss work that is directly related to this thesis in Chapter 8.

2.3.1. Regression Test Selection

The rough intuition behind most RTS techniques is that for each test case a set of (code) entities is collected and if any of these has changed, all tests depending on the modified entity are selected [174]. These per-test dependencies can either be collected through dynamic or static program analysis at different levels of granularity (e.g., basic blocks [52, 131, 147], functions [43, 175, 178], classes or files [16, 45, 46, 87, 88, 163], modules [156, 157, 163], or combinations thereof). In the case of dynamic program analysis, the per-test dependencies reflect the per-test execution trace, whereas static analysis can only approximate the run-time behavior of a test by considering all potentially executed entities [174]. As described in Section 2.2.1, if no test that can potentially expose a fault is excluded, the RTS technique is considered to be *safe*. RTS techniques that target *safe* selection require comprehensive program analysis information. *Unsafe* RTS techniques that rely on other types of information, such as test failure history, can be cheaper to apply in practice (e.g., no instrumentation overhead), but may potentially miss failures (and thus faults) as these techniques only approximate the set of relevant test cases [107]. In the following, we group existing RTS techniques into approaches that use (1) dynamic and (2) static program (or code) analysis, as well as (3) other information sources for test selection. For each group, we explain one representative RTS technique in more detail and list similar techniques and studies.

Dynamic Program Analysis

To collect per-test dependencies at run-time, the execution of the SUT needs to be monitored. Therefore, dynamic RTS techniques harness various dynamic program analysis techniques. These instrument the SUT by adding analysis code that is executed as part of the normal test execution [12, 127]. For instance, by instrumenting each function, a per-test execution trace can be obtained, which contains all the functions executed by a test. Dynamic program analysis techniques add instrumentation code at the source code, intermediate, binary, or system level with different trade-offs regarding performance and transferability between platforms, programming languages, and compilers [127].

In the following, we explain EKSTAZI, a dynamic RTS technique for the JVM proposed by Gligoric et al. [45, 46] in 2015. The reason why we pick EKSTAZI is that it is one of the few RTS techniques adopted in practice [45, 46] and because the underlying concepts have influenced the development of a build system aware multi-language RTS technique as part of this thesis [37] (see Chapter 3). EKSTAZI operates at the granularity level of files, meaning that it identifies all files a test depends on during execution. Therefore, EKSTAZI employs lightweight instrumentation of Java bytecode to track (1) implicitly accessed files (`.class` files containing executed Java bytecode) and (2) explicitly accessed files (through Java standard library methods) for each test [45]. For each test, the set of accessed files is stored together with each file's checksum, where EKSTAZI optimizes the checksum computation by ignoring debug information from Java bytecode. After a developer introduces a change, EKSTAZI compares all previously collected file checksums to the current checksums and selects those tests that depend on a changed file. EKSTAZI is thereby safe for changes to code files and, conceptually, also for external (non-code) files. Yet, safety violations were found for EKSTAZI in case of non-code changes by RTSCHECK, a framework for checking the correctness of RTS tools [179]. In an empirical study on 32 open-source projects, Gligoric et al. [46] show that EKSTAZI's file-level dependency granularity is more efficient than method-level RTS for the studied Java projects.

EKSTAZI has further been transferred to .NET projects, where the resulting technique EKSTAZI# has been shown to significantly reduce testing effort compared to module-level RTS [163]. The idea of dynamic, file-level RTS was also adapted by RTSLINUX [16], which goes beyond JVM boundaries by capturing files opened through system calls on Linux. Unlike EKSTAZI, RTSLINUX instruments the system call interface and thus implements dynamic analysis at the operating system level rather than the Java bytecode level.

Regarding more fine-grained dynamic RTS, FAULTTRACER was proposed as a method-level technique for Java in 2013, albeit primarily focusing on fault localization for affected failing tests [175]. A more recent dynamic technique for function-level RTS is TESTSAGE presented in 2019, which instruments functions at the intermediate code representation of the LLVM compiler infrastructure [99] and has been deployed at Google for testing large-scale C++ web services [178].

Finally, Zhang [174] has proposed HYRTS, a dynamic RTS technique for the JVM that combines file- and method-level analysis. Thereby, HYRTS achieves better precision when only parts of a file are changed (i.e., for method-level changes), whereas analysis overhead is still low in all other cases, e.g., file additions or deletions.

To our knowledge, open-source tools implementing dynamic RTS techniques from research are EKSTAZI [45], HYRTS [174], and FAULTTRACER [175] for Java, as well as EKSTAZI# [163] for .NET, and PYTEST-RTS [67] for Python.

Static Program Analysis

Static RTS techniques collect per-test dependencies by analyzing source code [130], intermediate representations such as Java bytecode or LLVM bitcode [43, 87, 88], or binary code [15] without executing the tests.

In the following, we explain RTS++, a static RTS technique for C++ proposed by Fu et al. [43] in 2019. RTS++ follows the concept of a *(class) firewall* [77, 90] which has influenced several static RTS techniques [60, 87, 88, 131, 165], and—as it also targets C++ software—is related to BINARYRTS, a hybrid function- and file-level dynamic RTS technique for C++ binaries developed as part of this thesis [35] (see Chapter 4). RTS++ operates at the function level and collects the set of functions that each test depends on by traversing the call graph of an LLVM bitcode file. For each function, RTS++ stores a checksum of its body into a so-called *annotated dependency graph* [43]. After a developer makes a change, RTS++ constructs the dependency graph on the new program version and compares all checksums to the graph of the previous version to find changed functions (this is slightly simplified, RTS++ also considers `virtual` functions) [43]. RTS++ then selects all potentially affected tests that transitively depend on any changed function. This transitive closure of affected functions can also be considered as the *firewall* [77]. According to Fu et al. [43], RTS++ is safe for changes in C++ code except for changes to non-primitive global variables, function pointers, or `setjump/longjump`.

Also building on the idea of the (class) firewall, Legunsen et al. [87] proposed two RTS techniques for method- and class-level test selection in Java programs. As the class-level approach performed better in empirical studies, the authors later adapted the approach and embodied it in the RTS tool STARTS [88]. While naturally being less precise due to the employed static analysis, STARTS achieved comparable performance to EKSTAZI in 22 open-source projects [87].

To our knowledge, open-source tools implementing static RTS techniques from research are STARTS [88], AUTORTS [130], and DIRTS [58] for Java, and SELFECTION for C projects that compile to ARM ELF binaries [15].

Other Information Sources

Several unsafe RTS techniques exist that embed information from other sources than program analysis into rule-based heuristics or machine-learned models to approximate the set of relevant test cases [107]. Despite the lack of deterministic safety guarantees, unsafe RTS techniques have been widely applied in industry [2, 4, 28, 32, 72, 89, 107, 123, 135, 158]. Notably, these techniques typically rank tests by their likelihood to fail and then select only a subset of the test suite based on some cut-off criterion (e.g., time constraints) [34, 158]. In this section, we only discuss techniques that have explicitly been used for test selection, but—since any RTP technique represents a test ranking model—the RTP approaches discussed in the next section (see Section 2.3.2) naturally exhibit overlap.

Here, we describe the *predictive test selection* approach presented by Machalica et al. [107] in 2019. We pick this RTS technique as it has actually been deployed at industry-scale in Facebook's CI infrastructure, uses a variety of information sources other than program analysis, and incorporates several ideas from other prior RTS and RTP approaches as well as defect prediction research [14, 28, 32, 56, 158, 182]. To predict the failure of a test

suite for a given changeset, the authors train an ML model on features from CI and VCS metadata and static build dependencies. Their best-performing model uses a feature set containing information about (1) the file extensions inside the changeset, (2) the change history for the files in the changeset, (3) historical failure rates of the test suite, (4) the project name (could point to project breakage patterns), (5) the number of tests in the test suite, and (6) the minimal distance in the build dependency graph between any of the files in the changeset and the test suite [107]. Machalica et al. [107] further report that flaky tests, i.e., tests that non-deterministically pass and fail on the same code, can impact the training and evaluation of unsafe RTS. We briefly discuss the impact of flaky tests on RTO in Section 9.2.

Notably, the idea to use information about failure history to rank tests, as opposed to only relying on program analysis, was first proposed by Kim and Porter in 2002 [71] and was later studied for unsafe RTS in an industry-scale CI environment at Google by Elbaum et al. [32] and Spieker et al. [158].

To our knowledge, there are no open-source tools implementing any of the unsafe RTS techniques from research yet.

2.3.2. Regression Test Prioritization

In 1999, Rothermel et al. [149] formally defined and first studied the problem of RTP in isolation [149], whereas Wong et al. [168] had previously experimented with prioritizing test cases after applying RTS. The intuition behind most RTP techniques is that tests are ranked based on some *surrogate* property in order to detect faults earlier in the testing process [172]. If the test ordering achieves this goal of earlier fault detection, developers receive feedback faster and can start debugging failures earlier. A commonly applied baseline strategy for RTP is random ordering of tests [171]. As with the RTS techniques described before, we group the RTP approaches by the type of information they require, namely (1) dynamic, (2) static, or (3) other types of information.

To our knowledge, there are no open-source and ready-to-use RTP tools that implement the proposed RTP approaches at the time of writing; however, some studies have published supplemental material and artifacts that could serve as a starting point for implementing RTP tools.

Dynamic Program Analysis

Early RTP techniques greedily prioritize test cases based on total or additional code coverage at different granularity levels (e.g., statement, branch, function) [30, 31]. The assumption behind these coverage-based surrogates is that early maximization of structural coverage correlates with early maximization of fault detection, the objective of RTP [172]. On top of coverage-based RTP approaches, Rothermel et al. [148] propose to use mutation analysis [22, 49] to evaluate each test's fault-exposing potential and prioritize tests according to this potential [25].

Besides greedy prioritization algorithms, several search-based algorithms have been used to implement dynamic RTP approaches [39, 95, 164]. These techniques formulate the RTP problem as a multi-objective optimization problem, which they solve using meta-heuristic search algorithms (e.g., genetic algorithms). Examples for competing objective

functions are the minimization of test execution time or test resource costs, and the maximization of code coverage or non-code (e.g., requirements) coverage, whereas optimization constraints can be test precedence or dependence [51]. Notably, even though most existing studies on search-based RTP use coverage information, dynamic program analysis is not required per se.

Another RTP technique utilizing dynamic program information is adaptive random test case prioritization [61], which transfers the idea of adaptive random testing [19] to RTP. Hereby, test cases are iteratively selected until the final test order is created. More specifically, a test is selected from the set of available test cases based on its distance to the set of already selected test cases. For instance, the next selected test can have the largest minimum pair-wise distance to the already selected tests [103]. To compute the pair-wise distance between two tests, the Jaccard distance[7] between the sets of statements covered by each test has been used in prior studies [61, 103–105].

Static Program Analysis

Since the involved dynamic program analysis to collect coverage information for RTP can be expensive, several more lightweight RTP techniques have been proposed that use statically obtained information [86, 112, 113, 119, 134, 152, 176].

Zhang et al. [176] and Mei et al. [119] utilize Control-Flow Graph (CFG) information of test cases to implement method- and statement-level RTP approaches, respectively. The idea behind using CFGs is that a test which statically depends on more statements or methods than another test has a higher *testing ability* (i.e., can cover more code) and should thus be prioritized [176]. Analogous to the greedy dynamic RTP techniques, a *total* and an *additional* approach exist, where the former simply prioritizes tests by their testing ability and the latter prioritizes tests by testing ability, but excluding methods (or statements) already covered by prior tests [105].

Ledru et al. [86] propose to use *string edit distances* to estimate test case similarity and prioritize tests that are most dissimilar first. Hereby, the underlying hypothesis is that textually different test cases also execute different parts of the code and therefore can reveal different failures [105]. To compute the pair-wise string-distance between two tests, prior work shows that the Manhattan (L_1) distance function[8] works especially well [86].

More recently, Peng et al. [134] and Mattis et al. [112, 113] have studied RTP using Information Retrieval (IR) approaches, which was originally proposed by Saha et al. [152] in 2015. IR-based RTP techniques rank tests according to their textual similarity to the code *diff*, i.e., the modified code parts in evolving software [134]. To compute the textual similarity between the code diff and the tests, various text vectorization algorithms from natural language processing have been applied [134]. Although IR-based RTP techniques can only approximate (run-time) relationships between tests and code under test—which ideally resembles the per-test coverage information used by dynamic RTP techniques—, IR-based techniques have been shown to outperform both, RTP approaches using dynamic and static program analysis [134, 152]. Notably, IR-based RTP techniques are practical and efficient: while they do need source code access, they can be designed to be (mostly) language-agnostic and do not require any program instrumentation [152].

[7] The *Jaccard distance* between two sets A and B is defined as $1 - \frac{|A \cap B|}{|A \cup B|}$ [61].

[8] The *Manhattan* or L_1 *distance* between two strings x and y of length n is defined as $\sum_{i=1}^{n} |x_i - y_i|$ (if strings have different lengths, the shorter string is zero-padded) [86].

Other Information Sources

The often high cost of RTP approaches based on dynamic analysis and the imprecision of those techniques using static analysis has inspired several RTP techniques that utilize other types of information [9, 14, 32, 71, 109, 118, 123, 158, 170].

Kim and Porter [71] were among the first to propose RTP (and unsafe RTS) based on test failure history in resource constrained environments. They rank tests using exponential smoothing of historical failure frequency, where more recent test failures are weighted higher than older failures [71]. The idea of using test failure history to rank tests has inspired the development of several other surrogates or heuristics for RTP. Elbaum et al. [32] first applied history-based RTP at industry-scale in Google's CI environment. They prioritize tests by the last time they have failed, thus favoring more recently failed tests. The underlying hypothesis is that tests which have previously failed are "proven performers" [51, p. 3] and execute error-prone code. Several studies have reported successful results when using history-based RTP techniques based on test failure history [109, 123, 158]. Notably, more recent research has found that the presence of flaky tests might impede history-based RTP approaches [40, 89].

In addition to test failure history, various other information sources have been employed, e.g., VCS metadata such as the number of distinct authors [89, 118]. More recent RTP techniques incorporate multiple information sources (e.g., dynamic or static program information, test history, VCS metadata) and train machine-learned test ranking models on these data [9, 14, 170]. While these ML models are reportedly accurate, they are more expensive to develop and maintain than the rather simple rule-based heuristics, and naturally harder to interpret by developers.

Part II.

Methodological and Technical Solutions

3. Build System Aware Multi-language Regression Test Selection in Continuous Integration

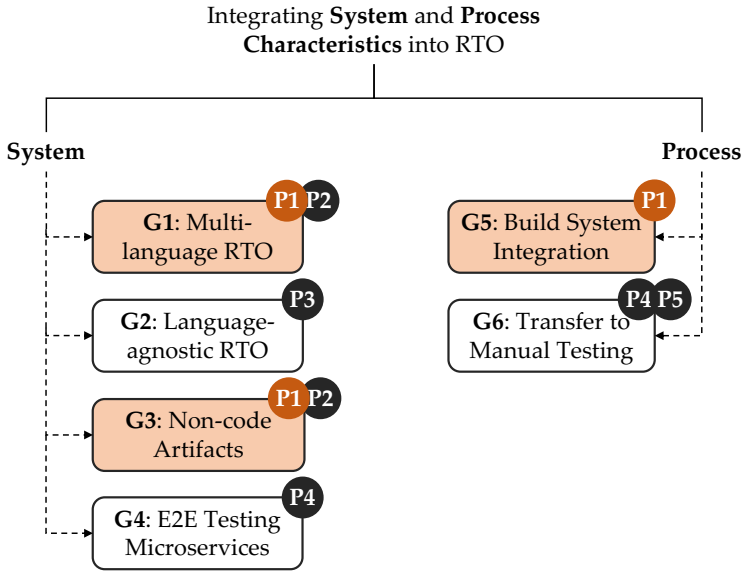


Figure 3.1.: Big picture of research gaps addressed by this publication (P1)

N.B.: Parts of the following summarizing paragraphs have been adopted from [37]. Due to the obvious content overlapping, quotes are not marked explicitly.

Summary This paper presents a novel RTS technique for the large-scale multi-language code base and CI environment of our industry partner IVU. In contrast with existing RTS approaches, the proposed technique is aware of changes to the build system configuration and to source code files in different languages. The RTS technique selectively (1) compiles code modules and (2) executes regression tests for pull requests in CI pipelines and has been deployed for all release branches at IVU since September 2021.

- *Problem:* At IVU, compiling the entire code base and running thousands of regression tests for each pull request results in intolerable CI feedback times of up to several hours and is therefore not economically feasible. However, unsafe RTS is not appropriate in this context, since pull requests to release branches may be part of support patches that are built directly from those branches every day; this implies that as few regressions as possible should slip into release branches.

- *Gap:* Existing RTS techniques (1) require a fully built workspace or are unsafe in case of changes to the build system configuration, and (2) are impractical if regression testing is performed across multiple languages and makes use of non-code artifacts.
- *Solution:* We present a build system aware RTS technique which harnesses probe-based system call tracing (using `DTrace` [47]), Java class loader instrumentation, and static analysis to collect file-level per-test dependencies across language boundaries. Using this dependency information, our technique presents a novel algorithm for selectively building code modules and running tests to verify the correctness of pull requests in CI pipelines.
- *Contribution:* Our empirical study across 397 pull requests including roughly 2,700 commits shows that our technique safely excludes up to 75% of tests on average (no undetected real failures slip into the target branches). End-to-end CI pipeline time, including stages for building, analyzing, and testing a pull request, is reduced by up to 63% on average. We further show that RTS performs significantly better in pull requests on a maintenance release branch than one with active development.
- *Limitations:* In case of changes to (binary-compiled) Dynamic-link Libraries (DLLs), our RTS technique is imprecise, as it selects all tests that open the DLL, even if the change does not affect the C/C++ code the test actually covers. Another limitation is that there can be safety violations if the used per-test traces are outdated (currently collected once per day) or in case of changes related to dependency injection mechanisms.

Author Contributions D. Elsner and his supervisor A. Pretschner conceived and defined the initial problem statement. The idea of the paper was then discussed with IVU engineers M. Graber, R. Dammer, and S. Reimer. The theoretical solution was developed by D. Elsner, who discussed the transfer into a technical solution with R. Wuersching and M. Schnappinger. During the implementation D. Elsner was assisted by R. Wuersching. The empirical study and the corresponding data analyses were carried out by D. Elsner and the results were discussed with M. Graber, R. Dammer, and S. Reimer. The manuscript was drafted primarily by D. Elsner in consultation with M. Schnappinger and A. Pretschner. All authors reviewed later drafts of the paper.

4. BinaryRTS: Cross-language Regression Test Selection for C++ Binaries in CI

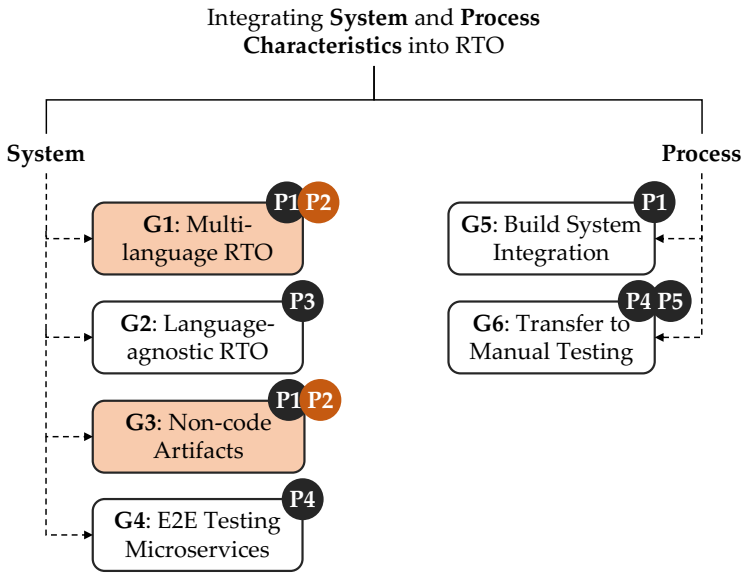


Figure 4.1.: Big picture of research gaps addressed by this publication (P2)

N.B.: Parts of the following summarizing paragraphs have been adopted from [35]. Due to the obvious content overlapping, quotes are not marked explicitly.

Summary This paper presents BINARYRTS, a novel RTS technique for software which makes use of (1) cross-language links to compiled C++ binaries and (2) non-code artifacts during regression testing. In order to be transferable to both, binary executables and DLLs, BINARYRTS is the first RTS technique that employs dynamic binary instrumentation to collect covered functions and accessed files for each test. We release BINARYRTS as a publicly available RTS tool for software involving C++ code; it has been deployed at IVU since November 2022.

- *Problem:* From our previous work on RTS at IVU [37] (see Chapter 3), we find that the problem of imprecise file-level test selection in case of changes to binary DLLs remains unsolved both for Java and C++ tests. Contemporary work on safe RTS for C++ software is sparse, though, and unsafe RTS is unsuitable for pull requests to release branches at IVU.
- *Gap:* Existing safe RTS techniques for modern C++ software (1) target LLVM-based C++ projects, (2) do not cope with cross-language links to C++ binaries, (3) ignore

changes to non-code artifacts, and (4) either do not support dynamic linking of libraries or operating systems other than Linux [43, 178].

- *Solution:* We present BINARYRTS, a novel RTS technique and tool which uses dynamic binary instrumentation to collect per-test information in the form of covered functions and accessed files. This way, tests from any language with interoperability to C++ binaries can be analyzed, e.g., Java via Java Native Interface (JNI).
- *Contribution:* Our empirical study in IVU's CI infrastructure across 385 pull requests indicates that BINARYRTS safely selects on average 26%–37% of C++ tests compared to *retest-all* (and roughly 50% of the tests selected by module-level C++ RTS), and 57%–64% of Java tests compared to our file-level RTS technique from [37]. The test duration for C++ is thereby on average reduced by up to 68% compared to *retest-all* and up to 45% compared to module-level RTS (the previous strategy at IVU). BINARYRTS is compiler-agnostic, supports C and C++ binaries on Windows and Linux out-of-the-box, integrates well with established Java and C++ testing infrastructure, and can be transferred to other compiled languages.
- *Limitations:* When collecting per-test traces using dynamic binary instrumentation, the run time overhead can be substantial. While this does not influence the perceived developer feedback time in pull request CI pipelines, as traces for release branches are obtained during off-peak hours (currently every night), it can potentially cause safety violations if the used per-test traces are outdated. Another potential source of unsafe RTS behavior is the analysis of changes to non-functional entities (e.g., macros, global variables), as we limit the analysis scope since it may be impractical to always scan the entire—potentially immense—code base for changed entities.

Author Contributions D. Elsner and his supervisor A. Pretschner conceived and defined the initial problem statement. The idea of the paper was then discussed with IVU engineers A. Habermann, M. Graber, and S. Reimer. The theoretical solution was developed by D. Elsner, who discussed the transfer into a technical solution with S. Kacianka and S. Lipp. D. Elsner implemented the tool BINARYRTS with minor assistance by S. Kacianka and A. Habermann. The empirical study and the corresponding data analyses were carried out by D. Elsner and the results were discussed with A. Habermann, M. Graber, and S. Reimer. The manuscript was drafted primarily by D. Elsner in consultation with S. Kacianka, S. Lipp, and A. Pretschner. All authors reviewed later drafts of the paper.

5. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration

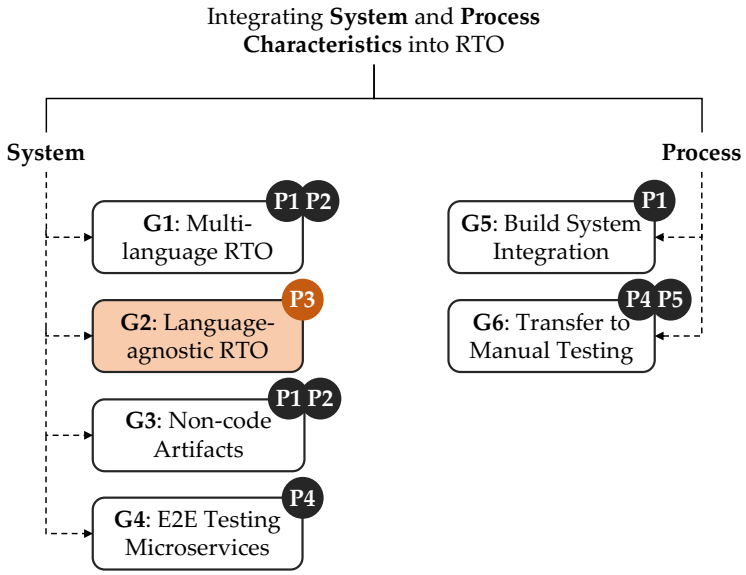


Figure 5.1.: Big picture of research gaps addressed by this publication (P3)

N.B.: Parts of the following summarizing paragraphs have been adopted from [34]. Due to the obvious content overlapping, quotes are not marked explicitly.

Summary This paper presents a methodology and a large-scale empirical study to comparatively evaluate the cost-effectiveness of RTO approaches from prior research that rely exclusively on readily available information from CI and VCSs. Because these language- and platform-agnostic metadata are typically collected in CI environments, such RTO approaches are easily applicable and transferable. By applying the developed methodology and insights from this study, practitioners will be able to build cost-effective RTP or unsafe RTS approaches suitable to their context.

- *Problem:* Effective traditional RTP and safe RTS techniques typically rely on language-specific dynamic or static program analysis which is often too costly in large-scale CI testing environments. Besides, these techniques are impractical if regression tests operate across language boundaries.

- *Gap:* Existing works have developed lightweight, less intrusive RTP and unsafe RTS techniques. However, they often include context-specific information, making them difficult to transfer and compare. In addition, results in the literature are equivocal about cost-effectiveness trade-offs and their sensitivity, and lack empirical guidance on how practitioners should calibrate these techniques.
- *Solution:* We propose a methodology that provides a generic process for exploiting readily available CI and VCS metadata to (1) build RTP and unsafe RTS approaches from prior research and (2) evaluate their cost-effectiveness sensitivity.
- *Contribution:* We conduct a large-scale empirical study on 23 open-source and industrial projects and find that approaches from our methodology save on average 84% of test time while detecting 90% of the failures for unsafe RTS and significantly outperform RTP baselines. We also apply our methodology at IVU, where it saves 20% of testing time while detecting 93% of failures. In addition, we derive practical guidelines: (1) it can be beneficial to limit training data, (2) features on test history work particularly well (but might be biased by flaky tests depending on the project [40, 89]), and (3) simple heuristics often outperform ML models.
- *Limitations:* By nature of empirical studies, we cannot easily generalize our findings beyond the studied projects, yet, the methodology is expected to work in most CI environments. The methodology does, however, neither include IR-based or Reinforcement Learning (RL) ranking models, nor do we perform hyperparameter optimization, which could lead to better test prioritization [9, 134, 170]. Our empirical results are further limited by the fact that we assume a one-to-one failure-to-fault mapping and do not explicitly consider flaky tests.

Author Contributions D. Elsner conceived and defined the initial problem statement. The idea of the paper was then discussed with F. Hauer and A. Pretschner, and later also with IVU engineer S. Reimer. The methodology and empirical analyses were developed and implemented by D. Elsner, who then discussed the results with F. Hauer and S. Reimer. The manuscript was drafted primarily by D. Elsner in consultation with F. Hauer and A. Pretschner. All authors reviewed later drafts of the paper.

6. Challenges in Regression Test Selection for End-to-End Testing of Microservice-based Software Systems

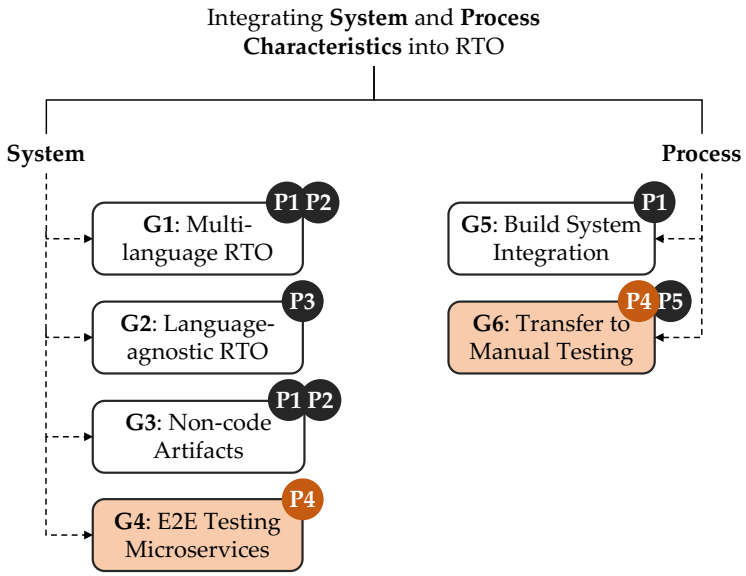


Figure 6.1.: Big picture of research gaps addressed by this publication (P4)

N.B.: Parts of the following summarizing paragraphs have been adopted from [33]. Due to the obvious content overlapping, quotes are not marked explicitly.

Summary This paper presents the dynamic RTS technique MICRORTS which targets microservice-based systems. We describe challenges in designing it and applying it to (manual) end-to-end testing in a case study on the German COVID-19 contact tracing application, a microservice system. MICRORTS uses the polyglot observability and distributed tracing infrastructure OpenTelemetry[9] to collect end-to-end traces; hence, the concept underlying MICRORTS can be used to implement RTS in distributed systems with different communication protocols and programming languages.

- *Problem:* Although distributed systems are particularly costly and time-intensive to test at the integration and system level, most RTS techniques focus on single-process systems and are thus not suitable for these systems as their analysis is limited to a single process.

[9]OpenTelemetry: `https://opentelemetry.io`

- *Gap:* There are a few RTS studies on web applications and services, but they (1) instrument server code, which is incomplete for end-to-end testing of microservices where clients may contain extensive business logic and orchestrate requests to multiple services, (2) are limited to a single language and network communication protocol, (3) lack analysis of the instrumentation overhead and the impact of the collection granularity for per-test dependencies, and (4) do not investigate the context of (manual) end-to-end tests in microservice-based systems.
- *Solution:* We present MICRORTS, a dynamic RTS technique that combines distributed tracing infrastructure and custom Java bytecode instrumentation to address end-to-end testing in microservice-based systems.
- *Contribution:* We highlight challenges related to instrumentation granularity and overhead by conducting experiments on a microservice benchmark and find that most overhead occurs at service startup and is mainly caused by the distributed tracing framework. Furthermore, we elaborate on experiences when applying MICRORTS in a case study on a (manual) end-to-end test suite in a real-world context. The results of the case study indicate that, in line with previous findings [66], if manual tests are under-specified, MICRORTS cannot provide benefits over retest-all. However, through semi-automated filtering of test traces, we can reduce the number of selected tests by 10%–50%, albeit only when using method-level traces.
- *Limitations:* MICRORTS is currently limited to Java microservices and Android clients, as these were the subjects that the case study was conducted on. In addition, the size of the case study on the COVID-19 application is relatively small, covering only 20 manual end-to-end tests and 12 software versions.

Author Contributions D. Elsner and his supervisor A. Pretschner conceived and defined the initial problem statement. The idea and theoretical solution was then further developed by D. Bertagnolli and D. Elsner and later discussed with R. Klaus from T-Systems. During the implementation of the technical solution, D. Bertagnolli was supported by D. Elsner. The case study and the corresponding data analyses were carried out by D. Bertagnolli, again with support by D. Elsner, and the results were discussed with D. Elsner and R. Klaus. The manuscript was drafted primarily by D. Elsner in consultation with A. Pretschner. All authors reviewed later drafts of the paper.

7. How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies

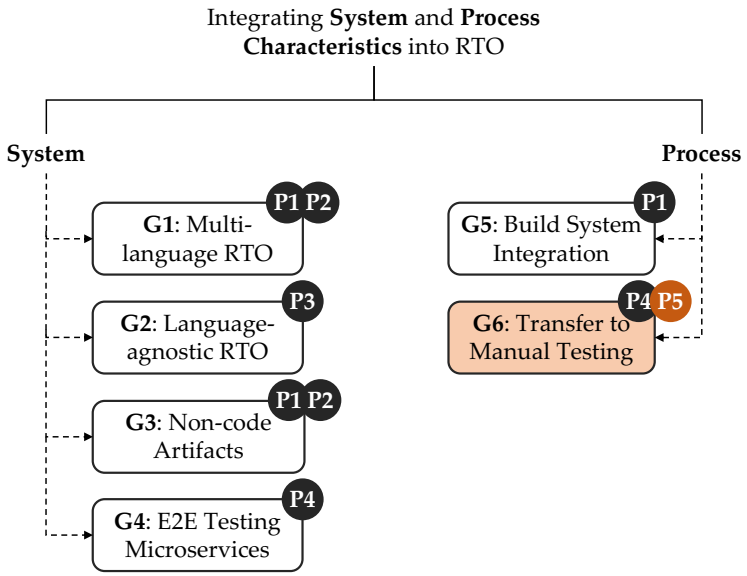


Figure 7.1.: Big picture of research gaps addressed by this publication (P5)

N.B.: Parts of the following summarizing paragraphs have been adopted from [48]. Due to the obvious content overlapping, quotes are not marked explicitly.

Summary Although manual (regression) testing is widely applied in industry, research for optimizing testing mainly focuses on techniques for automated tests without transferring them to manual testing. This paper presents a developer survey on the state-of-practice in manual testing and derives optimization guidelines for respective processes. Using these guidelines, practitioners are able to identify and exploit optimization potential in their manual testing processes. The study was conducted in cooperation with our industry partner CQSE.

- *Problem:* To improve (regression) testing cost-effectiveness, there has been significant research effort, yet mainly focusing on automated tests. However, transfer of these techniques to manual testing is often difficult as required data from program analysis or version control and CI systems are missing for manual testing processes.

- *Gap:* It is unclear how readily available data can be leveraged to transfer and implement test optimization techniques, such as RTO approaches, for manual testing, and how to integrate them into respective test processes.
- *Solution:* We survey 38 test practitioners from 16 companies to capture the state-of-practice in manual testing including its prevalence, characteristics, problems, and optimization potential. By combining the findings from the survey with optimization techniques from the literature, we synthesize optimization guidelines into an annotated manual software testing process model. This model summarizes the prerequisites and caveats of optimization techniques and highlights potential integration points.
- *Contribution:* We provide evidence that manual testing is widely employed and will be in foreseeable future, which stresses the importance of transferring optimization techniques. The optimization guidelines derived from our survey and the literature can help practitioners to optimize their manual testing processes. Particularly relevant for this thesis, we conduct a case study at IVU to demonstrate the applicability of the optimization guidelines. In the case study, optimization potential related to RTP could be identified resulting in a reduction of feedback time.
- *Limitations:* The responses received in the survey might not be representative, as many context-specific factors influence manual testing processes. The scope of the case study is limited to a single industrial project but, by nature of case study research, is not meant to generalize. We have recently evaluated several black-box RTP techniques for manual system testing in another industry study co-authored by the author of this thesis [169].

Author Contributions D. Elsner and R. Haas conceived and defined the initial problem. The idea was then discussed with E. Juergens, A. Pretschner, and S. Apel. The survey was prepared and conducted by D. Elsner and R. Haas, who then derived the guidelines from the responses and incorporated them into the manual software testing process model. The process model was then improved in discussion with E. Juergens, A. Pretschner, and S. Apel. R. Haas conducted the first (acceptance testing) and D. Elsner the second (regression testing) case study. The results of the case studies were discussed by all authors. The manuscript was primarily drafted by D. Elsner and R. Haas. All authors reviewed subsequent drafts of the paper.

Part III.

Related Work and Conclusion

8. Related Work

This chapter discusses related work and highlights the research gaps addressed by this thesis. Parts of this chapter have appeared in peer-reviewed publications [33–37, 48, 58, 169], co-authored by the author of this thesis.

In the following, we discuss related works by structuring them according to the system and process characteristics addressed by this thesis (see Section 1.1 and Figure 1.1). Alongside, we describe the research gaps introduced in Section 1.2 and visualized in Figure 1.2.

8.1. System Characteristics Affecting Regression Testing

This thesis addresses system characteristics that we group into the system-related context factors *size*, *complexity*, and *type* in line with prior regression testing research [1] (see Section 1.1).

Regarding *size*, there has been notable research progress in developing RTO techniques for large, industrial software systems in the past two decades. While this thesis evaluates the developed techniques in large-scale software, we do not address a specific research gap related to the size of a system, but merely contribute to the existing body of literature on applying RTO in large-scale systems (e.g., [14, 32, 89, 107, 123, 135, 178, 181]).

8.1.1. Multilingual Software

There are two lines of work that investigate RTO solutions for multilingual software: (1) techniques that use program analysis approaches which work across language boundaries and (2) techniques that use entirely language- and platform-agnostic information.

For (1), we identify the following relevant studies (all targeting RTS): Celik et al. [16] propose RTSLinux, an RTS technique for Linux systems that uses system call interception to trace the files that are accessed by a test running in a JVM. Since the system call interception code runs at the level of the operating system process, their analysis naturally goes beyond JVM boundaries: RTSLinux can also observe calls to native code via the JNI and accesses to other non-Java files, even from transitive child processes. Besides, RTSLinux is generally applicable to non-JVM projects, albeit with low precision for interpreted or binary-compiled languages [16]. Similar to Ekstazi (see Section 2.3.1), RTSLinux uses file checksums for selecting tests. In a study on 21 open-source Java projects, RTSLinux skipped 74% of tests and saved 53% of test execution time compared to retest-all. Despite these results, RTSLinux has a few limitations in practice: RTSLinux relies on a modification of the Linux kernel which is hardly transferable to closed-source operating systems such as Windows [37], and may raise maintainability (e.g., kernel releases may break it) or security (e.g., risk of corrupting the kernel) concerns [36]. Since RTSLinux is not publicly available, transferability is further limited. In

addition, RTSLINUX is imprecise for changes in Java Archives (JARs) or DLLs, since a change in any parts of those archives and binaries will lead to a selection of all tests that depend on the JAR or DLL. Other RTS techniques that potentially suit multi-language software use program analysis approaches that operate on intermediate code representations such as Java bytecode [45, 46, 88] or LLVM bitcode [43, 178]. These RTS techniques are (possibly) applicable to multi-language code bases with various source languages that compile to the same intermediate representation, e.g., Java and Scala to Java bytecode or Rust and C++ to LLVM bitcode [43, 45]. Yet, except for EKSTAZI which has been applied to Scala and Java code bases [45], to our knowledge none of these RTS techniques has been applied to multi-language software.

In the second line of work (2), a much larger set of RTP and RTS techniques has been studied in the literature. These techniques use easily accessible information, such as test failure history from CI systems or VCS metadata (e.g., [2, 28, 32, 72, 78, 89, 96, 153, 158, 180]), as well as additional information, such as static or dynamic code or dependency analysis (e.g., [3, 4, 9, 14, 18, 55, 56, 71, 86, 107–110, 113, 120, 123, 128, 129, 134, 135, 150–152]). We deem the following works to be most relevant for this thesis: Elbaum et al. [32] are the first to apply RTP and unsafe RTS in large-scale CI environments at Google. Therefore, they use simple heuristics which prioritize tests that have recently failed. From their empirical study on a dataset of Google CI failures, they conclude that their heuristics are cost-effective for both, RTP and unsafe RTS. Notably, subsequent research [40, 89] has shown that flaky tests may have biased these empirical results, as the exclusive use of test failure history to prioritize tests can lead to an over-ranking of flaky tests that arbitrarily and frequently fail. Spieker et al. [158] are the first to apply RL to RTP and unsafe RTS to automatically learn to rank tests according to their failure likelihood. They also use the dataset of Google CI failures from Elbaum et al. [32] and enrich it with test histories from CI systems at ABB. Their RL-based technique operates on test failure history and achieves competitive performance to the simple RTP heuristics by Elbaum et al. [32]. More recent work by Bertolino et al. [9] has further underlined the effectiveness of RL-based RTP using additional information such as code complexity metrics on open-source Java projects. In an industrial study from 2016, Busjaeger and Xie [14] train an ML-based test ranking model on black- and white-box features obtained from Salesforce's VCS and CI system. The empirical results on three months of development history at Salesforce indicate that by executing only 3% of all ranked tests they still detect 75% of the test failures for unsafe RTS. While the analyzed code base is multilingual, some of the used information (e.g., code coverage) may not be available in other contexts and for different programming languages. Machalica et al. [107] develop an ML model to rank test targets at Facebook according to their likelihood to fail and use it for unsafe RTS. Feature are extracted from CI and VCS metadata as well as static build dependencies and project identifiers (see Section 2.3.1). The authors report a reduction of testing infrastructure cost by 50% and test executions by more than 66% while still detecting 95% of test failures in Facebook's multilingual code base. While the proposed RTS technique itself is language-agnostic, access to project- or organization-specific information (e.g., static build dependencies) is not guaranteed in arbitrary CI environments and may limit the technique's transferability. Leong et al. [89] propose an evaluation framework to evaluate RTS techniques in CI systems with respect to on test transitions rather than failures and incorporate test flakiness signals into the analysis. Using the framework, they simulate the performance of five lightweight heuristics for unsafe RTS at Google. These heuristics leverage metadata

from the version control and CI system on recent commits that impact tests (based on a test's transitive build dependencies), test execution history, and directory overlap of tests with changed files. The results show that flaky test failures can significantly impact the evaluation results of RTS techniques. Furthermore, heuristics using the number of recent commits that impact a test and the number of distinct commit authors give the best cost-effectiveness. To conclude, several studies have shown that language-agnostic RTP and (unsafe) RTS techniques based on various ML models or heuristics using various sources of information can be beneficial in different contexts. Thus, it is important to understand how sensitive the cost-effectiveness of these techniques is to the size, timeliness, and variety of data. Prior research investigates how much historical data is beneficial for RTO [2, 9, 158]. In a study conducted at Microsoft, cost-effectiveness of RTS was fluctuating over time [56]. Several other studies have investigated the influence of data variety by measuring the effectiveness of individual predictive features [3, 9, 14, 107]. However, these studies either focus on specific industrial contexts or use context-dependent information beyond widely available data such as test failure history.

Summary To summarize, we identify two research gaps (**G1** and **G2**) in the two lines of work on multilingual software.

For (1), we identify that most existing RTO techniques rely on language-specific information which is limited for multilingual software. The few existing RTS techniques that use program analysis which captures behavior beyond language boundaries are limited to projects running in the JVM or Linux systems and have not been applied to industry-scale multilingual software systems (⇒ **G1: Multi-language RTO**).

For (2), we identify that existing language-agnostic RTO techniques often include project- or organization-specific information that is not available in other CI contexts, making the techniques difficult to transfer and compare. We are not aware of prior work that uses only readily available information in CI environments to build RTP and unsafe RTS techniques, studies how sensitive their cost-effectiveness is to associated parameters, performs cost-aware evaluation on real-world failures from industrial and open-source projects, and derives empirical guidelines for calibrating the techniques in practice (⇒ **G2: Language-agnostic RTO**).

8.1.2. Configurable Software

Most traditional RTO techniques primarily focus on code changes, while non-code changes are largely ignored [125]. However, these changes can affect test outcomes as non-code artifacts are commonly used to configure the SUT, the test suite, or the test environment [125] and it has been shown that one third of commits in open-source software includes changes to non-code artifacts [10]. Therefore, previous research has developed various approaches to make RTO techniques aware of configuration-related changes to non-code artifacts in configurable software systems (e.g., [16, 20, 46, 50, 111, 125, 137, 138, 167]). From this line of research, we consider the following studies to be most relevant for this thesis.

Qu et al. [138] were the first to study configuration-aware regression testing in 2008. They study six releases of the `vim` program (from the Software Infrastructure Repository [24]) and find that the applied configuration strongly impacts the outcome of regression testing and that reordering configurations can improve the rate of fault detec-

tion. In a follow-up study from 2012, Qu et al. [137] developed a configuration selection approach, which selects configurations for regression testing a modified program version using change impact analysis. On two open-source and one industrial system, they demonstrate that their configuration selection approach identifies 15%–60% of configurations as redundant and saves 20%–55% of testing time compared to retest-all, while fault detection capability and code coverage are retained [137].

Cheng et al. [20] have recently studied RTP for configuration testing. Configuration testing depicts a testing technique for detecting misconfigurations before deploying a configuration in a production environment [160]. A configuration test (*ctest* [160]) verifies that the SUT behaves according to some desired behavior given a set of configuration parameters [20]. The authors evaluate their ctest-specific RTP approaches against traditional RTP techniques on five Java-based open-source projects and 66 real-world configuration change files. The empirical results indicate that their novel RTP techniques can significantly accelerate misconfiguration detection.

In 2011, Nanda et al. [125] presented the first RTS technique that is partially aware of non-code changes. More particularly, their approach tracks per-test dependencies to databases and configuration files to determine the set of affected tests in the presence of configuration- and database-related changes. Therefore, they instrument the Java property and the Java database Application Programming Interface (API). Their preliminary empirical results on 5 and 7 versions of two software systems indicate that their RTS technique is more precise and safe for changes to `.properties` files than a traditional Java RTS technique by Harrold et al. [52].

Most relevant to this thesis, Celik et al. [16] develop RTSLINUX a file-level RTS technique for software on Linux systems. RTSLINUX is conceptually akin to EKSTAZI by Gligoric et al. [46], but wraps the Linux kernel's system call interface instead of instrumenting the Java standard library to track accesses to non-code files. We have already alluded to why maintaintability and security concerns remain in practice, even if RTSLINUX was publicly available and the kernel modification could be applied to closed-source operating systems (see Section 8.1.1).

Summary Existing RTO techniques that are aware of changes to non-code artifacts target JVM-based software or are limited to Linux systems and ignore compile-time configuration, which may also affect test outcomes (⇒ **G3: Non-code Artifacts**).

8.1.3. Distributed Systems

In distributed software systems, adequate regression testing often involves end-to-end tests that operate across system or—in the context of web services—service boundaries. Consequently, RTO techniques relying on program analysis need to collect relevant information, even if a test involves a chain of requests invoking code on various services and physical machines. There are a few existing RTS techniques [101, 124, 178] for web applications and services that acknowledge these system boundaries, where we deem the following to be most relevant to this thesis.

Nakagawa et al. [124] develop a dynamic, method-level RTS technique for manual end-to-end tests in legacy Java web applications. If testers execute a manual test case in their browser, each HTTP request that is sent to the Java web server receives a custom

header. The additional header information is then associated with covered Java methods in the corresponding JVM thread of the server. From a case study on two industrial applications, the authors conclude that for large modifications most tests need to be executed, but for small changes only roughly 25% of the test cases were selected compared to the manual selection by developers. Moreover, the performance overhead from the code instrumentation was reported to be non-negligible and concerning to developers. The proposed RTS technique traces a single web server and is thus unsuitable for contexts with interaction of multiple web services.

Another RTS tool for web applications, WEBRTS, is proposed by Long et al. [101]. Similar to EKSTAZI [45], WEBRTS collects and stores per-test information at the class/file level using Java byte code instrumentation and can also instrument multiple instances of a Java web server. WEBRTS targets web applications that rely on server-side page rendering and use the HTTP protocol for communication. This restricts the applicability of WEBRTS, as modern (micro-)services often use more efficient remote procedure call mechanisms such as gRPC [178]. In case of changes to client code, such as JavaScript files, WEBRTS falls back to a retest-all strategy. The authors evaluate WEBRTS on up to eight versions of two small example projects and two industry projects and find that WEBRTS can in the best case reduce testing time by up to 75% compared to retest-all.

Zhong et al. [178] develop TESTSAGE, a dynamic, function-level RTS technique for testing large-scale C++ web services at Google. To collect the functions covered by each test across different services, TESTSAGE relies on LLVM `XRay` [100], a lightweight function tracing component. TESTSAGE is thereby limited to C++ web services and does not support parallel test tracing. The authors report that TESTSAGE reduces the required testing time by up to 50% compared to retest-all.

Summary There are a few existing RTS techniques targeting distributed systems. However, they (1) instrument server code, which is incomplete for end-to-end testing of microservices where rich web or mobile clients may contain significant business logic and orchestrate requests to multiple services, (2) are limited to specific communication protocols, or (3) lack analysis of the instrumentation overhead during service startup and test execution and the impact of the collection granularity for per-test dependencies. We are not aware of any prior work that targets program analysis based RTS, evaluates instrumentation overhead and per-test dependency granularity, and applies RTS in the context of end-to-end testing for microservice-based systems (⇒ **G4: End-to-End Testing Microservices**).

8.2. Process Characteristics Affecting Regression Testing

8.2.1. Continuous Integration Testing

In recent years, automated regression testing is largely done in CI environments, as CI infrastructure facilitates the provision of isolated (ephemeral) build and test environments and relieves developers of complex local test configuration or setup. This has led to extensive research efforts on RTO for CI environments (e.g., [9, 14, 17, 32, 62, 63, 72, 96, 107, 118, 135, 157, 158, 173]). Since CI pipeline execution time is driven by compilation, (static) analysis, and testing efforts [32, 37, 107, 157], it is essential for the design of adequate RTO

techniques to consider how regression testing is embedded into the CI process. Therefore, we consider the following studies most relevant to this work, as they incorporate surrounding factors in CI processes that may affect CI pipeline time.

Shi et al. [157] propose GIBSTAZI, an RTS technique built on top of EKSTAZI [45] and the module-level static incremental build tool `GIB` [44]. GIBSTAZI aims to not only reduce testing time, but also the required build time by selecting Maven modules and tests for compilation and execution in CI pipelines, respectively. In case of changes to non-code files, GIBSTAZI falls back to retest-all. When evaluated on 22 open-source projects, GIBSTAZI thereby achieves better safety than EKSTAZI (which does not consider non-code changes) and reduces end-to-end build and test time by 23% in a cloud-based CI environment. The authors further report that RTS saves time and resources and can also help developers avoid wasting time debugging flaky failures that are unrelated to their changes.

Jin and Servant [63] evaluate ten techniques from prior research that aim to improve CI testing by reducing the feedback time or computational cost for CI pipelines (including RTP [32, 109, 161] and RTS [46, 56, 107] techniques). The studied techniques therefore either (1) skip entire CI builds or tests from the test suite or (2) prioritize potentially failing builds or tests to execute them first. The authors conduct empirical analysis on 100 open-source projects from the TRAVISTORRENT dataset [8] and, among many other things, find that RTS techniques could skip full test suites or even builds, in order to save test and build preparation time. They also report that skipping builds with changes to non-code files is unsafe, as this way configuration or compilation problems can be missed which account for 35% of the failing builds. Furthermore, the authors argue that future CI optimization techniques should combine RTS with RTP and apply build selection techniques, to reduce the effort for uncritical builds. Overall, the results indicate that RTO techniques used in CI pipelines should be carefully integrated with build and test infrastructure for maximum cost-effectiveness.

Summary Through CI practices, regression testing is tightly coupled to building the software, as CI pipelines typically first compile and link the code before executing tests. Adequate RTO techniques should integrate with the build system as (1) changes to the build system configuration may impact the test results and simply running a full build and retest-all for non-code changes is imprecise, and (2) selectively compiling only the code relevant for the changes and affected tests can save valuable time inside CI pipelines. To our knowledge, no existing techniques address both issues (⇒ **G5: Build System Integration**).

8.2.2. Manual Testing

Manual regression testing is widespread and because of the costly human effort involved, there is a great need and potential for optimization. Existing research on RTO primarily considers automated unit-level testing [14]. These techniques are often difficult to transfer to manual regression testing as required data may not be available (e.g., code coverage or static call graphs) [53, 54] or techniques cannot be easily integrated into manual testing processes (e.g., manual testers might perform system-level black-box testing) [53]. Despite these difficulties, the following works have applied RTO techniques to manual regression testing [27, 53, 66, 80, 124].

Hemmati et al. [53] were the first to investigate RTP for manual black-box system testing on Mozilla Firefox in 2015. Their results indicate that using test failure history is an effective RTP surrogate in agile development environments compared to prioritizing tests by test diversity based on the textual test case descriptions.

Lachmann et al. [80] propose an ML-based RTP technique for manual system tests that incorporates test execution history (i.e., test failures and their priority as well as test execution cost), requirements coverage, and textual test case descriptions. Their RTP approach performs favorably compared to random ordering in a case study on two industrial projects. The results further indicate that RTP performance can be improved by extracting features from test case descriptions in addition to only using test history and test execution cost.

Juergens et al. [66] conduct an industrial case study to demonstrate challenges in applying RTS to manual system tests using method-level per-test traces obtained via dynamic analysis. They find that the common under-specification of manual tests can lead to unstable traces, which limits effectiveness for dynamic RTS techniques. Therefore, the authors suggest a semi-automated process, where testers inspect the results of RTS and then, based on domain knowledge, manually decide which tests to execute.

Eder et al. [27] propose an RTS technique that tries to recover trace links between source code and manual system tests written in natural language using static analysis. In their case study using four test cases on a single system, their technique is more effective than a random selection baseline strategy and correctly links 90% of the source code methods to tests. However, the calibration and evaluation of the RTS technique still requires dynamic program analysis, which limits its transferability [54].

Finally, we have already described the work of Nakagawa et al. [124] on RTS for manual end-to-end testing of legacy Web applications (see Section 8.1.3), which showed that large code changes lead to effectively selecting all tests and the performance penalty imposed by program instrumentation was concerning to developers.

Summary The few existing studies on manual regression testing indicate that, unlike automated testing, manual testing is often decoupled from VCS or CI systems and performed at the system level in a black-box manner without access to program analysis information. Yet, we are unaware of any work that (1) investigates which prerequisites and caveats practitioners have to expect when transferring RTO techniques to their manual testing process and (2) provides guidelines on how RTO techniques can be integrated into these manual testing processes ($\Rightarrow$ **G6: Transfer to Manual Testing**).

9. Conclusion

This chapter concludes the doctoral thesis, summarizes the results, discusses implications and limitations, and provides an outlook and ideas for future work. Parts of this chapter have appeared in peer-reviewed publications [33–37, 48, 58, 169], co-authored by the author of this thesis.

9.1. Summary

RTO aims to improve the cost-effectiveness of regression testing. Although significant research has proposed numerous techniques, modern software systems can only partially benefit from these advances: Industrial software systems are commonly written in multiple programming languages, are highly configurable and distributed, and require testing processes that involve both continuous automated and effort-intense manual testing. Most traditional RTO techniques are unsuitable if the outlined challenging system and process characteristics are present; these techniques are often limited to small, monolingual, and monolithic systems tested primarily through automated unit tests [14]. This thesis aims to bring state-of-the-art research closer to practice by integrating system and process characteristics that affect regression testing into RTO techniques. Therefore, we adapt existing and present novel RTO techniques to address challenges related to these characteristics by harnessing program analysis approaches or utilizing information that is readily available from testing processes. The main results of this thesis emerge from empirical studies conducted on open-source and industrial projects, where we show that these RTO techniques effectively reduce developer feedback time and testing efforts, while still reliably detecting bugs: For instance, we were able to effectively reduce the test duration on average by 42%–72% compared to *retest-all* for pull requests in CI environments at our industry partner IVU without failing to select any real test failures [35, 37]. Some of the techniques developed as part of this thesis are currently used on a daily basis at IVU. We have also publicly released tools and supplemental material to foster future research and improve practical tool support [34–36, 48, 58].

The practical implications from this thesis are the following: First, if RTO is implemented for systems that exhibit characteristics such as multi-language programming and high configurability, it is crucial to go beyond language-specific analysis and consider changes to non-code artifacts as well in order to maximize the fault detection capability. Depending on the information collected during the testing process, different techniques can be used. We find that techniques using (1) dynamic program analysis as well as (2) development metadata can be cost-effective in software with challenging system or process characteristics. Second, dynamic program analysis techniques that are agnostic to the programming language of the SUT can be effectively applied for RTS in CI processes of large-scale multilingual software. In two industrial RTS studies [35, 37], we have successfully used (1) system call instrumentation as an effective extension to language-specific analysis to account for changes to non-code artifacts or code in multi-

ple programming languages and (2) dynamic binary instrumentation which allows fine-grained run-time analysis of binaries compiled from binary-compiled languages (e.g., C or C++), even if these binaries are accessed through cross-language links. If the SUT is a distributed system, existing polyglot observability and distributed tracing frameworks provide a solid backbone for efficiently implementing dynamic RTO techniques that reliably capture the end-to-end behavior of multiple (micro-)services [33]. Third, if dynamic program analysis is infeasible or prohibitively expensive, one can still exploit the variety of metadata for effective RTO that are readily created during testing. We have demonstrated that (1) by only relying on readily available information from VCS and CI systems, we can significantly reduce test feedback time and effort [34] and (2) if the availability of data is limited, as it is often the case for manual testing processes, various simple RTO approaches can reduce the time to find failures [48]. It is worth noting that sometimes simple, well-understood heuristics (primarily using test failure history) seem to work equally well than more complex ML-based techniques for prioritizing tests—without the complex intricacies of deploying and maintaining an ML model.

9.2. Limitations

This thesis is subject to several limitations. We have already shortly discussed limitations and threats to validity for each included publication in the respective chapters, whereas more detailed discussions are part of the publications themselves. Therefore, in the following, we summarize the most important overarching limitations.

Scope and Generalization This thesis presents RTO techniques that incorporate several system and process characteristics which can affect regression testing. The implication is that there might be other relevant context-specific characteristics (see Section 9.3 for future work) and that our results cannot easily generalize beyond the studied (open-source and industrial) software projects. Nevertheless, many of our results are consistent with related RTO research and we have based the implementation of our solutions on established technology stacks to facilitate use and portability. The techniques developed and evaluated in this thesis are limited by some design decisions that were necessary to reduce the large space of implementation parameters. We took these decisions according to the findings from prior research or by discussing them with developers and testers from an industrial context. Particularly, this concerns the granularity of a test, which could be a test suite, a test case, or a test module; the applied type and granularity of program analysis (static vs. dynamic; file- vs. function- vs. module-level); or the compared ML approaches (supervised vs. reinforcement learning). Furthermore, the scope of this thesis is limited to functional regression testing, whereas there are other types of regression testing for non-functional quality aspects, such as performance [121] or security [41].

Industry-Relevance of Problems The research conducted as part of this thesis was carried out in publicly-funded and bilateral projects with two industrial companies, IVU and CQSE. Naturally, the insights gained from discussions with these industry partners have highly influenced the understanding of the problems addressed by this thesis, including the discussed challenges and their relevance. While these problems have also been reported to be industry-relevant by other researchers and practitioners [1], our work is

thus limited by the choice of studied problems, whereas other problems might be more relevant elsewhere due to the context-specific nature of software engineering.

Measuring Testing Cost Testing costs are manifold and may involve the required run time, expenses for dedicated testing hardware, setup efforts, or involved human expert effort (e.g., salary for senior testers or developers for manual testing). Similar to prior studies [114, 134, 158], we partially rely on the measured test execution time reported by testing frameworks in test reports to approximate costs for a test case. This way, we can conduct empirical studies across large time ranges (months, years) of CI testing activity [34, 35, 37]. These measured test durations in CI environments may, however, be affected by fluctuations due to other pipelines running in parallel on the same CI machines. For our case studies on manual testing [33, 48], we do not have accurate test duration measurements as they are often not tracked for manual tests [48, 169] and thus report the test selection ratio or assume equal test costs, which could lead to biased results [18, 46].

Cost of Information Collecting the required information for different RTO techniques involves costs, where typically collecting information from dynamic program analysis is the most expensive and readily available metadata is the cheapest. In this thesis, we have used probe-based system call instrumentation, which itself introduces relatively small overhead [16, 36], but has the limitation that each test should run in its own forked process. Nevertheless, the presented system call instrumentation can also be used as an extension to language-specific instrumentation (i.e., without process forking) for collecting more complete test traces including external accessed files. For our RTS technique targeting C++ binaries, BINARYRTS, we find that the applied dynamic binary instrumentation introduces substantial overhead—partially due to the framework used to implement BINARYRTS [35]. While the overhead does not directly impact developers' perceived feedback time in the studied context, as the test traces are generated in off-peak hours, less costly instrumentation approaches are still desirable and have been proposed for x86-64 ELF binaries [69]. Finally, in our study on readily available information from CI or VCS metadata, we have only measured the training cost of test ranking models without considering the feature computation costs [34]. Notably, recent work has found that most of the features we have used are relatively inexpensive to collect, whereas coverage-related features are the most expensive [170].

Fault Detection Capability RTO techniques share the goal to ideally retain fault detection (e.g., by safely selecting tests) and, in the case of RTP, further optimize for earlier fault detection. In practice, information about observed test failures (e.g., in a CI pipeline) is usually stored, whereas information about faults is often not available [134]. Similarly, the severity of faults (or failures) is commonly unknown [134]. For our work on RTP, we lack this information as well and thus, similar to prior research [113, 134, 155], we assume an one-to-one failure-to-fault mapping and equal fault severity [34, 48]. For the developed RTS techniques, we find that the fault detection capability (i.e., RTS safety) is mainly limited by outdated test traces and Dependency Injection (DI) mechanisms [35, 37]. However, we recently developed a tool for DI-aware RTS in Java projects [58].

Intermittent Test Failures There are various reasons for intermittent test failures, such as non-determinism in the tests, the code under test, the infrastructure, or the test environment [84]. Tests that non-deterministically pass and fail on the same version of the code are also referred to as flaky tests [7]. Prior research has shown that the prevalence of intermittent failures or flaky tests can impact the evaluation results for RTP and RTS techniques [40, 83, 89, 107, 134, 157]. Some of the empirical results reported as part of this thesis (mainly Chapter 5 [34]) may also be impacted by intermittent test failures. To minimize this threat to validity, we have invested significant manual effort to inspect the root causes for test failures in more recent studies [35, 37]. Notably, the automatic detection, root causing, and repair of flaky tests has become an active area of research itself in the past decade [7, 74, 79, 81, 82, 106, 133, 140, 154, 177].

9.3. Outlook and Future Work

Further System and Process Characteristics There are further system and process characteristics that affect regression testing beyond those addressed by this thesis. Recall our conceptual framework from Chapter 1 (see Figure 1.1) where we have listed examples for further characteristics, partly reported by practitioners in the study from Ali et al. [1]. These include, for instance, embedded systems, where unreliable Hardware-in-the-Loop test environments can sometimes substantially impact test results [64]. Furthermore, also if testing processes imply less frequent testing than with CI processes, RTO may provide significant benefits as, for example, early fault detection is also crucial if regression testing is only performed before releases (e.g., yearly or quarterly) [169]. Similarly, there are still open challenges concerning the system and process characteristics which this thesis addresses, such as recent approaches for configuration testing in configurable systems [20] or highly distributed serverless cloud software without direct control of the execution environment.

Manual Regression Testing We have shown that techniques and case studies on manual regression testing are underrepresented in academic and industrial research, despite the high involved testing effort and its persistent spread in practice [48]. As the few existing techniques are not unequivocal in their results on what works well and what does not, further research is needed to investigate RTO for manual testing and its trade-offs in practice. That includes, for instance, studying IR-based RTP, which shows promising results in automated testing [134], as well as developing adequately efficient dynamic program analysis techniques since their overhead has been concerning to developers in manual regression testing [124].

Effect of Outdated Test Traces When deploying some of the developed RTS techniques in practice, we collect per-test execution traces *off-line*, i.e., in off-peak hours in decoupled CI pipelines [35, 37]. Depending on the frequency of these tracing pipelines, RTS can become unsafe due to outdated test traces (see discussed limitations above in Section 9.2). This raises the questions of how quickly the (costly to obtain) test traces become outdated and at what point this can lead to safety violations. Future research should therefore study how to systematically determine a cost-effective strategy (i.e., frequency) for recollecting test traces, as running instrumented tests for each commit is usually infeasible

in large-scale CI environments [32, 35, 37, 89, 107]. Potentially, the durability of per-test execution traces could be prolonged by statically analyzing (and approximating) the impact of changes introduced since the traces were collected.

Further Programming Languages In this thesis, we focus on software systems that—while making use of non-code artifacts or cross-language links to other programming languages—predominantly use Java and C++ as their main GPL. This is in line with the majority of RTO research which also targets C, C++, or Java systems. However, with the rise of serverless cloud software and ML-based systems, interpreted languages such as Java-/TypeScript and Python are commonly used as the primary GPL[10]. Hence, future research should investigate how RTO techniques can be successfully applied in these language environments as they might challenge some dynamic program analysis approaches such as system call analysis [16].

New Approaches for Program Analysis We have investigated different dynamic program analysis approaches for RTO in this thesis and have pinpointed potential for novel, non-intrusive ways of analyzing systems at run-time using probe-based dynamic instrumentation in another work [36]. Several research works have also demonstrated the effectiveness of IR-based RTP techniques [134], which extract test code using static analysis and then rank tests by their similarity to introduced changes. While these relatively inexpensive approaches for static program analysis are promising, experiences on large-scale industrial deployment of these techniques is yet missing. Since the wide applicability of program analysis techniques also enables wider support of RTO techniques, future research should explore novel ways for dynamic and static program analysis that are applicable to different languages and to large code bases with frequent changes.

[10] According to the Stack Overflow Developer Survey 2022, JavaScript has been the most commonly used language for the past ten years, while Python was the second most used GPL [159].

A. Appendix

This chapter lists and includes the peer-reviewed publications included in this publication-based doctoral thesis.

A.1. Overview

The contributions made by this publication-based dissertation have previously appeared in the following peer-reviewed publications:

P1 **Daniel Elsner**, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. *Build System Aware Multi-language Regression Test Selection in Continuous Integration.* Proceedings of the International Conference on Software Engineering: Software Engineering in Practice, pages 87–96, 2022 ***(core publication 100%)***

P2 **Daniel Elsner**, Severin Kacianka, Stephan Lipp, Alexander Pretschner, Axel Habermann, Maria Graber, and Silke Reimer. *BinaryRTS: Cross-language Regression Test Selection for C++ Binaries in CI.* Proceedings of the International Conference on Software Testing, Verification and Validation, pages 327–338, 2023 ***(core publication 100%)***

P3 **Daniel Elsner**, Florian Hauer, Alexander Pretschner, and Silke Reimer. *Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration.* Proceedings of the International Symposium on Software Testing and Analysis, pages 491–504, 2021 ***(core publication 100%)***

P4 **Daniel Elsner**, Daniel Bertagnolli, Alexander Pretschner, and Rudi Klaus. *Challenges in Regression Test Selection for End-to-End Testing of Microservice-based Software Systems.* Proceedings of the International Conference on Automation of Software Test, pages 1–5, 2022

P5 Roman Haas*, **Daniel Elsner***, Elmar Juergens, Alexander Pretschner, and Sven Apel. *How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies.* Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1281–1291, 2021 ***(core publication 50%)***

According to the regulations for publication-based doctoral theses of the Technical University of Munich, the publications P1 to P3 are core publications (100%) and P5 is a 50% core publication (because of the shared first authorship). Publication P4 is no core publication because it is a short paper (5 pages).

*Both authors contributed equally

A.2. Copyright Policies by Publishers

The publications have been published in conference proceedings by IEEE and ACM. In the following, we include the copyright policies by the publishers which allow us to include the full articles in Section A.3.

Build System Aware Multi-language Regression Test Selection in Continuous Integration

Conference Proceedings:
2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)

Author: Daniel Elsner

Publisher: IEEE

Date: May 2022

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

BACK CLOSE WINDOW

BinaryRTS: Cross-language Regression Test Selection for C Binaries in CI

Conference Proceedings:
2023 IEEE Conference on Software Testing, Verification and Validation (ICST)

Author: Daniel Elsner

Publisher: IEEE

Date: April 2023

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

ACM Author Gateway

Author Resources

ACM Author Rights

ACM exists to support the needs of the computing community. For over sixty years ACM has developed publications and publication policies to maximize the visibility, impact, and reach of the research it publishes to a global community of researchers, educators, students, and practitioners. ACM has achieved its high impact, high quality, widely-read portfolio of publications with:

- Affordably priced publications
- Liberal Author rights policies
- Wide-spread, perpetual access to ACM publications via a leading-edge technology platform
- Sustainability of the good work of ACM that benefits the profession

Choose

ACM gives authors the opportunity to choose between two levels of rights management for their work. Note that both options obligate ACM to defend the work against improper use by third parties:

- **Exclusive Licensing Agreement:** Authors choosing this option will retain copyright of their work while providing ACM with exclusive publishing rights.
- **Non-exclusive Permission Release:** Authors who wish to retain all rights to their work must choose ACM's author-pays option, which allows for perpetual open access to their work through ACM's digital library. Choosing this option enables authors to display a Creative Commons License on their works.

Post

Otherwise known as "Self-Archiving" or "Posting Rights", all ACM published authors of magazine articles, journal articles, and conference papers retain the right to post the pre-submitted (also known as "pre-prints"), submitted, accepted, and peer-reviewed versions of their work in any and all of the following sites:

- Author's Homepage
- Author's Institutional Repository
- Any Repository legally mandated by the agency or funder funding the research on which the work is based
- Any Non-Commercial Repository or Aggregation that does not duplicate ACM tables of contents. Non-Commercial Repositories are defined as Repositories owned by non-profit organizations that do not charge a fee to access deposited articles and that do not sell advertising or otherwise profit from serving scholarly articles.

For the avoidance of doubt, an example of a site ACM authors may post all versions of their work to, with the exception of the final published "Version of Record", is ArXiv. ACM does request authors, who post to ArXiv or other permitted sites, to also post the published version's Digital Object Identifier (DOI) alongside the pre-published version on these sites, so that easy access may be facilitated to the published "Version of Record" upon publication in the ACM Digital Library.

Examples of sites ACM authors may not post their work to are ResearchGate, Academia.edu, Mendeley, or Sci-Hub, as these sites are all either commercial or in some instances utilize predatory practices that violate copyright, which negatively impacts both ACM and ACM authors.

Distribute

Authors can post an Author-Izer link enabling free downloads of the Definitive Version of the work permanently maintained in the ACM Digital Library.

- On the Author's own Home Page or
- In the Author's Institutional Repository.

Reuse

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is notthe editor, requires permission and usually a republication fee.
- Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

Create

ACM's copyright and publishing license include the right to make Derivative Works or new versions. For example, translations are "Derivative Works." By copyright or license, ACM may have its publications translated. However, ACM

Authors continue to hold perpetual rights to revise their own works without seeking permission from ACM.

Minor Revisions and Updates to works already published in the ACM Digital Library are welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

- If the revision is minor, i.e., less than 25% of new substantive material, then the work should still have ACM's publishing notice, DOI pointer to the Definitive Version, and be labeled a "Minor Revision of"
- If the revision is major, i.e., 25% or more of new substantive material, then ACM considers this a new work in which the author retains full copyright ownership (despite ACM's copyright or license in the original published article) and the author need only cite the work from which this new one is derived.

Retain

Authors retain all perpetual rights laid out in the ACM Author Rights and Publishing Policy, including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images intended for exploitation in other contexts
- All patent and moral rights
- Ownership and control of third-party permissions to use of software published by ACM

A.3. Publications

A.3.1. Build System Aware Multi-language Regression Test Selection in Continuous Integration

Build System Aware Multi-language Regression Test Selection in Continuous Integration

Daniel Elsner
Roland Wuersching
Markus Schnappinger
Alexander Pretschner
firstname.lastname@tum.de
Technical University of Munich
Munich, Germany

Maria Graber
René Dammer
Silke Reimer
{grm,rda,sre}@ivu.de
IVU Traffic Technologies
Berlin, Germany

ABSTRACT

At IVU Traffic Technologies, continuous integration (CI) pipelines build, analyze, and test the code for inadvertent effects before pull requests are merged. However, compiling the entire code base and executing all regression tests for each pull request is infeasible due to prohibitively long feedback times. Regression test selection (RTS) aims to reduce the testing effort. Yet, existing *safe* RTS techniques are not suitable, as they largely rely on language-specific program analysis. The IVU code base consists of more than 13 million lines of code in Java or C/C++ and contains thousands of non-code artifacts. Regression tests commonly operate across languages, using cross-language links, or read from non-code artifacts. In this paper, we describe our build system aware multi-language RTS approach, which selectively compiles and executes affected code modules and regression tests, respectively, for a pull request. We evaluate our RTS technique on 397 pull requests, covering roughly 2,700 commits. The results show that we are able to safely exclude up to 75% of tests on average (no undetected real failures slip into the target branches) and thereby save 72% of testing time, whereas end-to-end CI pipeline time is reduced by up to 63% on average.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Software testing, regression test selection, continuous integration

ACM Reference Format:
Daniel Elsner, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. 2022. Build System Aware Multi-language Regression Test Selection in Continuous Integration . In *44nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3510457.3513078

ICSE-SEIP '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9226-6/22/05…$15.00
https://doi.org/10.1145/3510457.3513078

1 INTRODUCTION

Regression testing is regularly performed on software systems to ensure changes did not inadvertently affect existing system behavior [23]. The simplest, yet expensive strategy to perform regression testing, *retest-all*, is to execute every test case after each change. However, with increasingly large test suites and limited physical resources this approach is often too costly, especially in continuous integration (CI) testing [11, 13, 24, 37]. To reduce the costs of regression testing, regression test selection (RTS) [15, 22, 29, 31, 32, 34, 38] has been extensively studied since the 1970s [14].

RTS techniques are considered *safe*, if they select *all* tests that are affected by the changes to the code base, such as the changeset of a pull request. Therefore, they collect dependencies for each test. This is implemented either through static (e.g., class dependency graph) or dynamic (e.g., code instrumentation) program analysis [15, 16, 21, 22, 34, 38, 39]. These per-test dependencies are then used to determine the relevant test cases. However, the involved language-specific program analysis can be expensive, which is why existing safe RTS techniques often bear (prohibitively) extensive costs [11, 24, 30]. Lightweight, less intrusive yet *unsafe* RTS techniques use CI or version control system (VCS) metadata to select tests, but the underlying statistical models can only provide project-specific empirical safety trade-offs [7, 8, 11, 12, 20, 24, 30, 35].

IVU Traffic Technologies is one of the world's leading providers of public transport software solutions. The software system considered in this study accounts for approx. 13.5*M* Java and C/C++ lines of code (LOC). A large variety of domain specific language (DSL) source files and non-code artifacts including build and other configuration files, expected test results, and resources complement the code base. IVU maintains the ten most recent releases of the system on dedicated release branches. Before pull requests are merged into any of these release branches, they are thoroughly tested for regressions. However, compiling the entire code base and running all of the thousands of regression tests for each pull request can take up to several hours despite a high degree of parallelization within the CI pipelines. This results in intolerable feedback time and is economically infeasible: When the number of queued pull requests increases, developers often need to wait until the next day for test feedback. Consequently, reducing the overall testing efforts in pull requests through selective compilation and testing is required.

Probabilistic RTS techniques have already been successfully employed at IVU in CI pipelines for the main development branch [12]. However, applying these unsafe techniques to pull requests on

release branches bears fundamental risks: Support patches are directly built from these release branches every day, hence, code from those branches is potentially deployed into customer's infrastructure. Therefore, when testing pull requests for release branches, we aim for test selection that is as safe as reasonably achievable.

Yet, existing safe RTS techniques are not applicable in the given industrial context: The complexity of the software requires test scenarios at integration and system level, where tests commonly operate across languages and intensively use non-code resources. To the best of our knowledge, there is no RTS technique that addresses this problem for Windows environments, as opposed to Linux-only approaches [10]. Furthermore, prior RTS approaches select a test case if the checksum of any per-test dependency has changed. There are two shortcomings to this approach: First, these RTS techniques might miss tests in case of changes made to the build system. For instance, adding a new runtime dependency in a configuration file might completely change a test's behavior, even though checksums of previously recorded per-test code dependencies are unchanged. Second, these techniques assume that a fully compiled workspace is readily available. At IVU, compiling the Java code base already takes roughly half an hour on average. Building only the relevant modules that are affected by changes or contain selected tests can have a significant impact on end-to-end CI pipeline execution time—especially for small changesets.

In this paper, we propose a build system aware RTS technique which harnesses dynamic and static program analysis to collect file-level per-test dependencies across language boundaries: We combine language-agnostic system call tracing, Java class loader instrumentation, as well as static code and build dependency analysis. This yields more complete per-test dependencies than pure language-specific approaches, thus leading to safer module and test selection for compilation and test execution, respectively.

We evaluate our RTS technique on 397 pull requests and measure the time savings in the CI pipelines across five weeks, covering roughly 2,700 commits. In addition to traditional measures, such as the ratio of selected tests, we also consider the observed real-life end-to-end time saving, i.e., compilation plus test execution time. The results on two evaluation branches show our approach can select tests safely (no undetected failures slip into target release branches) and thereby saves on average 42% and 72% of test execution time, depending on how recent the release branch is. End-to-end CI pipeline time is further reduced by up to 63% on average, when compared to pull requests with full build and retest-all strategy. Although we evaluate our RTS technique in just one industrial context, we expect it to be applicable to other multi-language software systems. Due to the resulting shorter feedback cycles, our RTS approach is now deployed to all release branches at IVU.

2 CONTINUOUS INTEGRATION TESTING AT IVU TRAFFIC TECHNOLOGIES

This paper describes the optimization of the CI testing process for pull requests at IVU using a novel build system aware multi-language RTS approach. To better understand the context of this study, we first explain the system under test and the testing process for pull requests at IVU. Second, we elaborate on established state-of-practice RTS techniques and their drawbacks in the given context.

2.1 System Description

At IVU, source code of the main software products is stored in a monolithic code repository. The code repository is split into two subtrees, one containing mainly C/C++ source code (approx. $9.5M$ LOC) and one with mainly Java source code (approx. $4M$ LOC). Additionally, both subtrees contain, amongst others, hundreds of thousands LOC written in Java-/TypeScript, C#, Perl, Python, SQL, and Assembly, as well as millions of lines in non-code artifacts such as XML, CSV, YAML, Java Properties, and plain text files.

The code repository is structured through >4,000 Maven[1] modules, yet Maven is only used as the build system for the Java subtree. For the C/C++ subtree, a self-maintained build tool is employed, which wraps Microsoft's Visual C++ compiler, as most software products primarily target Microsoft Windows.

The regression test suite is composed of unit, integration, and system level test cases. Tests written in Java are naturally located in the Java subtree, whereas C/C++ tests reside in the C/C++ subtree. In this study, we focus on the Java tests. These are further separated into so-called *short-running* and *long-running* test cases. Both types of test cases frequently interact with databases, which makes them inherently costly to run. Opposed to short-running tests, long-running tests operate on real databases instead of in-memory databases and use the Java Native Interface (JNI) to interact with dynamic-link libraries (DLLs) built from the C/C++ subtree. Executing the entire suite (~10,000 test cases) yields unbearable feedback times of around 2 hours (excluding compilation and code analysis; see next section). Motivated by this high potential for improvement, this paper describes how we reduce the effort for building and testing Java code in pull request CI pipelines.

2.2 Pull Request CI Testing

To continuously integrate code changes, IVU uses a Jenkins[2] CI system. Respective CI pipelines (1) build, (2) analyze, and (3) test the code base. These pipelines are continuously running for the main development branch and release branches, which contain currently supported and already released versions of the software.

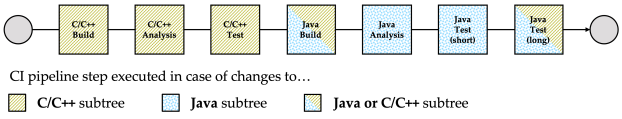


Figure 1: Pull request CI pipeline as executed after every change to a pull request branch

Before developers integrate changes into one of these branches, they have to create a *pull request*. These pull requests contain changesets which typically implement one feature, bug fix, or other enhancement. When a pull request is opened, a new CI pipeline is created for it. This pipeline will rebase the pull request branch (*source*) onto the *target* branch (e.g., the release branch) and execute

[1] Maven: https://maven.apache.org
[2] Jenkins: https://www.jenkins.io/

the above mentioned steps. Every new commit to the source branch triggers a new run of the pipeline. Since there exists only a limited amount of build machines for these pull request CI pipelines, a pipeline run is typically queued before being executed. Once a machine is available, the CI pipeline for the pull request runs as depicted in Fig. 1. First, in case there are any changes related to the C/C++ subtree, the C/C++ code is built, analyzed, and tested. Second, the Java code is built, analyzed, and tested. Notably, exclusively changing the C/C++ subtree nevertheless triggers the Java build and the Java long-running tests, as these tests operate across language boundaries and potentially use C/C++ DLLs[3]. If no changes to the C/C++ code were made, the necessary DLLs are loaded from a central artifact repository.

Before the improvements we describe in this study, the Java-related pipeline steps took around 170 minutes on highly parallelized build machines (64–96 cores): Building took on average 28 minutes, static code analysis 24 minutes, short-running tests 38 minutes, and long-running tests 80 minutes.

2.3 Alternative Test Selection Approaches and State-of-Practice

The state-of-practice knows several safe RTS techniques and associated tools, specifically for Java projects [2, 4, 10, 15, 16, 19, 21, 22, 27, 29, 33, 34, 38, 39]. Existing techniques statically [2, 21, 22, 33] or dynamically [4, 10, 15, 16, 27, 34, 38, 39] collect per-test dependencies at the level of basic-blocks [19, 29], methods [4, 39], classes/files [15, 16, 21, 22], modules [2, 33], or combinations thereof [34, 38] to select tests. These techniques have been shown to effectively reduce the testing effort in various studies—especially on open-source software projects, yet particularly not at the scale of the industrial code base of IVU. However, existing techniques and tools suffer the following limitations that are crucial in the context of IVU.

2.3.1 Build System Awareness. We identify two requirements related to the build system.

First, the majority of more recent RTS techniques use file or method checksums of per-test dependencies to identify those dependencies that have changed since the last test execution [10, 15, 16, 21, 22, 38]. While this approach is easily transferable and does not require integration with the VCS, it is based on the assumption that a fully built workspace is readily available. At IVU, compiling the code of all >2,000 Java Maven modules already requires a significant time effort, on average roughly half an hour despite high parallelization. However, selecting only relevant modules for compilation provides a great time-saving potential. Thus, an adequate RTS technique in this context needs to select tests without having compiled sources available.

Second, existing RTS techniques are unsafe considering the build system configuration. This is because they either use language-specific analysis or collect per-test dependencies during test execution. Yet, build configuration files such as Maven `pom.xml` files, are not part of per-test dependencies since these files define the compile and runtime dependencies *before* the test is executed. To mitigate this threat, Shi et al. [34] recommend to use hybrid techniques, which combine static module and dynamic file-level analysis. However, their proposed technique, `GIBstazi`, selects *all* tests, if *any* changes to non-code files occur. Due to the large amount of non-code artifacts in our system, this does not provide significant advantages over retest-all. At IVU, we require a more precise approach that actively checks for build system changes and prunes the set of selected tests and modules to be compiled to a minimum.

2.3.2 Multi-language Test Traces. Since most existing RTS techniques for Java software rely on language-specific static analysis or code instrumentation, their collected per-test dependencies are impracticable for software that operates across language boundaries [10, 40]. At IVU, we intensively use cross-language links to implement Java tests that cover code from the Java and C/C++ code base, and code written in other programming languages. Moreover, our code base includes millions of lines in non-code artifacts, which are used for configuration purposes at run-time or serve as test resources (e.g., expected test results). Consequently, a safe RTS technique in our context needs to rely on test traces that cover multiple languages and non-code artifacts, e.g., by tracing calls issued to native DLLs during test execution.

To address this problem, Celik et al. [10] propose `RTSLinux`, an RTS technique that modifies the Linux kernel to intercept operating system calls related to file accesses and process management. This way, `RTSLinux` collects all file dependencies of a Java virtual machine (JVM) process executing a test and is thereby also aware of calls made from the JVM to native libraries via the JNI. However, even if `RTSLinux` was publicly available, it would not be applicable in our case: First, perhaps trivially, our software targets Windows environments and therefore—at least the C/C++ parts—cannot be executed on Linux. Since Windows is a closed-source operating system, directly modifying the kernel and system call application programming interfaces (APIs) is not easily possible. Second, assuming said kernel modification was technically feasible, IVU would most probably decide against using modified operating systems on their CI machines, as this would imply maintaining that extension for future versions of the kernel with utmost care to avoid kernel panics. Third and yet more importantly, `RTSLinux` also relies on file checksums to compute the set of changed files and thus requires a compiled workspace. Hence, `RTSLinux` does neither provide an applicable approach for multi-language software on Windows, nor does it address the need to reduce compile time.

2.3.3 Tool Support. Beyond the conceptual limitations of existing RTS techniques, we did not find a single RTS tool for JVM projects that was publicly available (released on Maven Central) and worked out-of-the-box in the given context. The main reasons are that existing tools do not support JUnit 5 (`Ekstazi` [15], `HyRTS` [38]), or fail with Java Development Kit (JDK) versions newer than 9 (we use 11) or specific language features such as Java type annotations[4] (`Clover` [4], `STARTS` [22]).

In summary, we need an RTS solution that is (1) build system aware, i.e., it is safe concerning changes to the build system configuration and can selectively build only those modules relevant for testing

[3] Although C/C++-only changesets are supposed to trigger the Java build and Java long-running tests since they might break tests, this has been partially deactivated throughout the course of our study, due to the significant overhead caused for small C/C++ changesets.

[4] Clover GitHub Issue on Java Type Annotations: https://github.com/openclover/clover/issues/20

introduced changes; (2) capable of collecting per-test dependencies to non-code artifacts and across programming languages.

Similar to prior RTS research [12, 15, 16, 21, 22], we target class- rather than method-level test selection. Therefore, unless otherwise stated, we refer to a Java test class (i.e., JUnit test suite) in the rest of the paper when we talk about a *test*.

3 BUILD SYSTEM AWARE MULTI-LANGUAGE TEST SELECTION

Existing RTS techniques are either unsafe or unapplicable at IVU. This motivates a novel approach, which is build system aware and capable of multi-language RTS. In this section, we first explain how to collect multi-language test traces as input for the test selection. Second, we show how to integrate the test selection with the build system to selectively compile modules for safe and cost-efficient testing. Last, we elaborate on integration details of our RTS technique into the pull request CI at IVU.

3.1 Collecting Multi-language Test Traces

To address the requirement of multi-language support for our RTS technique, we need to collect per-test dependencies to non-code artifacts and across language boundaries. Therefore, we harness and combine practical approaches for system call tracing and JVM class loader monitoring. By integrating these approaches with the JUnit testing framework [5] and the Maven Surefire Plugin [3], we can collect the required test traces at file-level granularity.

3.1.1 Probe-based System Call Tracing. System calls represent the interface to the operating system kernel that is visible to application programmers [36]. During a test's execution, its low-level behavior can be analyzed by tracing the invoked system calls. Thereby, we can collect the set of all accessed files, even if they are accessed by loaded DLLs or transitive child processes. The most straightforward way to trace all system calls issued by a test is to execute each test in isolation in its own forked JVM process which is supported by standard test execution frameworks [3, 10]. This further increases the reliability of test results as it prevents shared test state pollution or test-order dependencies [6, 28]. We can thus obtain stable file-level test traces by tracing all process- and file-related system calls for each JVM, i.e., for each test.

A similar approach is employed by `RTSLinux` [10]. However, none of the techniques for tracing system calls evaluated by Celik et al. [10] is available for the Windows operating system. As motivated in Sec. 2.3.2, even if `RTSLinux` was available for Windows, we would prefer a less intrusive, more maintainable approach. Yet, from the results reported by Celik et al., we learn that an efficient system call tracing approach has to operate in *kernel mode*, since tracing *all* system calls and filtering in *user mode* is prohibitively expensive (approx. four times the overhead of kernel mode tracing) [10].

In order to implement practical and efficient system call tracing, we use `DTrace` [9][5]. `DTrace` provides capabilities to dynamically instrument so-called *probes*. These are static or dynamic instrumentation points for which one can specify instructions to be executed if the probe fires (e.g., when entering a system call). Because this selective probe-based instrumentation is highly efficient and guaranteed to run safely inside the kernel [18], it has been deployed in production environments at Netflix, amongst others [17].

Our `DTrace` script takes a process identifier (PID) as input and instruments relevant system calls related to accessing files or spawning new processes. This way, we capture complete traces and finally store all relevant information (e.g., timestamps, PIDs, accessed filepaths) in a *tracing log*.

3.1.2 JVM Class Loader Monitoring. There are two drawbacks to relying *solely* on tracing system calls for Java tests:

First, in case a test loads a Java `.class` file that is located inside a Java archive (JAR), that JAR file will be part of the test trace, but not the actually used `.class` file. This could lead to imprecision in the test selection, as it stipulates to select that test if any of the files inside that JAR has changed [15]. Since many of our tests use classes from other Maven modules which are typically packaged as JARs, addressing this potential imprecision is crucial. Therefore, we attach a Java agent[6] to each JVM executing a test. The agent monitors whenever a new class is loaded by the class loader. If the corresponding `.class` file is located inside a JAR, it is added to the tracing log. If it is not located inside a JAR, we can safely ignore it, as we already cover it with our system call instrumentation. Note that we are only interested if a `.class` file was *ever* loaded during the execution of a test. Hence, as every test is running in its own forked JVM, we do not need to instrument the loaded file itself.

Second, similarly, in case a resource, such as an XML file, is loaded that is located inside a JAR, no separate system call to open that resource is invoked by the JVM. Instead, it is read from the already loaded JAR. Therefore, we instrument the `getResource(String)` method in the `java.lang.ClassLoader` class, as it is used for loading resources from JARs.

3.1.3 Integration with Testing Infrastructure. At IVU, we rely on JUnit 5 as our testing framework for Java. To execute each test in isolation in its own JVM process, we use Maven Surefire's forking mechanism[7]. This creates a new JVM per JUnit test class and we parallelize testing across Maven modules on all available CPU cores. To link the individual tests to the accessed files and spawned processes from our tracing log, we further need to obtain information about when a test started and terminated. Therefore, we register a custom JUnit test execution listener and subscribe to a test's start and end event [5]. The listener creates a *testing log* which contains start and end timestamps, the identifier of the test, as well as the JVM PID. We cannot only use the PID of the test to find its file accesses, as Windows may reuse PIDs after a process has been terminated.

Eventually, the tracing and testing logs are combined and we are able to compute a file-level test trace for each test[8] after all tests have been executed. We store the test traces in a CSV file, where each row contains a test name and a filepath accessed by that test.

[5] `DTrace` stands for Oracle Solaris Dynamic Tracing Facility

[6] Java Agent API for run-time code instrumentation on the JVM [1]

[7] `mvn surefire:test -DforkCount=1 -DreuseForks=false -T1.0C`

[8] Recall that we refer to a Java test class (i.e., JUnit test suite) when we talk about a *test*

3.2 Build System Aware Test and Module Selection

Our proposed RTS technique acknowledges changes to non-code artifacts and to the build system, e.g., changed dependencies. Algorithm 1 contains the pseudo-code of our build system aware test and module selection. Our algorithm has three inputs: (1) the changeset from the VCS, (2) the multi-language file-level test traces described in Sec. 3.1, and (3) *DLL-to-source* mappings for target and pull request branch. The latter are retrieved by analyzing the C/C++ compiler output which contains static compile dependencies for each DLL. More specifically, we parse `.tlog` files that are emitted by the Microsoft C++ compiler toolchain[9] and extract which source files each DLL depends on. The resulting mapping is stored in a CSV file, where one row has two columns; one contains the DLL file name and one a source filepath this DLL depends on. We need the information contained in this mapping since our test traces only include accesses to DLL files, not to actual C/C++ source files.

Changeset Analysis. Our algorithm analyzes the changeset by iterating over all changed files. For each Java source file, we search the Java source file for all `class`, `enum`, and `interface` definitions. From those definitions we create corresponding `.class` filepaths that match `.class` files generated by the Java compiler. For instance, if a Java source file that is part of the package `a.b.c` contains two classes `X` and `Y`, the two generated class filepaths are `a/b/c/X` and `a/b/c/Y`. These will also match accessed `.class` filepaths of nested or anonymous classes, e.g., if `X` contained one anonymous class, the Java compiler would output a file `a/b/c/X$1.class`, which can also be matched by `a/b/c/X`. Then, we check for presence of JUnit test method annotations. If the file does indeed contain test methods, it is considered a test suite and thus added to the set of tests to be executed. This way we safely select all newly created or updated test classes. For each C/C++ source file, we retrieve all affected DLLs from the DLL-to-source mappings and add the DLL paths to the set of affected filepaths. For changed Maven `pom.xml` files or files that can affect the build results (e.g., `.wsdl` or `.xsd` files), we select all tests from the changed module and all downstream modules, by retrieving them from the Maven reactor. The filepaths of all other changed files (e.g., `.xml` or `.csv` files) are also added to the set of affected filepaths.

While our approach intuitively works for additive and modifying changes, it is not immediately clear, how deletions have to be treated for each file type: We address deletions of and inside of `.java` files by parsing both, the old and the new revision (if existing) of the file. This way, we can also find all affected tests that covered any `.class` filepaths of the old revision. Deletions related to C/C++ are already covered, since we use the DLL-to-source mappings from both, the target branch and the pull request branch. Hence, if the pull request contains a C/C++ deletion, the deleted filepath will still be part of the DLL-to-source mapping of the target branch. For deletions of other file types, we simply search for the old filepath inside the test traces.

Test Selection. We iterate over all *test traces* and select those tests that have accessed files that match any of the affected filepaths.

[9] `.tlog` files in Microsoft Visual C++ [26]

Algorithm 1: Build System Aware Test and Module Selection

Input: changeset, test traces, DLL-to-source mappings
Output: selected tests, selected modules

```
1  selectedTests ← {}
2  selectedModules ← {}
3  affectedFilePaths ← {}
4  foreach change in changeset do
5      if isJavaFile(change) then
           /* get .class filepaths from .java file (before/after) */
6          affectedFilePaths +← getClassFilePaths(change)
           /* if @Test annotation is present, select test suite   */
7          if containsTests(change) then
8              selectedTests +← getTestSuiteIdentifier(change)
9      else if isCppFile(change) then
           /* look up affected DLLs in DLL-to-source mappings     */
10         affectedFilePaths +← getDLLFilePaths(change)
11     else if isRelevantForBuild(change) then
           /* select all tests of changed Maven module and from all
              transitive downstream modules                       */
12         selectedTests +← getTestsForModule(change)
13     else
14         affectedFilePaths +← change
   /* select modules for changes in pom.xml, .java, .xsd, and
      .wsdl files                                                 */
15     if isRelevantForBuild(change) then
           /* get enclosing Maven module for compilation          */
16         parentModule ← getParentMavenModule(change)
17         selectedModules +← parentModule
           /* include upstream modules (transitive)               */
18         selectedModules +←
             getUpstreamModules(parentModule)
           /* include direct downstream modules with their
              transtive upstream modules                          */
19         selectedModules +←
             getDownstreamModules(parentModule)
   /* compute affected tests from test traces                     */
20 selectedTests +← getAffectedTests(affectedFilePaths)
   /* search affected parent Maven modules for selected tests     */
21 testModules ← getAffectedModules(selectedTests)
22 selectedModules +← getUpstreamModules(testModules)
23 return selectedTests, selectedModules
```

Module Selection. We select all Maven modules for compilation that either (1) are directly impacted by the changeset or (2) enclose any of the selected tests. For (1) we further need to add all upstream modules to the set of selected modules, as they are required to compile the changed modules. Additionally, since changes in modules from (1) could break direct downstream modules, these also need to be selected including their transitive upstream modules, in order to be buildable. For (2) we need to add upstream modules as well, since they are required to build the tests' modules.

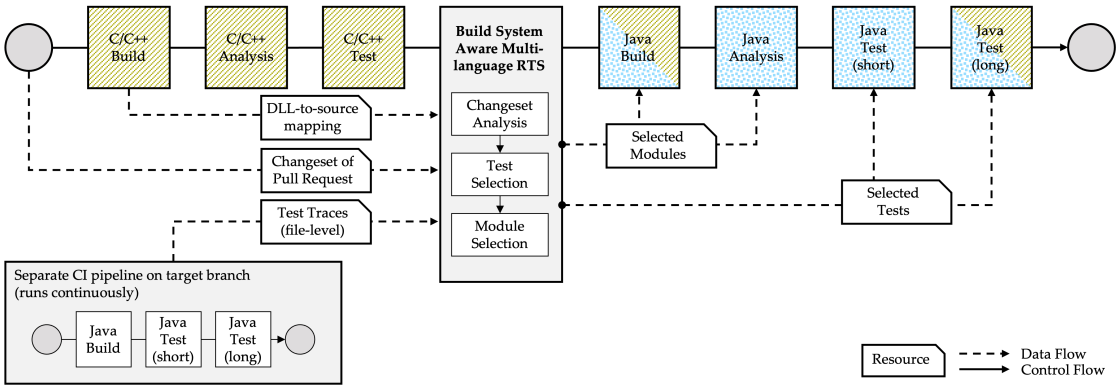


Figure 2: Pull request CI pipeline extended by our RTS technique

In the next section, we describe how we use these two results—the sets of selected tests and modules—inside the pull request CI to instruct Maven and Maven Surefire to selectively compile and execute modules and tests, respectively.

3.3 Integration into Pull Request CI

We integrated our RTS technique into the pull request CI, as illustrated in Fig. 2. Therefore, we added an additional step to the pipeline that calls our RTS algorithm, which we implemented as a simple command line interface (CLI) tool.

To provide the test traces to our tool, we created a separate *tracing* CI pipeline, that continuously runs for the target branches chosen in this study (see Sec. 4.1 for our evaluation setup). Currently, we update the test traces approx. once per day for each release branch, which provides good trade-offs regarding effort and effectiveness in our context (see our results in Sec. 4.2). We only update test traces for passing tests, since failing tests might yield incomplete traces (e.g., if the test terminated early).

Our tool receives the following inputs: First, the changeset of that pull request; second, the most recent test traces collected with our separate tracing CI pipeline; third, the DLL-to-source mapping from the most recent C/C++ build of the target branch. If changes were made to the C/C++ subtree within the pull request, we also provide the DLL-to-source mapping from the C/C++ build step. With these inputs, our tool computes the set of selected tests and modules and stores them in text files for subsequent pipeline steps.

To build and analyze only the selected Java modules, we extend the Maven reactor mechanism [25] to read the modules from that text file. We need this extension as the reactor API currently does not offer an option to specify the list of modules as a file[10]. To execute only the selected short- and long-running tests, we make use of Maven Surefire's test inclusion mechanism.

4 EVALUATION

The effectiveness of safe RTS techniques is typically evaluated by comparing the number of selected tests to the retest-all strategy and by measuring the overall test duration of the selected tests [10, 16, 21]. Regarding the evaluation of a technique's safety, prior work often semi-formally proves safety under certain assumptions [10, 16, 21], such as safety for code changes [16]. However, prior research on checking RTS tools has shown that these assumptions cannot be guaranteed to hold in practice [40]. We have already explained in Sec. 2.3 why non-code artifacts, cross-language links, and build system changes pose particular threats to the safety of existing RTS techniques. Thus, we need to empirically determine how safe our proposed RTS approach is for changesets of pull requests and discuss scenarios where our approach can be unsafe (see Sec. 4.3.2). Since no existing RTS technique is considered universally safe and could therefore serve as a reference, the only way to empirically check for safety violations is to find real (non-flaky) failures that were not selected by the RTS technique [34]. We perform an empirical study on five weeks of real development activity to answer the following research questions (RQs):

- **RQ**$_1$: How much testing effort reduction can we achieve by selecting tests using our RTS technique?
- **RQ**$_2$: How safe is our RTS technique for changesets of pull requests in terms of real missed failures?
- **RQ**$_3$: How much end-to-end CI pipeline execution time does our RTS technique save per pull request CI pipeline run?

4.1 Experimental Setup

In order to be able to answer RQ$_{1-3}$, we need to measure (1) testing effort reduction, (2) missed failures, and (3) end-to-end CI pipeline execution time savings for pull requests. We therefore added an invocation of our RTS CLI tool before the Java pipeline steps on the previous two release branches, R_{V_1} and R_{V_2}, and in the current release branch, R_{V_3}. Additionally, for all three branches we log the start and end timestamp for each Java pipeline step and the newly added RTS pipeline step. The build machines used for executing pull request pipelines are drawn from a fixed set of machines (64–96 cores), independent of the target branch of the pull request. Table 1 provides a summary of relevant descriptive statistics for the three release branches considered in this evaluation.

To answer RQ$_1$ and RQ$_2$ and measure (1) and (2), we only consider pull requests to R_{V_1} and R_{V_3}. On these branches, we still execute all tests and only store the set of selected tests for later

[10] Due to command line length limitations on Windows, we cannot use the `--projects` option.

Table 1: Relevant statistics for 3 considered release branches

Branch		R_{V_1}	R_{V_2}	R_{V_3}
Dataset Statistics	# PRs	88	58	251
	# PR pipeline runs	157	73	356
	# Commits	896	120	1,681
	Median changeset size (# files)	5	4	9

analyses (i.e., retest-all with *simulated* RTS). This way, we get a retest-all test report for each pull request CI pipeline run on these branches. From this report and the set of selected tests, we can compute the fraction of selected tests and test duration, and count the missed failures that occurred during retest-all. The rationale behind the choice of release branches is that at IVU release branches follow a specific lifecycle: On the current release branch, R_{V_3}, the most development activity is expected, as small features and bug fixes are still added. The less recent release branch, R_{V_1}, receives fewer development activity, which primarily concerns maintenance tasks, where pull requests with smaller changesets are expected. By using both R_{V_1} and R_{V_3} to answer RQ$_1$, we can also investigate to what extent RTS on pull requests to older release branches can achieve higher savings due to the smaller changesets.

To answer RQ$_3$ and measure (3), we need to compare end-to-end pipeline durations with RTS against retest-all. However, using both, RTS *and* retest-all on the same pull requests introduces large overhead—in the worst case a factor of 2, if all tests and modules are selected. In our industrial setting, this evaluation approach is rendered too expensive. From previous experience and analyses we know that CI pipeline runs take approx. the same time for pull requests to R_{V_2} and R_{V_3}, although they can be subject to natural variations due to infrastructure side-effects [12]. Hence, we can compare the time distribution for each pipeline step in R_{V_2}, where RTS is actually used, against R_{V_3}, where retest-all is used. In addition to this *inter-branch* evaluation approach, we validate our results by also using RTS on R_{V_3} for the final week of our experiments. This allows to investigate the pipeline runtimes before and after RTS activation in an *intra-branch* evaluation.

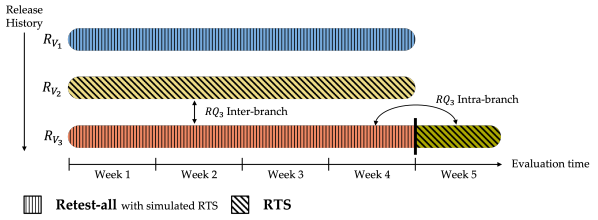


Figure 3: Evaluation approach for different release branches

The evaluation approach is illustrated in Fig. 3. Overall, our approach captures two aspects that have not been considered in prior research: First, we investigate the impact of the current lifecycle phase of a target release branch on RTS effectiveness. Second, we analyze how savings in each pipeline step contribute to the achieved end-to-end time savings for pull request CI pipeline runs.

4.2 Results

RQ$_1$: Testing Effort Reduction. Fig. 4 depicts the testing effort reduction we achieve, by comparing our RTS technique to a retest-all strategy. As described in the previous section, we compute the fraction of selected tests and test duration by combining retest-all test reports from pull requests on R_{V_1} and R_{V_3} with the respective sets of selected tests. The results indicate that our RTS technique selects on average 25% of tests for pull requests on R_{V_1} and 52% for R_{V_3}. The selected tests further take on average 28% and 58% of test duration for R_{V_1} and R_{V_3}, respectively. Hence, RTS was particularly effective for the maintenance branch R_{V_1}, which had smaller changesets (median changeset size is only roughly half of R_{V_3}).

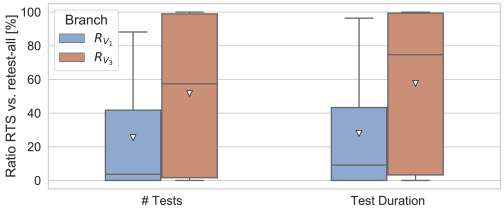


Figure 4: Comparison of our RTS technique to retest-all strategy regarding ratio of selected tests and test duration

> **RQ$_1$:** *We find that on two evaluation branches our RTS technique selects on average 25% and 52% of tests per pull request CI pipeline run realizing a test duration reduction by 72% and 42%, respectively. RTS performs significantly better in pull requests on a maintenance release branch (R_{V_1}) compared to a release branch with active development (R_{V_3}).*

RQ$_2$: Safety. To find real failures which would not have been selected by our RTS technique, we first create the set difference of all failed tests in the retest-all test report and the selected tests. This yields a total of 305 pull request CI pipeline runs with missed failures across pull requests to R_{V_3} and R_{V_1}. To filter out failures that are *not* introduced by the changeset of the pull request itself, we need to re-run the failing tests at the revision of the target branch which the pull request was rebased on. If a test also fails on the target branch, it is probably not related to the pull request (e.g., a *won't fix*) and can be discarded. If a test does *not* fail on the target branch, we need to manually check if it is a failure introduced by the pull request and was therefore missed by the RTS technique. To keep the tedious manual effort at a reasonable level, we randomly sampled 50 from the 305 runs and manually inspected 2,176 missed test failures. Most of the missed failures stem from a database technology switch that was made on the build machines during the considered time period. This switch caused many long-running tests to fail due to memory leaks or failing schema updates during the first days of operation. We further observe a few flaky tests that failed due to non-deterministic test behavior that is partially known to the developers. However, none of the inspected missed test failures are actually related to the changes introduced in the pull requests. We can therefore conclude that our RTS technique

is empirically safe regarding failures that were introduced by the considered pull requests. Yet, we discuss potential reasons for safety violations in Sec. 4.3.2. We can further confirm previous findings that RTS techniques can be helpful in avoiding flaky test failures and thereby reduce associated debugging costs [34].

> **RQ$_2$:** *We find that our RTS technique does not miss any real failures that are introduced in the considered pull requests.*

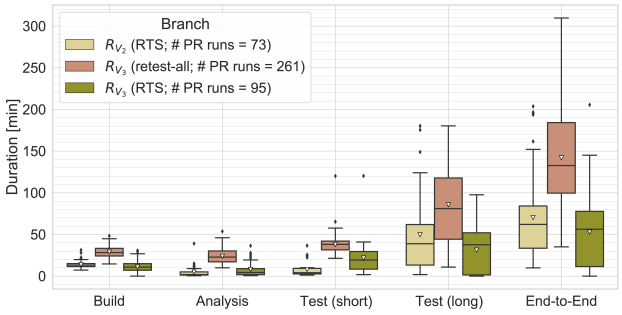


Figure 5: Inter- and intra-branch comparison of our RTS technique to full build and retest-all strategy

RQ$_3$: End-to-End Savings in Pull Requests. Fig. 5 shows the distributions of the duration for each Java pipeline step and the Java end-to-end runtime for R_{V_2} and R_{V_3}. As described in our experimental setup, we perform two kinds of comparisons, *inter-branch* and *intra-branch*. Regardless of the applied evaluation, we observe significant savings: In the inter-branch comparison, the results indicate that we can save on average 50% (71 minutes) of end-to-end pipeline runtime with RTS on R_{V_2}. In the intra-branch comparison, when comparing R_{V_3} (retest-all) against R_{V_3} (RTS), which was used in the final week of our experiments, we achieve even better results: On average, 63% (89 minutes) of end-to-end pipeline runtime is saved. Despite small discrepancies in the achieved time savings, our results show similar trends using either of the two evaluation techniques and thus confirm each other. Comparing the median time savings, there is only a discrepancy of 6 minutes between the evaluation methods.

Regarding the individual contributions to this overall end-to-end time, we report the following average savings for the individual pipeline steps for R_{V_2}: 53% for *Build*, 80% for *Analysis*, 79% for *Test (short)*, and 42% for *Test (long)*. We discuss the reasons for the comparatively smaller savings in long-running tests in Sec. 4.3.1.

We further find that computing the selected modules and tests is inexpensive: The mean and median of the RTS pipeline step was roughly 3 minutes across all three branches. Since we generate test traces and DLL-to-source mappings in *separate* CI pipelines every day, the end-to-end pipeline time for pull requests is unaffected.

> **RQ$_3$:** *We find that our RTS technique helps save on average 50% and 63% end-to-end pipeline execution time for pull requests on two release branches.*

4.3 Discussion

In the following, we discuss weaknesses related to the precision and safety of our RTS approach and share feedback we received from developers working on the system.

4.3.1 Imprecision of DLL-to-source Mappings. Our results for RQ$_3$ indicate that time savings achieved for the long-running Java test step are lower than those for the short-running test step. The reason is that if there are changes in core modules of the C/C++ subtree, commonly the majority of long-running tests is selected. This implies that these changes affect any DLL used by many long-running tests. We do not have any runtime information about which C/C++ source files that are part of a DLL are actually covered by each test. Hence, our selection in such cases is rather coarse-grained and imprecise. To address this problem and obtain more fine-grained runtime information, we are currently investigating extensions to our approach such as instrumenting the DLLs, intercepting native function invocations from the JNI, or using `DTrace` for tracing additional relevant system events.

4.3.2 Potential Reasons for Safety Violations. Test traces and DLL-to-source mappings are created in separate CI pipelines continuously running on the target release branches. Depending on the frequency of these pipelines (we run them once per day), test traces and DLL-to-source mappings might be outdated, which in turn may lead to unsafe test selection. Similarly, if a test fails over multiple runs in the separate tracing CI pipeline, that test's trace will not get updated until it passes again, also leading to outdated test traces.

In case of changes related to dependency injection mechanisms, affected tests might be missed: For instance, if a new default Java EE bean implementation is added inside a pull request, all tests that use the default bean will change their behavior. Yet, none of the files inside the test trace is directly affected by the addition. However, in such cases, typically another change that affects the test trace is part of the pull request, such as adjusting any other file that uses the newly added bean implementation—hence, odds are low that we effectively miss any affected tests.

Eventually, our RTS technique might be unsafe, if new non-code artifacts are added to the code base that are implicitly used by tests, while tests are not changed themselves; for instance, if a test walks the file tree and opens all files with a certain file extension, rather than explicitly opening a file through its filepath.

Finally, we found that external configuration changes, e.g., to the database environment, can cause tests to fail. However, our RTS technique only considers artifacts tracked by the VCS and therefore did not select these tests. We believe this to be expected RTS behavior, since these failures are not related to the changeset of a pull request.

4.3.3 Developer Feedback. Since IVU engineers, architects, and testers are directly impacted by our changes to the pull request pipelines, we regularly asked them for feedback on our work. Overall, our RTS approach has wide support among developers as it significantly reduces feedback times in their daily work; they are convinced that the RTS approach adds great value. Therefore, we deploy it to all other release branches as well. Furthermore, as requested by developers, we are working on extending the test

selection to C/C++ tests. While the system call tracing is language-agnostic and the integration with the testing framework is straightforward, we require further language-specific instrumentation, as DLL test trace granularity is too coarse (see our discussion on imprecise DLL-to-source mappings above).

4.4 Threats to Validity

4.4.1 External Validity. As for most industrial case studies, the main threat to validity concerns the generality of our results: We have specifically designed our RTS approach to address the shortcomings of existing techniques in the context of IVU. Nonetheless, our results show similar trends as prior RTS research done on open-source software and we can confirm empirical findings that dynamic file-level RTS can indeed significantly reduce regression testing efforts [10, 16, 34]. Furthermore, the technology we use to implement our RTS tool is publicly available and we rely on standard frameworks and tools, such as JUnit and Maven, that are frequently used across research [10, 16, 21, 38]. This eases a replication of our study in other software projects.

Furthermore, similar to previous studies [12, 34], the measured times in the CI pipelines can contain irregular fluctuations stemming from infrastructure or environment issues. While this could affect our evaluation results, we address this threat by reporting not only (potentially biased) average values, but the distributions for time savings across analyzed pull request pipeline runs.

Finally, to assess the safety our RTS approach, we manually checked if there were any real missed failures, but limited the inspection to 50 randomly sampled pull requests, which might not be representative. However, as opposed to most prior studies on safe RTS, we discuss potential safety violations and perform an empirical study to find any occurrences. We do this even though we rely on concepts that have been shown to work in other safe RTS approaches. Furthermore, to the best of our knowledge, we have still re-run and inspected the largest number of missed test failures in any existing RTS study to date.

4.4.2 Internal Validity. The main internal threats emerge from the implementation of our RTS tool and the proper functioning of Maven Surefire, JUnit, `DTrace`, and the ByteBuddy library[11], which we use to instrument the Java class loader. To address these threats, we wrote unit and integration tests for our RTS tool and manually checked selection results of pull requests for their validity.

5 RELATED WORK

Throughout this paper, we have referenced RTS techniques that have been proposed to effectively reduce regression testing efforts (see Sec. 2.3). Among the many existing studies, we consider the following to be most relevant for our work:

Gligoric et al. [15, 16] propose `Ekstazi`, a dynamic file-level RTS technique for the JVM that relies on file checksums for computing the set of selected tests. In Sec. 2.3, we describe why `Ekstazi` is unsafe in our context, as it uses file checksums and is neither aware of cross-language links, nor does it consider non-code artifacts[12]. `Ekstazi` reduces the end-to-end testing time on average by 32% across 32 open-source projects. Furthermore, Gligoric et al. [16] find that selecting tests at the class level (i.e., JUnit test suites) achieves better results than selecting tests at method level (i.e., JUnit test methods). We acknowledge these results as our RTS technique also collects file dependencies per test class, rather than test method.

Shi et al. [34] extend `Ekstazi` by complementing it with the static incremental build tool `GIB` [2]. The resulting tool, `GIBstazi`, is thus the most similar existing approach to our hybrid approach of selecting modules and regression tests for compilation and test execution, respectively. However, `GIBstazi` selects *all* tests, if any changes to non-code files occur. Due to the large number of non-code artifacts in our system, this is too imprecise. Additionally, similar to `Ekstazi`, it is unsafe for changes to multi-language source or binary files, such as DLLs, and files accessed by those. Overall, `GIBstazi` achieves higher safety than `Ekstazi` and the empirical results on 22 open-source projects show that `GIBstazi` reduces end-to-end build and testing time in CI environments by 23%.

Celik et al. [10] propose `RTSLinux`, the first and only RTS technique that uses system call analysis to track accesses to files across JVM boundaries during testing. Similar to `Ekstazi`, `RTSLinux` uses file checksums for selecting tests, when a compiled workspace already exists. We have alluded to why `RTSLinux` is not applicable in Windows environments and why our lightweight, safe kernel instrumentation through `DTrace` is a more practical approach in an industrial setting than modifying the operating system kernel. `RTSLinux` saves 53% of test execution time compared to retest-all.

In a previous study at IVU, we evaluated the cost-effectiveness of unsafe RTS techniques that solely rely on readily available CI and VCS metadata [12]. When applied to the main development branch at IVU for six weeks, the best performing unsafe RTS technique achieved test time savings of on average 19.8% with 93.4% of failures being detected. Though, we have motivated before that for pull requests to release branches safe RTS is required.

In summary, we are not aware of any prior work that investigates safe RTS that is build system aware and operates across language boundaries. Moreover, neither do any of the previous studies evaluate safe RTS in a large-scale industrial CI setting, nor do they study how end-to-end time for pull request CI pipelines can be reduced.

6 CONCLUSION

At IVU, compiling, analyzing, and testing pull requests within CI pipelines has prohibitively long feedback times. To reduce testing effort for pull requests on release branches, *safe* RTS is required, since support patches for customers are directly built from release branches. However, existing safe RTS techniques are inapplicable, as tests at IVU commonly operate across languages and intensively make use of non-code artifacts. Moreover, prior RTS approaches are unsafe for changes to the build system configuration and require an already fully compiled workspace.

In this paper, we introduce a build system aware multi-language RTS technique that safely selects modules and tests for compilation and execution. We deploy our novel RTS technique in IVU's large-scale, multi-language code base and perform an extensive empirical study to evaluate its effectiveness. The results indicate that our RTS technique saves on average 42% and 72% of testing time on

[11] ByteBuddy: https://bytebuddy.net

[12] `Ekstazi` has a hidden Linux-only option to collect files loaded by the JVM, which is untested and disabled by default. Nonetheless, even when collecting files loaded by the JVM, file accesses made from native DLLs or transitive processes are missed [10].

two evaluation release branches. We thereby reduce end-to-end CI pipeline runtime for pull requests by up to 63% on average. Since this greatly reduces feedback time for developers while retaining failure detection, our introduced RTS technique is now deployed company-wide to all release branches. While our industrial case study provides insights for one specific context, we expect our RTS technique to be applicable to other multi-language software projects, as it is based on well-known concepts and widely used tools for dynamic and static program analysis.

ACKNOWLEDGMENTS

We thank Dennis Bracklow, Stefan Golas, Maximilian Pohl, and Stefan Sieber for their support while integrating our technique into IVU infrastructure. This work was partially funded by the German Federal Ministry of Education and Research (BMBF).

REFERENCES

[1] 2017. Java Agent API. https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/package-summary.html
[2] 2021. gitflow-incremental-builder (GIB). https://github.com/gitflow-incremental-builder/gitflow-incremental-builder
[3] Apache Maven. 2021. Maven Surefire Plugin – surefire:test. https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html
[4] Atlassian. 2017. About test optimization. https://confluence.atlassian.com/clover/about-test-optimization-169119919.html
[5] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt, and Christian Stein. 2021. JUnit 5 User Guide: Advanced Topics. https://junit.org/junit5/docs/current/user-guide/{#}launcher-api-listeners-custom
[6] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *Proceedings of the International Conference on Software Engineering*. 550–561. https://doi.org/10.1145/2568225.2568248
[7] Antonia Bertolino, Antonio Guerriero, Roberto Pietrantuono, Stefano Russo, Breno Miranda, and Roberto Pietran-Tuono. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *Proceedings of the International Conference on Software Engineering*. 1–12. https://doi.org/10.1145/3377811.3380369
[8] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: An industrial case study. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 975–980. https://doi.org/10.1145/2950290.2983954
[9] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. https://doi.org/10.5555/1247415.1247417
[10] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression test selection across JVM boundaries. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 809–820. https://doi.org/10.1145/3106237.3106297
[11] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 235–245. https://doi.org/10.1145/2635868.2635910
[12] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 491–504. https://doi.org/10.1145/3460319.3464834
[13] Kurt Fischer, Farzad Raji, and Andrew Chruscicki. 1981. A Methodology for Retesting Modified Software. In *Proceedings of the National Telecommunications Conference*. 1–6.
[14] Kurt F. Fischer. 1977. A test case selection method for the validation of software maintenance modifications. In *Proceedings of International Computer Software and Applications Conference*. 421–426.
[15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the International Conference on Software Engineering*. 713–716. https://doi.org/10.1109/icse.2015.230
[16] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*. 211–222. https://doi.org/10.1145/2771783.2771784
[17] Brendan Gregg. 2016. DTrace for Linux. http://www.brendangregg.com/blog/2016-10-27/dtrace-for-linux-2016.html
[18] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional.
[19] Mary Jean Harrold, Alessandro Orso, James A. Jones, Tongyu Li, Maikel Pennings, Saurabh Sinha, Ashish Gujarathi, Donglin Liang, and S. Alexander Spoon. 2001. Regression test selection for Java software. *ACM SIGPLAN Notices* 36, 11 (2001), 312–326. https://doi.org/10.1145/504311.504305
[20] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Soder, Agneta Nilsson, and Magnus Castell. 2015. Supporting Continuous Integration by Code-Churn Based Test Selection. In *Proceedings of the International Workshop on Rapid Continuous Software Engineering*. 19–25. https://doi.org/10.1109/rcose.2015.11
[21] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 583–594. https://doi.org/10.1145/2950290.2950361
[22] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *Proceedings of the International Conference on Automated Software Engineering*. 949–954. https://doi.org/10.1109/ase.2017.8115710
[23] Hareton K.N. Leung and Lee White. 1989. Insights into regression testing. In *Proceedings of the International Conference on Software Maintenance*. 60–69.
[24] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018
[25] Apache Maven. 2021. Maven Multi-Module Projects. https://maven.apache.org/guides/mini/guide-multiple-modules.html
[26] Microsoft. 2019. Visual Studio C++ Project system extensibility and toolset integration – .tlog files. https://docs.microsoft.com/en-us/visualstudio/extensibility/visual-cpp-project-extensibility?view=vs-2019{#}tlog-files
[27] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. 2011. Regression testing in the presence of non-code changes. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*. 21–30. https://doi.org/10.1109/icst.2011.60
[28] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the Performance of Maven's Test Isolation: Experience Report. In *Proceedings of the International Symposium on Software Testing and Analysis*. 249–259. https://doi.org/10.1145/3395363.3397381
[29] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 241–251. https://doi.org/10.1145/1029894.1029928
[30] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagppan. 2019. FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services. In *Proceedings of the International Conference on Software Engineering*. 408–418. https://doi.org/10.1109/icse.2019.00054
[31] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262
[32] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. 2000. Regression test selection for C++ software. *Software Testing, Verification and Reliability* 10, 2 (2000), 77–109. https://doi.org/10.1002/1099-1689(200006)10:2<77::AID-STVR197>3.0.CO;2-E
[33] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. In *Proceedings of the International Conference on Software Engineering*. 689–699. https://doi.org/10.1109/ICSE.2017.69
[34] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *Proceedings of the International Symposium on Software Reliability Engineering*. 228–238. https://doi.org/10.1109/issre.2019.00031
[35] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 12–22. https://doi.org/10.1145/3092703.3092709
[36] Andrew S. Tanenbaum and Herbert Bos. 2015. *Modern operating systems*. Pearson.
[37] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430
[38] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the International Conference on Software Engineering*. 199–209. https://doi.org/10.1145/3180155.3180198
[39] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2013. FaultTracer: A spectrum-based approach to localizing failure-inducing program edits. *Journal of Software: Evolution and Process* 25, 12 (2013), 1357–1383. https://doi.org/10.1002/smr.1634
[40] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *Proceedings of the International Conference on Software Engineering*. 430–441.

A.3.2. BinaryRTS: Cross-language Regression Test Selection for C++ Binaries in CI

BinaryRTS: Cross-language Regression Test Selection for C++ Binaries in CI

Daniel Elsner
Technical University of Munich
Munich, Germany

Severin Kacianka
Technical University of Munich
Munich, Germany

Stephan Lipp
Technical University of Munich
Munich, Germany

Alexander Pretschner
Technical University of Munich
Munich, Germany

Axel Habermann
IVU Traffic Technologies
Berlin, Germany

Maria Graber
IVU Traffic Technologies
Berlin, Germany

Silke Reimer
IVU Traffic Technologies
Berlin, Germany

***Abstract*—Continuous integration (CI) pipelines are commonly used to execute regression tests before pull requests are merged. Regression test selection (RTS) aims to reduce the required testing effort and feedback time for developers. However, existing RTS techniques are imprecise for tests with cross-language links to compiled C++ binaries or unsafe if tests use external files. This is problematic because modern software in fact involves several programming languages and (non-)code artifacts such as configuration files. In this paper, we present BINARYRTS, a novel RTS technique that leverages dynamic binary instrumentation to collect the covered functions and accessed external files for each test. BINARYRTS then selects tests depending on changes issued to C++ binaries or external (non-)code artifacts. When evaluating BINARYRTS in our large-scale industrial context, we are able to exclude on average up to 74% of tests without missing real failures. We release BINARYRTS as the first publicly available RTS tool for software involving C++ code.**

***Index Terms*—Software testing, regression test selection, C++, cross-language links, multi-language software, non-code artifacts**

I. INTRODUCTION

Regression testing is a software testing activity that checks if changes have negatively impacted existing system behavior [1]. In modern development practices, continuous integration (CI) pipelines are commonly used to regularly build the software and run its regression test suite [2]–[4]. The most straightforward testing strategy is *retest-all*, which executes every test case after each change. However, if fast feedback for developers is crucial or testing resources are limited, executing all tests from a large test suite is often prohibitively costly [5], [6]. To address this problem, regression test selection (RTS) has been studied since the 1970s [7] to reduce the testing effort by only running a subset of test cases [2]–[4], [8]–[17]. An RTS technique is considered *safe*, if this subset of test cases contains all tests that potentially expose a fault [18].

At IVU Traffic Technologies[1], CI pipelines execute a regression test suite consisting of more than 25,000 unit, integration, and system tests written in C++ and Java before pull requests are merged into release branches. However, running the full test suite for each pull request yields intolerable feedback times of several hours, despite a high degree of test parallelization. Therefore, we have developed and successfully deployed a file-level RTS technique for Java tests at IVU in prior work [19]. Yet, due to the complex nature of the multi-language code base at IVU, two problems remained unsolved: first, the majority of the 13.5 million lines of code (LOC) and the test suite is written in C++. These C++ regression tests are not supported by our current RTS solution, as file-level techniques are impractical for languages that compile to large binary files [20]. Second, there exist several thousand Java tests that use the Java Native Interface (JNI) to interact with dynamic-link libraries (DLLs) built from the C++ code. Hence, if any C++ source file that is part of a DLL changes, every test accessing this binary file is selected. In short, file-level per-test execution traces are too imprecise when Java tests use cross-language links to C++ binaries [19].

Although several language-agnostic, yet inherently unsafe RTS approaches are reportedly used in industry [2], [4], [17], [21], research on safe RTS for C++ software is relatively sparse: while most early RTS research considered (binary) compiled languages such as C or C++ [8], [9], [18], [22]–[25], recently proposed RTS techniques focus on Java [11]–[13], [20], [26]–[28]. Since C++ itself, the size and frequency of regression testing, as well as development tool chains have significantly evolved, insights on the design and benefit of RTS in modern, large-scale industrial C++ software are largely missing [29]. To our knowledge, in the past decade, only two published studies proposed RTS techniques for C++ software [29], [30]. However, these techniques (1) are only suitable for C++ projects using the LLVM [31] compiler infrastructure, (2) do not cope with cross-language links to C++ binaries, (3) ignore changes to external files, *e.g.,* non-code artifacts, and (4) either do not support dynamic linking of libraries or operating systems other than Linux [29], [30]. New approaches for RTS in modern C++ software and their industry-scale evaluation are therefore essential to address these gaps in research and practice [32].

In this paper, we present BINARYRTS, a novel RTS technique for software using C++ binaries throughout the testing process. The analyzed tests can be written in C++ or any other

[1]IVU Traffic Technologies is one of the world's leading providers of public transport software solutions.

language with interoperability to native binaries, *e.g.,* Java tests with cross-language links using the JNI. BINARYRTS leverages dynamic binary instrumentation to collect (1) covered functions and (2) accessed external files for each test. This allows more accurate and reliable test selection, as changes to C++ binaries, non-code artifacts, or source files in other (domain specific) languages, can be properly attributed to affected tests. The instrumentation and analysis within BINARYRTS is compiler-agnostic, supports C and C++ binaries out-of-the-box, and can be transferred to different platforms as well as operating systems, and other compiled languages.

We evaluate BINARYRTS in IVU's large-scale CI infrastructure by analyzing 385 pull requests across two release branches covering more than 1,000 commits. To investigate saved testing effort with BINARYRTS, we measure the test selection ratio for the C++ test suite and the cross-language Java test suite. Our results show that BINARYRTS selects on average 26%–37% of C++ tests and 57%–64% of Java tests. BINARYRTS never fails to select tests that reveal actual regressions in the studied pull requests. Due to these promising results, IVU is currently deploying BINARYRTS to all release branches. We provide BINARYRTS as the first publicly available C++ RTS tool to foster regression testing research on C++[2].

II. TESTING C++ PULL REQUESTS AT IVU

In the following, we explain the system under test and the testing process for C++ pull requests at IVU. We also elaborate on the few existing solutions for RTS in C++ software and their drawbacks in the given context.

A. System Description

The source code for the main IVU software resides in a monolithic repository. There are two major subtrees in the repository, one with mainly C++ (and some C) sources ($9.5M$ LOC) and one with mainly Java code ($4M$ LOC). Besides these two main general purpose programming languages (GPLs), IVU makes significant use of non-code artifacts, such as CSV or plain text files, as well as other GPLs and domain specific languages (DSLs), *e.g.,* TypeScript or XML.

While the Java source code is generally structured and built using Maven [33], the C++ sources are compiled through a self-maintained meta build tool (called BT hereafter). BT wraps Microsoft's C++ compiler toolchain [34], as most IVU software products target Microsoft Windows.

The C++ subtree contains code that compiles into 300+ executable binaries, including 200+ test executables and various applications. Alongside, 700+ binary DLLs are built from the subtree, which are linked against test executables and applications, both, at load-time and run-time[3].

For regression testing, unit, integration, and system tests are written in C++ or in Java. C++ tests are written using GoogleTest [35] and reside in the C++ subtree, whereas Java tests use JUnit [36] and reside in the Java subtree. However, since parts of the persistence logic are only implemented in C++, many Java tests heavily use the JNI to call C++ code for populating the database. In production, Java and C++ runtimes are usually separated. They are intertwined via JNI only to achieve two goals: (1) synchronization between the data generating parts in C++ and the data consuming parts in Java by using a single thread, and (2) allowing the test to run on an open database transaction which can easily be rolled back instead of having to commit and remove test data. For instance, a Java test might initiate a database session from within the Java virtual machine (JVM) process, hand the pointer to the session over to native code via JNI, load a few dozens DLLs at run-time and use them to generate test scenarios in the database, return control to the JVM to continue test case execution; then, update the test scenario by invoking native code again, return to the test case in the JVM, and finally clean up the database session from native code. This way, even complex test scenarios can be tested in a close-to-reality environment, which reduces the risk of late integration issues during release testing. However, these complex tests come at considerable costs: running the C++ and Java test suite takes up to 3 hours, despite high parallelization. This suggests to use RTS for more cost-effective, change-oriented testing.

In previous work, we discussed our build system aware multi-language RTS approach which we successfully deployed at IVU [19]. It uses system call analysis to trace file accesses and already takes into account that a test's outcome can be affected by changes to source code of multiple programming languages, non-code artifacts, and build system configuration files. However, the approach is limited to Java tests and imprecise in the case of C++ changes due to the analysis at file-level granularity: in case a DLL is changed, all Java tests that access the DLL during testing are selected, even if they do not execute the changed C++ code. At IVU, more than 16,000 regression tests are either part of binary executables or use DLLs at run-time. Hence, this paper focuses on precise RTS for C++ and Java tests that use C++ binaries, *i.e.,* executables or DLLs.

B. C++ Pull Request CI Pipeline

IVU usually provides support for the last ten released versions of their software products. Therefore, release branches are maintained next to the main development branch. Whenever developers want to integrate changes into any of these branches, they create a pull request. For each pull request, a CI pipeline is created that builds, analyzes, and tests the introduced changes.

If a pull request comprises changes related to the C++ subtree, the C++ code is analyzed, built, and tested through BT. Since a full C++ build, even with high parallelization, takes roughly an hour, BT uses a shared remote compile cache and only compiles binaries depending on changed sources. BT also steers the C++ test execution and only runs those C++ test executables that are either directly or transitively affected by the changes. Yet, this *module-level* test selection is too coarse-granular: even very small changesets often require running thousands of tests.

[2]BINARYRTS on GitHub: https://github.com/tum-i4/binary-rts

[3]We distinguish between *run-time* (during execution), *run time* (timespan taken by a run), and *runtime* (program execution environment) in this paper.

Since many Java tests also make use of C++ binaries built from the C++ subtree, more precisely DLLs, these tests should also be executed for changes to the C++ subtree. In the case of changes to the C++ *and* Java subtree, the Java tests are currently selected by the established file-level RTS approach for Java tests [19]. However, for *C++-only* changesets, running selected Java tests has recently been deactivated due to the significant time overhead even for small changesets, since the selection is too imprecise (see Sec. II-A). Although developers are encouraged to check the main CI pipelines of the target branch within the next day after their pull request has been merged, this imposes the risk of missing failures and, even worse, bugs slipping into release branches.

Fig. 1 summarizes how both Java and C++ tests can be affected by changes to binaries built from the C++ code base.

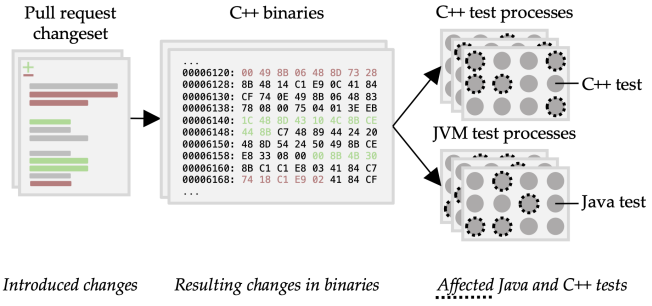


Fig. 1. C++ pull requests affecting Java and C++ tests

C. Existing C++ Test Selection Approaches

Over the past roughly 45 years, numerous RTS techniques have been proposed that harness static or dynamic program analysis at the level of basic blocks [9], [10], [37], functions [13], [26], [29], [30], classes or files [11]–[13], [19], [20], [27], [28], [38], or modules [14], [38], [39]. In the following, we iterate over RTS approaches for C++ software and outline why they are not applicable at IVU.

Early RTS research targeted compiled languages such as C [8], [22], [40] or C++ [9], [23]–[25], [41]. However, the described approaches were either never actually implemented [9], [41] or evaluated on small programs with a few unit tests rather than industrial-scale C++ projects [22], [24], [25]. Furthermore, most of the used analysis tools are not available today or cannot analyze modern C++ code bases, since the language and compilers have significantly evolved [29].

To our knowledge, in the past decade only two RTS approaches targeting C++ software have been presented, which have the following limitations in the given context: Fu *et al.* [29] propose RTS++, a static, function-level RTS technique. RTS++ relies on function call graphs constructed from LLVM bitcode and therefore can only analyze C++ projects targeting LLVM. Thus, RTS++ is not applicable to IVU's C++ code base, where compiling with `clang` [42] is possible, but linking can only be done using Microsoft's C++ compiler toolchain[4]. Yet, more importantly, RTS++ requires linking all libraries statically into a single binary test executable. While this may not be an issue in the comparatively small open-source projects which RTS++ has been evaluated on, it is infeasible at the scale of IVU, where many test executables and Java tests use the same hundreds of dynamically linked libraries.

To perform RTS for integration testing of distributed, large-scale C++ web services at Google, Zhong *et al.* [30] develop TESTSAGE, a dynamic, function-level RTS technique. Similar to RTS++, TESTSAGE is limited to LLVM-based projects that run on Linux or a few BSD descendants [43], whereas IVU targets the Windows operating system. TESTSAGE is further built on top of Google-internal infrastructure and code analysis tooling (*e.g.,* PIPER), which arguably limits transferability.

Besides, none of the proposed RTS techniques for C++ software has considered changes to non-code artifacts or source code of languages other than C or C++, even though they might affect test behavior [19], [20], [44]. Next to these conceptual and technical limitations, we did not find any publicly available tools that implement these RTS techniques.

In summary, we require an RTS technique that is (1) capable of analyzing tests which use arbitrary binaries at run-time, with the flexibility to support different compilers and operating systems; (2) aware of changes to non-code artifacts or source code of programming languages other than C or C++ that may affect test behavior. This motivates BINARYRTS, a novel dynamic RTS technique, which we describe in the next section.

III. BINARYRTS TECHNIQUE

This paper introduces BINARYRTS, the first RTS technique that harnesses dynamic binary instrumentation to reliably select affected regression tests that use C++ binaries or access external files, *e.g.,* source files from other languages or non-code artifacts. In the following, we first explain how BINARYRTS dynamically analyzes and instruments C++ and system binaries to generate per-test execution traces (*i.e.,* test traces), that include covered functions as well as accessed files. Second, we elaborate on the change-based test selection performed for C++ pull requests. Last, we explain how BINARYRTS has been integrated into IVU's CI test infrastructure.

A. Dynamic Binary Analysis

In order to implement any *dynamic* RTS technique, run-time information about tests is required, *i.e.,* per-test execution traces. To analyze the run-time behavior of a program, the target program needs to be *instrumented* or run in a monitored environment. Instrumentation refers to analysis code added to the program, which is executed as part of the normal program execution [45], [46]. Programs can either be instrumented statically, before the program runs, or dynamically, at program run-time. Instrumentation code can further be added through source code analysis or binary analysis. While the former is typically specific to the language and compiler, the latter is language- and compiler-agnostic but often more difficult to implement and more expensive in terms of run time overhead [45]. Yet, over the past two decades, instrumentation

[4]There is an ongoing effort to improve `clang`'s compatibility with MSVC projects: https://clang.llvm.org/docs/MSVCCompatibility.html

frameworks such as DynamoRIO [46], Intel PIN [47], or Valgrind [45] have evolved that ease the implementation of more efficient dynamic binary analysis (DBA) tools.

Nonetheless, existing dynamic RTS solutions rely on static source code analysis [30]. BINARYRTS is thus the first proposed RTS technique that leverages DBA to obtain per-test execution traces. In its current implementation, BINARYRTS relies on DynamoRIO [46], a popular and mature DBA framework [48], which supports a variety of operating systems and processor architectures, including Windows and x86-64, the primary target at IVU. DynamoRIO provides powerful application programming interfaces (APIs) to analyze and instrument basic blocks and to trace system call invocations. Therefore, DynamoRIO acts as a process virtual machine, by taking over control of the process executing the binary; it then creates and maintains a so-called *code cache* which contains a copy of the original code from the binary augmented by any added instrumentation code. Furthermore, DynamoRIO allows defining callback functions that are called whenever a new binary *module*[5] is pulled into the process [46]. This way, BINARYRTS can also analyze and instrument basic blocks from all DLLs that are dynamically loaded during execution. Since this flexibility naturally introduces run time overhead, we discuss performance considerations and implementation issues in Sec. IV-C.

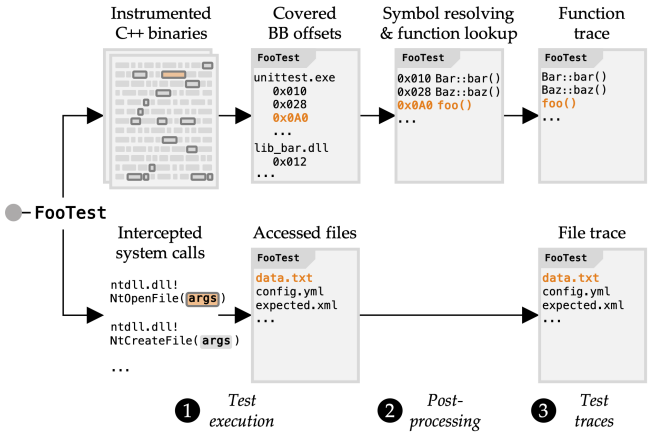


Fig. 2. Process for BINARYRTS instrumentation and test trace collection with **exemplary** covered basic block (BB) and accessed file

Fig. 2 illustrates how BINARYRTS obtains per-test execution traces using DBA: first, during *test execution* ❶, when a basic block is loaded into the code cache, BINARYRTS instruments it by (1) calculating its relative offset to the enclosing module's start address, (2) storing the triple $(module\ id, offset, hit\ count)$ in a table-like data structure, which keeps track of covered basic blocks, and (3) adding a single instruction at the beginning of the basic block to increment the hit count. Since DynamoRIO only loads basic blocks into the code cache when they are first executed or after cache invalidation, no irrelevant basic blocks are instrumented. Furthermore, BINARYRTS registers a *dump event listener* to write the basic block table to an output file (see *Covered BB offsets* in Fig. 2), either upon receiving the process exit event or a custom dump event. This can be triggered from the target program, *e.g.,* after each test case (see Sec. III-C). Alongside, BINARYRTS sets up interceptor functions that are called before system calls related to file accesses are invoked. BINARYRTS then extracts the requested file path from the provided system call arguments and stores it in a vector which is also written to a file by the dump event listener (see *Accessed files* in Fig. 2).

Second, in the *post-processing* stage ❷, all covered basic block offsets are resolved by querying the debug symbols using DynamoRIO's symbol access library. Note that BINARYRTS does not require debug builds, but merely debug symbols generated during compilation. These allow determining the source line a basic block offset corresponds to. Nevertheless, coverage will be more precise with debug builds. At IVU, we use release builds with function inlining disabled, to reliably detect all covered functions. Once the source line information has been obtained, source lines need to be mapped to C++ functions, both member or non-member functions. BINARYRTS currently uses the popular utility program `ctags` [49] to efficiently obtain C++ function declarations and definitions without the need for a preprocessor or compiler. We discuss extensions for compiler-specific function parsing in Sec. IV-C. BINARYRTS also supports resolving symbols during test execution, but shifting the work to a post-processing stage has been significantly more efficient at IVU.

Last, once each covered basic block has been resolved to its enclosing function, we generate per-test execution traces in a third step ❸. These traces contain the functions and external files a single test is associated with. Hereby, a function is stored with the attributes *file*, *signature* (*name* and *parameters*), *class* (optional), *namespace* (optional), and *meta-data* (*e.g.,* start/end line, `virtual`, `static`). A function's *identifier* is constructed by concatenating the optional scope attributes, namespace and class, and the function signature, *e.g.,* `ivu::Foo::bar(int x)`. These test traces can then be used to select tests in pull requests, as we describe next.

B. Changed-based Test Selection

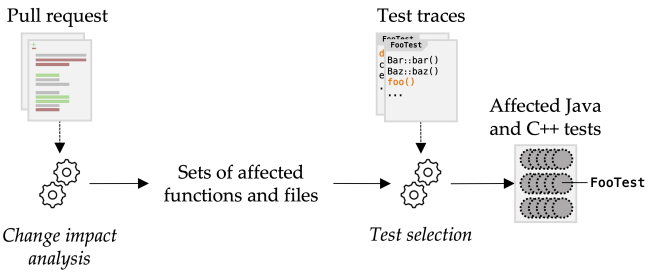


Fig. 3. BINARYRTS change impact analysis and test selection

Each pull request contains a set of changes, including additions, deletions, or modifications of files. Fig. 3 depicts that

[5]We stick to the DBA terminology, by collectively referring to binaries such as executables or DLLs as *modules*.

the analysis of a pull request changeset triggers the computation of affected functions and files. By combining these affected entities with the provisioned test traces, BINARYRTS can select the set of affected test cases.

The way BINARYRTS analyzes changesets is loosely inspired by Rothermel *et al.* [9] and Vokolos and Frankl [50], who use the Unix `diff` utility to locate differences between two program versions: we use `git diff` [51] to determine added, deleted, or modified files between the pull request branch (*new revision*) and the target release branch (*old revision*). Then, we run the change impact analysis shown in Algorithm 1 (simplified for presentation purposes) to compute affected functions and files, and finally derive the selected tests.

Algorithm 1: Change Impact Analysis and Test Selection

```
   Input: Changeset, Test Traces
   Output: Selected Tests
1  functions ← {}
2  files ← {}
3  foreach file ∈ changeset do
4      if isCppFile(file) then
5          if isAdded(file) then
6              newFunctions ← getFunctions(file.newRev)
               /* account for build system changes   */
7              functions +← stripFile(newFunctions)
8          else if isDeleted(file) then
9              functions +← getFunctions(file.oldRev)
10         else
11             functions +← findAffectedFunctions(file.oldRev,
                file.newRev)
12     else
           /* add external files, e.g., XML or SQL   */
13         files +← file
   /* query test traces for affected tests          */
14 tests ← findAffectedTests(files, traces)
15 tests +← findAffectedTests(functions, traces)
16 return tests
```

The algorithm iterates over each file in the changeset and checks if the file has a C++ file extension (*e.g.,* `.h` or `.cpp`). If not, the file is added to the set of affected files, since every test accessing this external file should be selected. If yes, the algorithm distinguishes between added, deleted, or modified files in the `git` repository. For added and deleted files, all functions in the file are added to the set of affected functions. Yet, for added files, these functions cannot exist in the test traces. Therefore, we strip the *file* attribute of the added functions to mark all functions with similar signature as affected. The rationale behind this over-approximation is that BINARYRTS aims to be agnostic about the build system. If a new source file F_{new} is added, which contains an implementation of function f, f could already be implemented in an existing source file F_{old}. Based on its configuration, the build system decides which source files to compile. Hence, if the configuration was changed to compile F_{new} instead of F_{old}, tests covering f will use the new implementation and are thus affected.

Algorithm 2: Finding Affected Functions

```
   Input: Old (oldRev) and New (newRev) File Revision
   Output: Affected Functions
1  Function findAffectedFunctions(oldRev, newRev):
2      affected ← {}
       /* find modified or newly added functions    */
3      foreach f_new ∈ getFunctions(newRev) do
4          isAddedFunction ← true
5          foreach f_old ∈ getFunctions(oldRev) do
6              if f_old.identifier = f_new.identifier then
                   /* functions with changed body   */
7                  if hasBodyChanged(f_old, f_new) then
8                      affected +← f_new
9                  isAddedFunction ← false
10                 break
11         if isAddedFunction then
               /* new overloading function          */
12             if hasParameters(f_new) then
13                 affected +← stripParameters(f_new)
               /* new virtual overriding function   */
14             if isVirtualOverride(f_new) then
15                 affected +← replaceClass(f_new, ∗)
               /* new scope overriding function     */
16             else if ¬ hasGlobalScope(f_new) then
17                 affected +← stripScope(f_new)
       /* find deleted functions                    */
18     foreach f_old ∈ getFunctions(oldRev) do
19         isDeletedFunction ← true
20         foreach f_new ∈ getFunctions(newRev) do
21             if f_old.identifier = f_new.identifier then
22                 isDeletedFunction ← false
23                 break
24         if isDeletedFunction then
25             affected +← f_old
26     return affected
```

For modified files, more elaborate change impact analysis is required as depicted in Algorithm 2: first, we iterate over all functions from the new and old revision of the modified file to determine all *modified* functions. Therefore, we check whether the function body has changed by comparing the code inside the body for textual equivalence, excluding comments and whitespaces. Due to static and dynamic dispatch in C++, a newly *added* function can affect the run-time program behavior, even in the absence of other changes, such as a modification of an existing function:

- **Function Overloading:** If a new function is added that has the same name as an existing function, but different parameters (*e.g.,* `foo(int)` and `foo(short)`), the compiler determines and uses the most suitable function at each call site. BINARYRTS therefore marks functions with the same name as the added one as affected. We limit ourselves to functions in the *same file*, since marking *all* functions with the same name can lead to high imprecision.
- **Virtual Function Overriding:** If a new member function is added to class B that overrides a virtual member function in B's parent class A, due to dynamic dispatch, all uses of the parent's member function need to be marked as affected. BINARYRTS thus marks all member functions with similar signature of any class (or struct) as affected.
- **(Scope) Function Overriding:** C++ allows defining functions in global, class, namespace, or local scopes and if multiple functions with similar signature exist in different scopes, it is up to the compiler to decide at each call site which function to call. Thus, if a new non-global function is added, BINARYRTS will by default mark all functions with similar signature as affected.

We also mark all *deleted* functions as affected to select all tests that previously executed the deleted function.

By handling the scenarios for static and dynamic dispatch as described, we deliberately design BINARYRTS to prefer safety over precision. We still remain flexible by not requiring more elaborate (and costly) compiler-specific static analysis which might be more accurate [9]. However, BINARYRTS provides run-time options to skip these over-approximations to trade increased RTS precision for reduced safety.

In addition, BINARYRTS has a run-time option to handle changes to *non-functional* code entities (*e.g.,* macros, global/member variables) [22]: we use `ctags` to locate all non-functional entities inside a C or C++ file, determine if they have changed, and then, similar to White *et al.* [25], perform a text-based lookup for (calling) functions that use the changed entity. These functions are then added to the set of affected functions. We further discuss this run-time option in Sec. IV-C1.

We list all run-time options in Sec. IV-A3 and evaluate them regarding their impact on safety and precision in our empirical study at IVU (see Sec. IV).

Finally, the affected tests are computed by querying the test traces with the affected functions and files. BINARYRTS uses efficient hash tables to minimize the time for finding tests that use affected functions or files. We provide measurements for the run time of the change impact analysis and test selection in Sec. IV-C3.

C. Integration into Pull Request CI at IVU

We integrated BINARYRTS into IVU's infrastructure as follows: to obtain test traces, we created new test tracing pipelines for each release branch considered in our evaluation (see Sec. IV). These tracing pipelines run all C++ and Java tests during off-peak hours (at night or on the weekend) with BINARYRTS's instrumentation enabled. For C++ tests, we add a GoogleTest test listener to BINARYRTS, which triggers a dump event after test setup and for every test case. BINARYRTS supports all GoogleTest test case types, including value- or type-parameterized tests [29], accounts for changes to (global) test setup code, and always selects newly added test cases. For Java tests, we use a Java agent [52] to attach BINARYRTS to the JVM process before the JUnit test suite starts. Similar to prior research [19], [20], [53], [54], we run each JUnit test suite in a forked JVM process for better test isolation and reliability. As the covered basic blocks will be dumped upon receiving the JVM process exit event, we do not need to trigger custom dump events for Java tests.

Once the test traces have been created for a release branch, they are serialized and stored on a network drive and can then be used inside pull requests for this release branch.

IV. EVALUATION

To evaluate BINARYRTS in a real-world industrial context, we perform a large-scale study in IVU's CI infrastructure. Our goal is to empirically determine the cost-effectiveness of BINARYRTS in terms of saved testing effort and how many real test failures BINARYRTS fails to select. In addition to the commonly used retest-all baseline, we compare BINARYRTS to IVU's internal module-level C++ test selection (BT) and our DLL-level Java test selection from prior work [19]. We further aim to understand precision and safety trade-offs for different run-time options of BINARYRTS (see Sec. III-B). Overall, we seek to answer the following research questions (RQs):

- **RQ**$_1$: How much testing effort can BINARYRTS save for C++ tests compared to retest-all and module-level RTS?
- **RQ**$_2$: How much testing effort can BINARYRTS save for cross-language Java tests compared to DLL-level RTS?
- **RQ**$_3$: How safe is BINARYRTS for changesets of pull requests in terms of real missed test failures?

A. Experimental Setup

1) Evaluation Branches: To conduct our experiments, we first pick two release branches, one rather old release branch receiving mainly maintenance changes (R_M) and one recent release branch with ongoing feature development (R_D). As shown in prior work, there can be significant differences in RTS effectiveness depending on the type of the release branch [19]. For both branches, we set up separate test tracing CI pipelines which run in off-peak hours, as described in Sec. III-C. Then, we modify the pull request pipelines for R_M and R_D: inside each pull request run[6] we invoke BINARYRTS to compute selected C++ and Java tests using the most recent test traces for the respective target release branch.

2) Evaluation Metrics: For each pull request run, we measure the reduction in testing effort by comparing (1) the number of selected tests and (2) their cumulative duration against the set of tests selected by a baseline regression testing strategy. Related research usually compares RTS techniques

[6]Recall that a new CI pipeline *run* is triggered whenever the pull request is updated, *i.e.,* if one or more commits are added.

against a retest-all testing strategy [19], [29], [30], [55], which we adopt for RQ_1. However, since the state-of-practice at IVU is better reflected by BT's module-level RTS strategy for C++ tests (RQ_1) and our DLL-level RTS strategy for Java tests (RQ_2), we add these as more realistic baseline strategies.

For RQ_1, we also report how often BINARYRTS excludes entire test executables. IVU's integration tests require a costly (global) database setup, which is performed when the process is started. Thus, skipping an entire test executable can significantly reduce overall test time.

Regarding RQ_2, recall that for *C++-only* pull requests, executing Java tests has currently been deactivated in the pull request CI pipelines, due to high execution times even with DLL-level RTS (see Sec. II-B). Yet, to perform our evaluation, we require the actual test verdicts and run time of Java tests. Therefore, we run the missing Java tests for *C++-only* pull requests in off-peak hours (at night and on the weekend) as selected by our DLL-level RTS strategy.

An RTS technique is considered safe, if it selects all tests that potentially expose a fault [18]. While safety for existing RTS techniques has been (semi-)formally proven under the assumption of code changes [9], [20], [27]–[29], prior research has shown that outdated test traces [19], [30] or changes related to non-code artifacts or cross-language links [14], [20], [44], [56] can compromise RTS safety. Hence, to perform a fair evaluation of RTS safety, we need to inspect all test failures that were not selected by BINARYRTS to understand if they actually reflect real regressions introduced in the respective pull requests. We discuss practical challenges in distinguishing between real regressions and flaky failures in Sec. IV-B.

3) BINARYRTS *Configuration:* Prior research has not yet investigated how testing effort and safety are affected by different levels of change impact analysis depth (see our discussion on static and dynamic dispatch in Sec. III-B). We are particularly interested in these trade-offs, as they provide practical guidelines on how to calibrate BINARYRTS during operation. Therefore, we run and compare BINARYRTS with the following run-time option configurations:

- B_{slim}: Disables all run-time options
- $B_{overload}$: B_{slim} + function overloading analysis
- $B_{override}$: B_{slim} + function overriding analysis
- $B_{virtual}$: B_{slim} + virtual function overriding analysis
- $B_{non\text{-}functional}$: B_{slim} + non-functional entity analysis
- B_{full}: BINARYRTS default; enables all run-time options

B. Results

We collect the dataset for evaluating BINARYRTS across four weeks of development, covering a total of 385 pull requests with 587 CI pipeline runs and more than 1,000 commits. Pull requests for R_D are more common (288) and have a larger median changeset size of 9 files than pull requests on R_M (97) with a median of 2 changed files.

RQ_1: C++ Testing Effort: Fig. 4 shows the distribution of the ratio of selected C++ tests compared to retest-all across all pull request runs on the two branches R_M and R_D for different BINARYRTS configurations as well as BT. Note that while

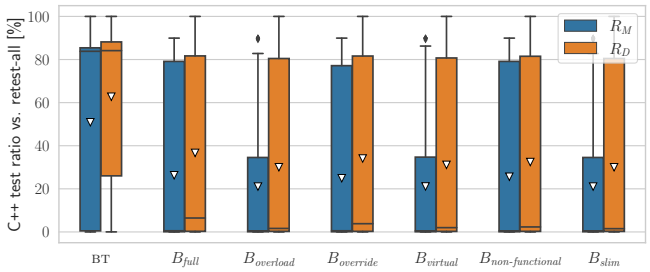


Fig. 4. Distribution of selected C++ test ratio of BINARYRTS configurations and BT compared to retest-all across all pull request runs

pull request CI pipelines only execute C++ tests selected by the static module-level RTS strategy of BT, BT still outputs a retest-all test report after execution. It can do so since it caches test results from unaffected test cases and can thereby report test verdicts from all executed plus cached tests. We compute the distribution plots for BT and all BINARYRTS configurations from these retest-all reports and the selected tests from BT and BINARYRTS, respectively. The results indicate that BT selects on average 51% (R_M) and 63% (R_D) of tests, whereas BINARYRTS selects on average 26% and 37% of tests with B_{full} configuration and 21% and 30% with B_{slim}. Moreover, the median selection ratios are 84% and 84% (BT), 0.5% and 6.5% (B_{full}), and 0.5% and 1.5% (B_{slim}).

In addition to the ratio of selected C++ tests, we compare the relative test duration for the different strategies against retest-all: the average relative test durations are 58% and 70% (BT), 32% and 44% (B_{full}), and 26% and 36% (B_{slim}).

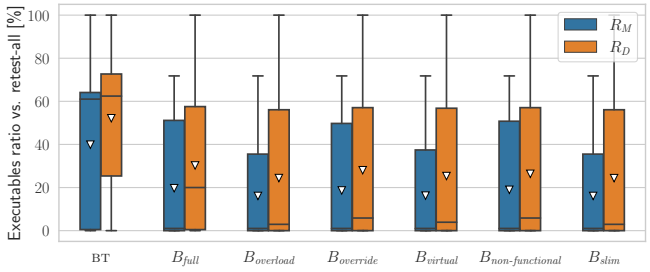


Fig. 5. Distribution of selected test executables ratio of BINARYRTS configurations and BT compared to retest-all across all pull request runs

The test execution costs are not only driven by the test duration, but also by the time required for process initialization and global (database) test setup for each test executable. Hence, if C++ test executables are skipped because no tests inside have been selected, end-to-end execution time for testing decreases. Therefore, we report the distribution of the ratio of selected test executables across all pull requests in Fig. 5.

Overall, we find that BINARYRTS saves considerably more testing effort than BT for both branches. Similar to our previous observations [19], the achieved savings for the maintenance branch (R_M) are higher than for the development branch (R_D). The difference between BINARYRTS configurations is small regarding the average test selection ratio ($\leq$7 pp), but becomes

more visible when looking at the median test selection ratio where B_{full} selects more than four times as many tests as B_{slim} for R_D. We discuss the impact on safety in RQ$_3$. BINARYRTS further selects on average only 20% (R_M) and 30% (R_D) of C++ test executables with B_{full}, whereas BT selects 40% and 52% test executables, respectively.

SUMMARY RQ$_1$. *We find that* BINARYRTS *selects on average 26% (R_M) and 37% (R_D) of all C++ tests with B_{full} on two release branches, whereas* BT *selects 51% and 63%, respectively. In 50% of the pull request runs on R_D, B_{full} selected more than four times as many test cases as B_{slim}.* BINARYRTS *further reduces the number of C++ test executables by 80% (R_M) and 70% (R_D).*

RQ$_2$: Java Testing Effort: In our prior study [19], we were able to save on average 42% of execution time for cross-language Java tests compared to retest-all using a file-level RTS strategy. However, in case of C++ changes, we encountered problems of imprecise test selection, since a test that uses a DLL is selected whenever changes are made to that DLL, regardless of whether the test covers the changed code or not.

To investigate the effectiveness of BINARYRTS for cross-language Java tests at IVU, we compare BINARYRTS against our file-level—or rather, DLL-level—RTS strategy. Similar to this pre-existing strategy, BINARYRTS selects Java tests at the level of JUnit test suites [19].

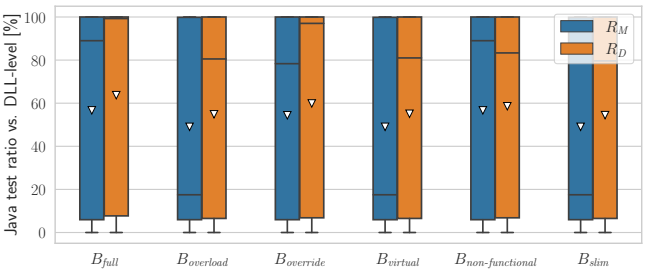


Fig. 6. Distribution of selected Java test ratio of BINARYRTS configurations compared to DLL-level RTS across all pull request runs

Fig. 6 depicts the test selection ratios for the different configurations of BINARYRTS. The results show that BINARYRTS with B_{full} selects on average 57% (R_M) and 64% (R_D) of the Java tests selected by the pre-existing DLL-level RTS strategy. The median selection ratios are higher at 89% and 99% (B_{full}), and 18% and 80% (B_{slim}).

In summary, we find that while BINARYRTS can be much more effective than DLL-level RTS for Java tests, DLL-level RTS selects almost the same number of tests in roughly 50% of pull request runs as B_{full} for R_D. We expect the reason to be that Java tests often use largely similar parts of the C++ code for their test setup and, therefore, are selected altogether for changes to core C++ components.

SUMMARY RQ$_2$. *We find that* BINARYRTS *(B_{full}) selects on average 57% (R_M) and 64% (R_D) of the Java tests selected by DLL-level RTS on two release branches. DLL-level RTS still performs comparatively well in roughly 50% of pull request runs for R_D.*

RQ$_3$: Safety: To assess the safety of BINARYRTS for different configurations, we compare the failed tests in a pull request run to the tests that BINARYRTS would have selected. If a test has failed in a pull request run and has not been selected by BINARYRTS, we need to check if this missed failure represents a real regression introduced in the pull request. Therefore, we check in the following order if the test (1) already failed on the target release branch, (2) failed due to infrastructure issues, (3) is a known flaky test, *i.e.,* a test with non-deterministic result, or (4) a newly detected flaky test which both, fails and passes within 100 reruns. We manually validate all missed failures, since especially test failures of categories (2) and (3) are sometimes difficult to automatically recognize from stack traces and often require discussion with IVU developers. If steps (1)–(3) do not provide a clear indication, we rely on rerunning tests (4) to detect flakiness, as is common practice in industry [57]–[59].

In total, we manually checked 179 pull request runs with missed C++ and Java test failures. The majority of missed failures either have already existed on the target branch or can be attributed to infrastructure issues, mainly database access problems. Most flaky tests we encounter are already known, but we still find a handful of new flaky tests that we reported to the responsible developers. Interestingly, we almost exclusively find flaky tests on R_D, indicating that flaky tests seem to get fixed eventually before a software version release.

Overall, we do not find any missed failures that are related to the changes introduced in the corresponding pull request for B_{full}. For B_{slim}, we also do not find any real missed failures. Despite these results, there are several sources for potential unsafe RTS behavior which we discuss in Sec. IV-C1.

SUMMARY RQ$_3$. *We find that* BINARYRTS *detects all real regressions in the considered pull requests with all configurations.*

C. Discussion

Next, we discuss weaknesses and possible improvements regarding safety, precision, and efficiency of BINARYRTS, and share feedback from IVU developers.

1) Safety: We expect BINARYRTS to be safe for changes to C++ functions if all run-time options are enabled (B_{full}), as BINARYRTS was designed following the example of safe function-level RTS techniques for C++ [29], [30]. In contrast with BT, which is only safe for changes to C++ files, BINARYRTS is further aware of changes to external files, thus ruling out a common source of unsafe RTS behavior [20], [56].

Yet, similar to existing techniques [19], [30], BINARYRTS can be unsafe if test traces are outdated. At IVU, we run the

tracing CI pipelines to update test traces for release branches in off-peak hours, but at least once per week. Another source for potential safety violations is that BINARYRTS ($B_{overload}$) only marks overloaded functions within the *same* file as affected by static dispatch (see Sec. III-B). Moreover, since BINARYRTS operates at run-time, expressions evaluated at compile-time, *e.g.,* macros or `constexpr`, are only considered if analysis for non-functional changes is enabled ($B_{non\text{-}functional}$). However, the text-based search to find usages of non-functional entities can become expensive for large C++ code bases if all parent directories of the changed source file are recursively searched. Therefore, BINARYRTS provides a parameter to control how many levels of parent directories to visit during the search. Safety violations can possibly occur if this parameter is set too low and thus functions making use of the changed non-functional entity are not marked as affected. Based on IVU conventions for the use of non-functional entities, we set the parameter to 2. Additionally, BINARYRTS allows defining a regular expression to match files that should trigger a retest-all strategy to anticipate context-specific safety challenges.

Due to these limitations and because prior research has revealed safety violations of supposedly safe RTS solutions [56], we conduct the safety trade-off experiments for RQ_3.

2) Precision: Since prior RTS research has studied more coarse-grained and less precise RTS at the level of files for Java and C# projects [11], [19], [20], [27], [38], we also investigate how our results from RQ_1 change if we aggregate our function-level test traces to file-level test traces. The results show that BINARYRTS with file-level analysis selected on average 41% (R_M) and 49% (R_D) of C++ tests. Hence, function-level gives better precision than file-level analysis.

We further see potential for improvement in how BINARYRTS deals with static and dynamic dispatch (see Sec. III-B). Currently, we resort to over-approximation approaches as we rely on the compiler-agnostic, yet simple analysis tool `ctags` to locate and parse functions. However, with more elaborate static analysis from C++ compilers, we could reduce the set of affected functions due to static and dynamic dispatch.

3) Efficiency: BINARYRTS is specifically designed to have low overhead inside CI pipelines of pull requests to compute the set of selected tests, commonly called the *analysis phase* of RTS systems [13], [29]. The time for selecting tests with B_{full}, the configuration with maximum overhead, was on average roughly 30 seconds, where reading and deserializing test traces from disk took most of the time. This could be further improved by using a database to minimize involved I/O.

For the so-called *collection phase*, *i.e.,* when collecting per-test execution traces in dedicated CI pipelines, we observe relatively high instrumentation overhead of roughly a factor of 2–3. While allowing compiler-agnostic code and system call instrumentation, the overhead introduced by dynamic binary instrumentation is generally expected to be higher than for static source code instrumentation (often several times slower than the original program) [45], [60], [61]. Other instrumentation tools for C or C++, such as `OpenCppCoverage` or `CodeCoverage.exe` by Microsoft, exhibit similarly high overhead [62]. Moreover, DynamoRIO has been shown to have significant performance impact in other contexts as well [47], [63]. It also lacks support for efficiently instrumenting dynamically generated code [64]. Therefore, we need to disable the JVM's just-in-time compiler when tracing Java tests which further increases instrumentation overhead. However, since the CI pipelines that collect the per-test execution traces are executed *offline* [13], meaning in off-peak hours independently of any pull requests, the instrumentation overhead does not impact the development process. To further increase the tracing frequency, we envision improvements related to using more lightweight binary instrumentation solutions (BINARYRTS has experimental support for the DBA tool `Frida` [65]) or implementing source code instrumentation through source-to-source transformation using `clang`. If the latter is properly implemented, compiling and linking the transformed source code would still be possible with any compiler and linker.

4) Developer Feedback: We have continuously discussed the design of BINARYRTS and its evaluation with IVU engineers to establish broad support among developers and testers. As they see great value in the proposed RTS solution, we are currently integrating BINARYRTS into all release branches. Moreover, developers have suggested to implement further developer-aiding tools for test coverage visualization and test gap analysis based on the test traces collected with BINARYRTS.

D. Threats to Validity

1) External Validity: BINARYRTS has been designed to suit the context-specific challenges at IVU and, therefore, our findings do not necessarily generalize to other software projects inside and outside of IVU. Moreover, even though BINARYRTS also supports Linux and multiple platforms, our evaluation at IVU was performed on C++ software built with Microsoft's compiler toolchain to x86-64 binaries on Windows. Nevertheless, our results confirm prior RTS research on C++ software that reported significant savings in testing effort [29], [30]. We publish the source code of BINARYRTS to ease transferability to other projects beyond the context of IVU.

Another threat to validity emerges from the fact that, similar to previous studies [14], [19], [55], we use test durations from test reports to measure test execution time. GoogleTest measures test durations only in millisecond resolution, which may distort results as some fast-running unit tests sometimes take less than one millisecond to execute. To address this threat, we also report the ratio of selected tests and test executables. The latter provides an indication regarding savings in (global) test setup costs, which can be substantial at IVU.

2) Internal Validity: Internal threats stem from the implementation of BINARYRTS, mainly related to using DynamoRIO for analyzing and instrumenting C++ binaries, and `ctags` for parsing C++ source files. To address these threats, we manually validated selection results with IVU engineers and wrote automated unit and integration tests for BINARYRTS.

V. RELATED WORK

We have referenced several RTS techniques throughout this paper that have motivated and partly inspired BINARYRTS (see

Sec. II-C). Below, we list studies targeting C or C++ software that we consider to be most relevant for this work.

Early related RTS research was primarily on C software. In 1994, Chen *et al.* [22] presented TESTTUBE, an RTS technique for C programs. Similar to BINARYRTS, it tracks covered functions and non-functional entities per test case. TESTTUBE employs source code instrumentation and static analysis, and selects a test if any of its covered entities has changed. Rothermel and Harrold [8], [40] proposed DEJAVU, an RTS technique for C which uses static control flow graphs and edge-level test traces obtained through source code instrumentation to compute affected tests. Using this fine-grained analysis, DEJAVU is safe for C code modifications [66], [67].

Later, Rothermel *et al.* [9] extended DEJAVU to object-oriented C++ software. Therefore, they combine interprocedural and class control flow graphs with edge-level test traces. Their proposed RTS technique also accounts for dynamic dispatch and polymorphism, but, due to the lack of adequate C++ analysis and instrumentation tools at the time, it was not actually implemented. In 1995, Kung *et al.* [23] presented an RTS technique for C++ based on static class dependency graphs. Using these graphs they compute a *class firewall* (see also Leung *et al.* [68]), that is the set of classes affected by changes, and derive which tests need to be selected to retest affected classes. Jang *et al.* [24] and White *et al.* [25] extended the class firewall approach for C++ software to improve precision and safety by adding fine-grained change impact analysis and data flow analysis, respectively. The class firewall approach has also inspired the development of DEJAVOO [10] and STARTS [12], [28], two class-level static RTS techniques for Java.

In the past decade, only two studies on RTS in C++ software were published, which we deem as most related to this work: Fu *et al.* [29] presented RTS++, a static RTS technique operating on function call graphs. RTS++ targets modern C++ programs that compile to LLVM bitcode and use the GoogleTest testing framework. Fu *et al.* evaluated RTS++ on 11 open-source projects and find that the number of selected tests is on average reduced by 61% compared to retest-all. RTS++ is not applicable at IVU, as it requires all libraries to be statically linked into a single executable binary.

To address integration and system testing in large-scale C++ web services at Google, Zhong *et al.* [30] developed TESTSAGE, a dynamic RTS technique for distributed systems. When deploying TESTSAGE to Google testing infrastructure, they achieved up to 50% reduction in testing time. TESTSAGE also targets LLVM-based projects, as it relies on a customized version of XRAY, a function instrumentation tool for LLVM. TESTSAGE further uses Google's internal version control system and code analysis tool PIPER to perform change impact analysis for test selection. In contrast, BINARYRTS is based on publicly available tools and frameworks, and, due to the employed binary instrumentation, works with different compilers, operating systems, and binary formats.

In two previous studies, we have studied unsafe and safe RTS techniques to implement RTS for the multi-language code base at IVU [19], [55]. Unsafe RTS is typically language-agnostic, as it relies only on readily available CI and version control system (VCS) metadata [55]. We found that the best performing unsafe RTS technique saved on average 19.8% of testing time while 93.4% of failures were still detected on the main development branch. Since unsafe RTS is not suitable for pull requests to *release* branches, we developed a safer RTS approach for Java [19]. This new approach is akin to EKSTAZI [11], [27] and RTSLINUX [20], two file-level RTS techniques for Java, and tracks opened files for each test through system call analysis. Similar to BINARYRTS, this makes it safe for changes to external files, such as configuration files, as well as source files in other programming languages, such as SQL or XML. Furthermore, the approach is build system aware, meaning it accounts for changes to the build system configuration, and selectively builds only those Java modules required for testing; this resulted in a an end-to-end CI pipeline time reduction for Java by 50%–63% on average. However, we also found that the file-level analysis granularity was too coarse grained when Java tests accessed DLLs, as it results in most tests being selected upon any C++ changes. BINARYRTS is built on top of these insights and provides a practical RTS solution for tests that use C++ binaries either directly (C++ tests) or through cross-language links (Java tests).

To summarize, no prior RTS research analyzes regression tests which use arbitrary C++ binaries, accounts for multi-language source files and non-code artifacts, or performs dynamic binary instrumentation for RTS. We are the first to evaluate C++ RTS for pull requests in industry-scale CI.

VI. CONCLUSION

In this paper, we present BINARYRTS, a dynamic RTS technique for reliably selecting tests that use C++ binaries during execution. It harnesses dynamic binary instrumentation to monitor covered functions and accessed files for each test at run-time. This way, BINARYRTS is also aware of cross-language links to source files in other programming languages and non-code artifacts used during testing. We evaluate BINARYRTS in IVU's large-scale CI infrastructure on roughly 16,000 C++ and Java tests, some of which cover code from hundreds of C++ binaries. Our results indicate that BINARYRTS excludes on average 63%–74% of C++ tests and 36%–43% of Java tests, thereby reducing test duration by on average up to 68% against a naive retest-all baseline. The improved testing time directly translates to faster feedback in CI testing, boosting developer efficiency and satisfaction, which is why IVU is currently deploying BINARYRTS to all release branches. To foster RTS research on languages other than Java, we publish BINARYRTS and its source code as the first publicly available RTS tool for C++ software.

ACKNOWLEDGMENTS

We thank Dennis Bracklow, René Dammer, Stefan Golas, Maximilian Pohl, and Stefan Sieber for their support. This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant Q-Soft 01IS22001B. The responsibility for this article lies with the authors.

REFERENCES

[1] H. K. Leung and L. White, "Insights into regression testing," in *Proceedings of the International Conference on Software Maintenance*, 1989, pp. 60–69.

[2] 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*, 2014, pp. 235–245.

[3] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagppan, "Fastlane: Test minimization for rapidly deployed large-scale online services," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 408–418.

[4] 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*, 2019, pp. 91–100.

[5] K. Fischer, F. Raji, and A. Chruscicki, "A methodology for retesting modified software," in *Proceedings of the National Telecommunications Conference*, 1981, pp. 1–6.

[6] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing Verification and Reliability*, vol. 22, pp. 67–120, 2012.

[7] K. F. Fischer, "A test case selection method for the validation of software maintenance modifications," in *Proceedings of International Computer Software and Applications Conference*, 1977, pp. 421–426.

[8] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 173–210, 1997.

[9] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression test selection for c++ software," *Software Testing, Verification and Reliability*, vol. 10, pp. 77–109, 2000.

[10] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2004, pp. 241–251.

[11] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *Proceedings of the International Conference on Software Engineering*, 2015, pp. 713–716.

[12] O. Legunsen, A. Shi, and D. Marinov, "Starts: Static regression test selection," in *Proceedings of the International Conference on Automated Software Engineering*, 2017, pp. 949–954.

[13] L. Zhang, "Hybrid regression test selection," in *Proceedings of the International Conference on Software Engineering*, 2018, pp. 199–209.

[14] A. Shi, P. Zhao, and D. Marinov, "Understanding and improving regression test selection in continuous integration," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2019, pp. 228–238.

[15] E. Knauss, M. Staron, W. Meding, O. Soder, A. Nilsson, and M. Castell, "Supporting continuous integration by code-churn based test selection," in *Proceedings of the International Workshop on Rapid Continuous Software Engineering*, 2015, pp. 19–25.

[16] B. Busjaeger and T. Xie, "Learning for test prioritization: An industrial case study," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2016, pp. 975–980.

[17] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2017, pp. 12–22.

[18] G. Rothermel and M. J. Harrold, "A framework for evaluating regression test selection techniques," in *Proceedings of the International Conference on Software Engineering*, 1994, pp. 201–210.

[19] D. Elsner, R. Wuersching, M. Schnappinger, A. Pretschner, M. Graber, R. Dammer, and S. Reimer, "Build system aware multi-language regression test selection in continuous integration," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 87–96.

[20] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression test selection across jvm boundaries," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017, pp. 809–820.

[21] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, pp. 1–34, 2022.

[22] Y. F. Chen, D. S. Rosenblum, and K. phong Vo, "Test tube: a system for selective regression testing," in *Proceedings of the International Conference on Software Engineering*, 1994, pp. 211–220.

[23] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," *Journal of Object-Oriented Programming*, vol. 8, pp. 51–65, 1995.

[24] Y. K. Jang, M. Munro, and Y. R. Kwon, "An improved method of selecting regression tests for c++ programs," *Journal of Software Maintenance and Evolution*, vol. 13, pp. 331–350, 2001.

[25] L. White, K. Jaber, B. Robinson, and V. Rajlich, "Extended firewall for regression testing: An experience report," *Journal of Software Maintenance and Evolution*, vol. 20, pp. 419–433, 2008.

[26] L. Zhang, M. Kim, and S. Khurshid, "Faulttracer: A spectrum-based approach to localizing failure-inducing program edits," *Journal of Software: Evolution and Process*, vol. 25, pp. 1357–1383, 2013.

[27] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[28] 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*, 2016, pp. 583–594.

[29] B. Fu, S. Misailovic, and M. Gligoric, "Resurgence of regression test selection for c++," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2019, pp. 323–334.

[30] 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*, 2019, pp. 430–440.

[31] LLVM, "Llvm compiler infrastructure." [Online]. Available: https://llvm.org/

[32] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*, 2018, pp. 1–23.

[33] A. Maven, "Maven." [Online]. Available: https://maven.apache.org

[34] Microsoft, "Msvc c++ toolset." [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/projects-and-build-systems-cpp

[35] Google, "Googletest." [Online]. Available: https://google.github.io/googletest/

[36] JUnit, "Junit 5," 2021. [Online]. Available: https://junit.org/junit5

[37] M. J. Harrold, A. Orso, J. A. Jones, T. Li, M. Pennings, S. Sinha, A. Gujarathi, D. Liang, and S. A. Spoon, "Regression test selection for java software," *ACM SIGPLAN Notices*, vol. 36, pp. 312–326, 2001.

[38] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. module-level regression test selection for .net," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017, pp. 848–853.

[39] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka, "Optimizing test placement for module-level regression testing," in *Proceedings of the International Conference on Software Engineering*, 2017, pp. 689–699.

[40] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *Proceedings of the International Conference on Software Maintenance*. IEEE, 1993, pp. 358–367.

[41] ——, "Selecting regression tests for object-oriented software," in *Proceedings of the International Conference on Software Maintenance*, 1994, pp. 14–25.

[42] Clang, "Clang compiler." [Online]. Available: https://clang.llvm.org/

[43] LLVM, "Llvm xray function call tracing." [Online]. Available: https://llvm.org/docs/XRay.htm

[44] D. Elsner, R. Wuersching, M. Schnappinger, and A. Pretschner, "Probe-based syscall tracing for efficient and practical file-level test traces," in *Proceedings of the International Conference on Automation of Software Test*, 2022, pp. 126–137.

[45] N. Nethercote, "Dynamic binary analysis and instrumentation or building tools is easy," Ph.D. dissertation, University of Cambridge, 11 2004.

[46] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, MIT, 9 2004.

[47] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the*

Conference on Programming Language Design and Implementation, 2005, pp. 190–200.
[48] DynamoRIO, "Dynamorio." [Online]. Available: https://dynamorio.org
[49] ctags, "Universal ctags." [Online]. Available: https://ctags.io
[50] F. I. Vokolos and P. G. Frankl, *Pythia: A regression test selection tool based on textual differencing*. Springer, 1997.
[51] git, "git." [Online]. Available: https://git-scm.com
[52] "Java agent api," 2017. [Online]. Available: https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/package-summary.html
[53] J. Bell and G. Kaiser, "Unit test virtualization with vmvm," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 550–561.
[54] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric, "Debugging the performance of maven's test isolation: Experience report," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2020, pp. 249–259.
[55] 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*, 2021, pp. 491–504.
[56] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, "A framework for checking regression test selection tools," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 430–441.
[57] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," *Proceedings of the International Conference on Software Engineering*, pp. 433–444, 2018.
[58] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developers perspective," in *Proceedings of the ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, Inc, 8 2019, pp. 830–840.
[59] 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*, 2020, pp. 1471–1482. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-study-on-the-lifecycle-of-flaky-tests/
[60] V. J. M. Mans, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.
[61] A. Engelke and M. Schulz, "Instrew: Leveraging llvm for high performance dynamic binary instrumentation," in *Proceedings of the International Conference on Virtual Execution Environments*, 2020, pp. 172–184.
[62] CQSE, "Performance impact of c++ profilers." [Online]. Available: https://docs.teamscale.com/howto/setting-up-profiler-tga/cpp/#performance-impact
[63] M. A. B. Khadra, D. Stoffel, and W. Kunz, "Efficient binary-level coverage analysis," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1153–1164.
[64] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, "Optimizing binary translation of dynamically generated code," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2015, pp. 68–78.
[65] Frida, "Frida." [Online]. Available: https://frida.re/
[66] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, pp. 529–551, 1996.
[67] G. Rothermel, "Efficient, effective regression testing using safe test selection techniques," Ph.D. dissertation, Clemson University, 5 1996.
[68] H. K. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Press, 1990, pp. 290–301.

A.3.3. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration

 Daniel Elsner, Florian Hauer, Alexander Pretschner, Silke Reimer, Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration, Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 491–504, July 2021.

In the following, the complete paper is included in its published form in accordance with the ACM author rights, DOI: 10.1145/3460319.3464834.

Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration

Daniel Elsner
Florian Hauer
Alexander Pretschner
daniel.elsner@tum.de
florian.hauer@tum.de
alexander.pretschner@tum.de
Technical University of Munich
Munich, Germany

Silke Reimer
sre@ivu.de
IVU Traffic Technologies
Berlin, Germany

ABSTRACT

Regression test selection (RTS) and prioritization (RTP) techniques aim to reduce testing efforts and developer feedback time after a change to the code base. Using various information sources, including test traces, build dependencies, version control data, and test histories, they have been shown to be effective. However, not all of these sources are guaranteed to be available and accessible for arbitrary continuous integration (CI) environments. In contrast, metadata from version control systems (VCSs) and CI systems are readily available and inexpensive. Yet, corresponding RTP and RTS techniques are scattered across research and often only evaluated on synthetic faults or in a specific industrial context. It is cumbersome for practitioners to identify insights that apply to their context, let alone to calibrate associated parameters for maximum cost-effectiveness. This paper consolidates existing work on RTP and unsafe RTS into an actionable methodology to build and evaluate such approaches that exclusively rely on CI and VCS metadata. To investigate how these approaches from prior research compare in heterogeneous settings, we apply the methodology in a large-scale empirical study on a set of 23 projects covering 37,000 CI logs and 76,000 VCS commits. We find that these approaches significantly outperform established RTP baselines and, while still triggering 90% of the failures, we show that practitioners can expect to save on average 84% of test execution time for unsafe RTS. We also find that it can be beneficial to limit training data, features from test history work better than change-based features, and, somewhat surprisingly, simple and well-known heuristics often outperform complex machine-learned models.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464834

KEYWORDS

software testing, regression test optimization, machine learning

ACM Reference Format:
Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464834

1 INTRODUCTION

Regression test selection (RTS) aims at identifying tests that are affected by a change to the code base, when executing every test in the test suite is prohibitively expensive [25, 76]. Effective traditional RTS techniques *safely* exclude those tests that cannot fail by relying on language-specific white-box program analyses, e.g., recording test-specific execution traces through code instrumentation [26, 42, 58, 65, 66, 70, 79]. However, they are often too costly in large-scale code bases with rapid continuous integration (CI) testing [21, 45, 62], not capable of collecting test dependencies across language boundaries in multi-language software [12, 45, 55], and cannot trace third-party libraries [41]. Regression test prioritization (RTP) aims to detect faults earlier by reordering tests through "surrogates" [76]. However, traditional RTP techniques that rely on code coverage surrogates suffer similar limitations [21, 71].

To address these limitations specifically in CI environments, researchers have proposed numerous lightweight, less intrusive RTP and *unsafe* RTS techniques: They use different surrogates and machine learning (ML) models to rank tests by their likelihood to fail and, in the case of RTS, select a subset based on some cut-off criterion [8, 11, 45, 60, 71]. The underlying *ranking models* exploit different information sources. These include CI test execution logs [3, 11, 21, 71], version control system (VCS) metadata (e.g., number of changed files in commit) [37, 62], (textual) differences in code churn [8, 49, 60, 67], and project- or organization-specific information such as static build dependencies [45], flaky test detection signals [45, 60, 62], or a black-box model of the program inputs [29]. Arguably, access to the latter types of information cannot be guaranteed for arbitrary CI environments. Since CI and VCS metadata are automatically generated throughout the development process, they are generally available and inexpensive. Unsafe RTS

and RTP techniques that solely use this information are language-agnostic, easy to transfer and to evaluate, and do not require program or code access. Methodologically, they both rely on ranking tests [8, 11]. Hence, in the following, we collectively refer to techniques as CI-RTP/S, if their ranking models *only* use CI and VCS metadata.

Even though the effectiveness of CI-RTP/S is demonstrated in various studies, they have the following limitations: First, while studies show the effectiveness on single projects [3, 16, 21, 38, 43, 68] and variations across projects [37, 44, 71, 81], we find *sensitivity* to be another important attribute after consulting with our industry partner IVU Traffic Technologies[1]: How sensitive is the cost-effectiveness of CI-RTP/S to size, timeliness, and variety of data they were calibrated on? Or, since calibration can be challenging, how much cost-effectiveness is sacrificed when using a ranking model that is only semi-optimally calibrated? Existing studies that discuss related issues were carried out in a specific industrial context and it is thus unclear, if the measured sensitivity carries over to other industrial and open-source projects. Second, empirical results of CI-RTP/S cost-effectiveness are often obtained on datasets with seeded faults instead of real-world failures [8, 13] or drawn from inaccessible industrial contexts which impedes reproducibility [11, 45, 62]. A recently published dataset, RTPTorrent [50, 51], closely resembles real-world development activity. Though it has not yet been used in related studies, this dataset bears the potential to improve transparency and thus transferability of results from CI-RTP/S studies. Last, in pursuit of ever more effective CI-RTP/S, aspects of the design and evaluation of techniques are scattered across existing work. Overall, findings in the literature are neither unequivocal, nor directly comparable. It is a cumbersome task for practitioners to identify insights that apply to their context, let alone to calibrate associated parameters for maximum cost-effectiveness.

Addressing these limitations, we consolidate existing ideas from prior research into a methodology to build and evaluate approaches for RTP and unsafe RTS that exclusively rely on CI and VCS metadata. We identify an *approach* by three parameters, namely (i) *how much* (i.e., amount of training data) of (ii) *which information* (i.e., choice of features) is used to (iii) *rank tests* (i.e., choice of ranking model). Our methodology does not propose new techniques, but provides a clear, generic process that first guides practitioners from exploiting their readily available CI and VCS metadata to building candidate CI-RTP/S approaches (from prior research) for their project. Then, the subsequent comparative evaluation of candidates yields not only the most cost-effective approaches, but also gives insights on how an approach's performance changes if any of the three parameters is varied, i.e., on its cost-effectiveness sensitivity. From a practitioners view, this methodology enables simple adoption of RTP and unsafe RTS without having to manually investigate existing techniques and their applicability. We thus address the practical questions of which data to gather, how to perform feature engineering and predictive modeling, how to comparatively evaluate different candidate approaches, and how to choose the project-specific best approach.

To estimate performance trade-offs, we apply our methodology in a large-scale empirical study on real test failures from CI and VCS histories of 23 industrial and open-source projects. We then conduct rigorous statistical analyses, yielding guidelines on which candidate approaches are, empirically, the most promising ones.

In summary, our **contributions** are as follows:

- **Methodology.** We consolidate existing research into an actionable methodology to build and evaluate approaches for RTP and unsafe RTS exclusively using CI and VCS metadata.
- **Empirical Study.** First, we analyze the sensitivity of cost-effectiveness to the parameters (i) training data amount, (ii) choice of features, and (iii) choice of ranking model. Second, we estimate performance trade-offs in RTP and unsafe RTS: Cost-effectiveness fluctuates across projects underlining the need for project-specific assessment using our methodology. In the studied projects, approaches chosen by our methodology help to save on average 84% of test execution time while detecting 90% of the failures for unsafe RTS and significantly outperform established RTP baselines.
- **Guidelines.** (1) It can be beneficial to limit training data, (2) features on test history work particularly well compared to change-based features, and (3) inexpensive simple heuristics of the kind "skip test if not failed in the last ten runs" often outperform complex ML models from prior work.
- **Dataset.** To foster comparable and evidence-based studies on CI-RTP/S, we publish our dataset consisting of 23 heterogeneous software projects from industry (3) and open-source development (20)[2]. It covers more than 37,000 CI test logs with real failures and 76,000 VCS commits. The open-source projects are drawn from the recently published RTPTorrent dataset [50, 51] which embodies CI test logs that we further enrich and process for CI-RTP/S.

2 METHODOLOGY

We have motivated our goal to build approaches useful for both, unsafe RTS and RTP, that solely rely on CI and VCS metadata (CI-RTP/S). Before providing details about the methodology, we describe the problem more formally and introduce notations used throughout this paper: Let P be a program, Δ be a modification introduced to P to create P', and $\mathcal{T}$ be a test suite. For each test $T \in \mathcal{T}$, the *ranking model*, M, first predicts T's failure score by using a *set of features*, F. Second, these scores are used to rank tests in $\mathcal{T}$, yielding an intermediary $\mathcal{T}^*$, i.e., a test order as aimed at by RTP. Then, based on a *cut-off criterion* only a subset $\mathcal{T}' \subseteq \mathcal{T}^*$ is selected as part of RTS. Depending on the desired strategy (RTP or RTS), $\mathcal{T}^*$ or $\mathcal{T}'$ can be used to test P' [11, 18, 45, 65].

We identify four consecutive process steps when creating and evaluating CI-RTP/S approaches which are shown in Fig. 1: Exploiting available data sources, engineering features from the collected data, building predictive ranking models, and evaluating CI-RTP/S. Since each of these steps involves methodological subgoals, we address them in the following subsections. Notably, this schematic process is inspired by the Cross-Industry Standard Process for Data Mining (CRISP-DM) [73].

[1] *IVU Traffic Technologies* is one of the world's leading providers of public transport software solutions: https://www.ivu.com/

[2] Dataset, source code, and detailed evaluation results are part of the supplemental material available at [22].

Figure 1: Schematic process of the methodology to build and evaluate CI-RTP/S approaches.

2.1 Data Source Exploitation

Goal: Exploit *generally available* data sources to collect raw data useful for failure prediction.

Modern software projects are developed in code repositories that use some sort of VCS. While there are various flavors of VCS, such as distributed (e.g., Git) or centralized (e.g., Apache Subversion), they share the notion of a *commit*, i.e., a code revision made by a single author. It will contain at least the following information: Identifier, author, commit timestamp, commit message, and a changeset. The changeset in turn includes all the added, modified, or deleted files.

Meanwhile, regression testing is typically performed in CI environments. CI tools, such as Jenkins or Travis CI, allow users to configure CI pipelines, which are regularly executed. We refer to a CI run at timestamp t as R_t. Most pipelines contain multiple stages, including a *build stage*, where artifacts necessary to run the tests are generated (e.g., compilation), a *test stage*, where (regression) tests are executed, and a *deploy stage*, which comprises of scripts to publish and deploy tested artifacts. After the test stage in R_t, a *test log* (also called test report) is typically generated from which the following information can be extracted per test $T_{t,i}$ in the executed test suite $\mathcal{T}_t = \{T_{t,1}, \ldots, T_{t,n}\}$: Test identifier (e.g., test class name[3]), result (e.g., passed or failed), and duration. It is thus a requirement to derive test result and duration information from the CI environment. However, most testing frameworks and CI systems (or plugins) already provide structured test logs in several output formats, but they can also be parsed from raw textual CI logs, e.g., by using regular expressions [50]. Depending on the configuration, a CI run R_t may be triggered to start either after each commit, after the previous run R_{t-1} has finished, or whenever required (hardware) resources are available. The set of introduced commits between two CI runs, R_t and R_{t-1}, Δ_t, will at least contain one commit, as it would be pointless to trigger a CI run without any modification except to detect flaky tests (see Sec. 3.6.1).

Fig. 2 shows how the outlined entities *commit* and *test log* are part of the software development process. The union of all available Rs with their $\mathcal{T}$ and Δ constitutes the input for the feature engineering process described in the next section.

2.2 Feature Engineering

Goal: Craft features for failure prediction from collected raw data that capture specific *defect* hypotheses.

For a given test T, ranking models use *features* to predict a failure score. These features are numerical representations of characteristics of T. For example, considering T's failure behavior, if T failed ten times in previous test runs, the feature *failure count* will have the value 10. A *good* feature is one that improves the model's predictive performance, hence, one that captures a valid *defect* hypothesis.

[3]We follow prior RTS research by analyzing tests at class (or file) rather than module or method granularity level [27, 70, 78].

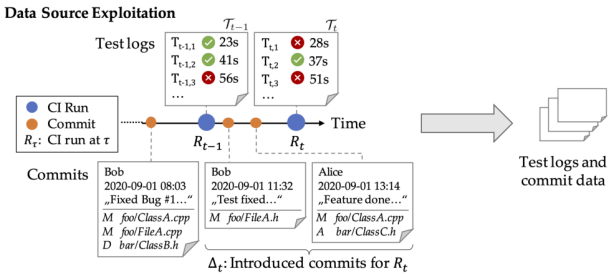


Figure 2: Exploiting VCS history and CI test logs used as input for feature engineering.

Note that we use *defect* as an umbrella term for *failures*, *faults*, or *errors*.

Feature engineering is concerned with deriving these defect hypotheses and computing respective features through a feature function ϕ. If applied to the collected raw data, i.e., to each test in all $\mathcal{T}$s with their respective Δ, we can construct a dataset, D, suitable for building and evaluating ranking models. More formally, for a given test $T_{t,i}$ in the test suite $\mathcal{T}_t$ available for R_t, we can calculate a vector of m features, $x_{t,i} = (x_{t,i,1}, \ldots, x_{t,i,m})$, from raw data using the feature function $\phi(T_{t,i}, \{\mathcal{T}_1, \ldots, \mathcal{T}_{t-1}\}, \{\Delta_1, \ldots, \Delta_t\})$. Note that this captures reality, where at timestamp t before regression testing, there are only historical test logs and commits available as well as the newly introduced set of commits Δ_t. To create D, this is done for every test suite $\mathcal{T}$ of the collected CI runs.

Hereafter, we describe 16 features for CI-RTP/S from existing work. For each feature, we state the underlying defect hypothesis, briefly explain how it is computed from outlined raw CI test logs and VCS commits, and reference prior research which already used it. Aligned with prior work, we semantically group them into four feature sets, $F = \{F_1, \ldots, F_4\}$, to increase comprehensiveness [4, 62]. The k-th feature in feature set j is denoted by $f_{j,k}$.

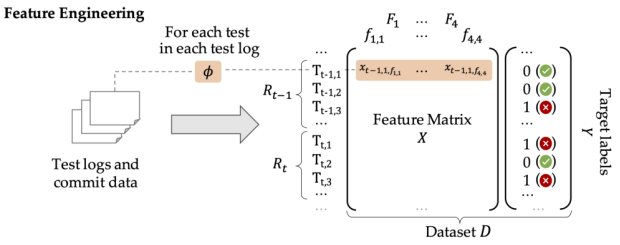


Figure 3: Structure of a dataset D for CI-RTP/S. Rows are chronologically ordered, i.e., from test executions of the earliest to the most recent CI run.

Fig. 3 illustrates how we derive D by computing F for all test executions in the test logs, e.g., test $T_{t,1}$ in CI run R_t, and labeling them with their respective test results as the ground truth, Y, e.g., 1 for *failed*. Based on the total amount of collected CI runs with test logs, $r = |R|$, D will have N rows, with $N = |\bigcup_{t=1}^{r} \mathcal{T}_t|$, and M+1 columns, with $M = |\bigcup_{j=1}^{4} F_j|$, including one column for $Y \in \{0, 1\}^{N \times 1}$. The

feature matrix, denoted by $X \in \mathbb{R}^{N \times M}$, stacks all computed feature values for each test execution $T_{t,i}$, $x_{t,i} = (x_{t,i,1}, \ldots, x_{t,i,M})$.

Test History Features (F_1): This feature set contains features computed only from CI test logs, i.e., for a test T in $\mathcal{T}_t$ we consider its individual execution history from $\{R_1, \ldots, R_{t-1}\}$.

The first hypothesis is: Tests that previously failed will test error-prone code and are therefore likely to fail again [28, 74, 76].

- Failure count ($f_{1,1}$) [4, 45, 54, 56, 57, 60]: Total number of times T failed.
- Last failure ($f_{1,2}$) [4, 21, 34, 47]: Amount of CI runs since last failure, i.e., for R_t, if T last failed in R_τ, the value of $f_{1,2}$ will be $t - \tau$.

A *test transition* occurs if a test changes its test result between two CI runs. For instance, if a test failed in R_{t-1} and passed in the subsequent run R_t (or vice versa), the test *transitioned* in R_t.

The second hypothesis is: Tests that previously transitioned will test critical code and are therefore likely to fail.

- Transition count ($f_{1,3}$) [43]: Total number of times T transitioned.
- Last transition ($f_{1,4}$) [43]: Amount of CI runs since last transition, analogous to the feature $f_{1,2}$.

The third hypothesis is: Tests with high execution duration involve complex, time-consuming tasks (e.g., networking, file access) which are prone to fail (e.g., timeout) [62]. Additionally, on the one hand, longer running tests might cover larger parts of the program which simultaneously increases the chance of covering a fault. On the other hand, fast-running tests might reveal faults more quickly, making test execution time a potentially useful feature [13, 60].

- Average test duration ($f_{1,5}$) [13, 60]: Average test execution duration of T across previous CI runs.

(Test, File)-History Features (F_2): The idea of this feature set is to identify associations between tests and files, often referred to as *test-to-code* traceability links [72]. Therefore, we record historical co-occurrences of test failures or transitions with changed files in a contingency table [37]. For example re-consider Fig. 2, if test $T_{t,3}$ failed in R_t where the *combined changeset* {*foo/FileA.h, foo/ClassA.cpp, bar/ClassC.h*} was introduced, we increment the co-occurrence frequencies of ($T_{t,3}$, *foo/FileA.h*), ($T_{t,3}$, *foo/ClassA.cpp*), and ($T_{t,3}$, *bar/ClassC.h*). These co-occurrences are referred to as *(test, file)-failures*, the same applies to *(test, file)-transitions*.

The first hypothesis is: Files that were in the changeset when a test *failed*, are related to this test's outcome. More precisely, if certain files are in the changeset of a commit, they might imply higher failure likelihood for specific tests.

- Maximum (test, file)-failure frequency ($f_{2,1}$) [3, 37, 54, 62, 68]: Given a test T in R_t, for each file in the combined changeset, we first obtain the total (T, file)-failure frequency from $\{\mathcal{T}_1, \ldots, \mathcal{T}_{t-1}\}$. Then, we determine the maximum across all files, as the combined changeset is as risky for T as its riskiest file.
- Maximum (test, file)-failure frequency (relative) ($f_{2,2}$) [3, 37, 54, 62, 68]: The relative frequency is calculated by dividing the (T, file)-failure frequencies by the number of times T has failed so far. For example, if test T failed a hundred times before R_t, and a file was part of the combined changeset half of the times, the relative frequency will be 0.5. Again, we determine the maximum across all files. This feature allows to discriminate between systematic and arbitrary co-occurrence of a changed file and a test failure.

The second hypothesis is: Files that were in the changeset when a test *transitioned*, are related to this test's outcome.

- Maximum (test, file)-transition frequency ($f_{2,3}$): Given a test T in R_t, we take the maximum (T, file)-transition frequency across all files in the combined changeset.
- Maximum (test, file)-transition frequency (relative) ($f_{2,4}$): We take the maximum relative (T, file)-transition frequency across all files in the combined changeset.

Even though we are not aware of prior work that proposes features $f_{2,3-4}$, they build on the existing understanding of test transitions [43] and failure-based features $f_{2,1-2}$.

(Test, File)-Similarity Features (F_3): These features embody lexical similarities between *names* and *paths* of a test and files in the changeset. This similarity proxies human perceived affiliation between a file and a test [45]. The hypothesis is: Conventions lead to tests and tested files with similar names and paths [72].

- Minimum file path distance ($f_{3,1}$) [11, 72]: We use the Levenshtein distance as proposed by White et al. [72] for test-to-code traceability links. Then, we determine the minimum distance, i.e., maximum similarity, across all files in the combined changeset.
- Maximum file path token similarity ($f_{3,2}$) [43, 45]: Based on the intuition of shared directories [43], we split file paths into tokens and count common tokens among test and file path. We determine the maximum similarity across all files in the combined changeset.
- Minimum file name distance ($f_{3,3}$) [72]: Similar to $f_{3,1}$, we use the minimum Levenshtein distance between a test name and each file name in the combined changeset.

Change Features (F_4): While the features described so far directly concern predicting the outcome of a *specific* test in a CI run R_t, there are also features that express how the introduced commits (i.e., *changes*), Δ_t, affect the failure likelihood level of *all* tests.

The first hypothesis is: Changes involving a larger number of distinct authors are more likely to cause failures [43, 52].

- Distinct authors ($f_{4,1}$) [43, 45]: Number of distinct authors within Δ_t, i.e., across all commits.

The second hypothesis is: Large changes are more difficult to review and therefore more error-prone [45].

- Changeset cardinality ($f_{4,2}$) [4, 45, 62]: Number of files in the combined changeset.
- Amount of commits ($f_{4,3}$) [4, 34]: Amount of commits in Δ_t, i.e., since last CI run.

The third hypothesis is: Certain file types are more likely to cause failures than others. As we want to provide a general methodology, we rather consider the variety of file extensions in a change, than specific file types [45, 62].

- Distinct file extensions ($f_{4,4}$) [45, 62]: Number of distinct file extensions in the combined changeset.

2.3 Predictive Modeling

Goal: Select and build effective and efficient ranking models.
In the following, we explain how we use a constructed dataset D to build ranking models for CI-RTP/S. Similar to related work, we target *point-wise* ranking models [11, 45]: Given a vector of feature values, i.e., one row in X, $x_{t,i}$, the model outputs a *score*, $\hat{y}$, between 0 and 1. $\hat{y}$ can be interpreted as a test's *estimated likelihood to fail* [45] and should be close to 1 for tests which are likely to fail. $\hat{y}$ is used to relatively rank tests yielding a test order as needed for RTP. For unsafe RTS, we can derive a *cut-off value*, $\theta \in \mathbb{R}_{[0,1]}$, based on some *cut-off criterion* (see Sec. 2.4.2). It defines the decision boundary to select the test for execution, if $\hat{y} > \theta$, or skip it otherwise. As any modeling technique that learns an accurate mapping from X to Y is suitable, we apply the following set of heuristic ranking models, $M_{h,f_{j,k}}$, and supervised ML classification algorithms, M_{1-5}.

2.3.1 Heuristic Ranking Models ($M_{h,f_{j,k}}$). Heuristic ranking models are widely applied across RTP and unsafe RTS research (e.g., [11, 13, 21, 60]; partly only used as baselines). The intuition is that these models, $M_{h,f_{j,k}}$, predict a failure likelihood solely by considering a single feature $f_{j,k}$. For example, a heuristic could select only those tests that have failed within the previous n CI runs [21]. The underlying ranking model orders tests only by the *last failure* feature, $f_{1,2}$, and selects tests based on a cut-off value n.

As the ranking model ultimately has to output a score $\hat{y} \in \mathbb{R}_{[0,1]}$, each value $x \in \mathbb{R}$ of the selected $f_{j,k}$ needs to be transformed. This is done by using a *min-max-scaler*: $\hat{y} = \frac{x - min(f_{j,k})}{max(f_{j,k}) - min(f_{j,k})}$. Since this inexpensive mathematical transformation requires close to no training effort, heuristic ranking models are naturally efficient. Consequently, in the given example, n must also be transformed by the scaler for $f_{1,2}$ to obtain θ.

2.3.2 Supervised Machine Learning (M_{1-5}). We draw the following five supervised ML classification algorithms from existing work on RTP and unsafe RTS based on how frequently they were used before (at least from two distinct authors): Logistic regression [57, 62] (M_1), Multi-layer perceptron [1, 46, 62] (M_2), Linear support vector machine (SVM) [11, 62][4] (M_3), Random decision forest [2, 8] (M_4), Gradient boosted trees [45, 62] (M_5).

These models' optimal performance will most likely depend on project-specific hyper-parameter tuning. There are manual, systematic, and random search techniques for finding the best set of hyper-parameters [7]. We show a straight-forward approach without hyper-parameter tuning in our empirical study (see Sec. 3.2).

2.4 Evaluation

Goal: Measure the cost-effectiveness of CI-RTP/S approaches for a given dataset D.

2.4.1 Cost-Effectiveness. Early RTS research classifies techniques among other attributes as *safe* and *precise*, if they select all potentially fault-revealing test cases in a modified program (i.e., effectiveness) by ignoring unnecessary test cases (i.e., cost) [64]. Similarly, the average percentage of faults detected per cost ($APFD_c$) metric is usually used to evaluate the fault detection capability (i.e., effectiveness) of RTP techniques with respect to test execution cost [17, 23]. More recent studies additionally employ traditional classification performance metrics, such as accuracy, F_1 score, or Area-Under-the-ROC-Curve (i.e., effectiveness) [4, 46, 62] and measure end-to-end test run-time including analysis overhead (i.e., cost) [26, 42, 79]. Furthermore, research on Pareto efficient multi-objective regression test optimization aims to find the optimal trade-off between multiple objectives, e.g., code coverage, historical fault detection capability, or execution cost [28, 74, 76, 77].

In line with these insights from research and discussions with practitioners from IVU Traffic Technologies, we conclude: Evaluation of CI-RTP/S cost-effectiveness requires considering the trade-off between the time (i.e., cost) invested in testing and the thereby achieved level of fault detection safety (i.e., effectiveness).

There is one caveat to this understanding which is the missing fault-to-failure mapping as we do not seed faults into programs, but use observed real-world failures, i.e., failing tests. We therefore rely entirely on detecting failures rather than faults, assuming a one-to-one mapping as proposed by previous research [50, 60, 69].

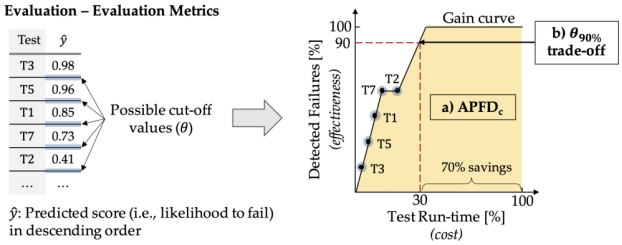


Figure 4: Process to derive the used evaluation metrics for a set of n tests with predicted failure scores ($\hat{y} \in \mathbb{R}^{1\times n}_{[0,1]}$) by using the gain curve.

2.4.2 Evaluation Metrics. Fig. 4 illustrates how evaluation metrics from prior research can be derived that reflect the cost-effectiveness trade-offs for CI-RTP/S: First, for a test suite scheduled for a CI run at timestamp t, $\mathcal{T}_t$, we obtain the failure scores, $\hat{y}$, predicted by the ranking model that is under evaluation. Second, these scores are ranked in descending order, creating a prioritized test suite $\mathcal{T}_t^*$. Third, for each possible cut-off value θ we draw a point into a coordinate system where the x-axis is the percentage of test run-time and the y-axis is the percentage of detected failures, both compared to *retest-all*, i.e., executing all tests. From these points, we can derive the following two commonly used metrics, (a) one for evaluating the total ranking as used for RTP and (b) one for unsafe RTS via some cut-off value θ:

(a) $APFD_c$. Connecting the points yields the so-called (cumulative) *gain curve* which can be further reduced to the area under the gain curve, a single aggregation measure between 0 and 1. This area is referred to as the $APFD_c$ where test costs, meaning the subscript c, are solely reflected by the test run-time [17]. Since the $APFD_c$ is an established *cost-aware* evaluation metric for RTP, we use it to assess the quality of the overall ranking model [13, 23, 60].

[4] Note that SVMs are non-probabilistic since they separate points into classes through hyperplanes. Yet, probability estimates, i.e., scores between 0 and 1, can be derived, e.g., by performing internal cross-validation [63].

(b) Cut-off Trade-offs. Yet, in the case of unsafe RTS, as opposed to RTP, reporting *only* the $APFD_c$ metric is insufficient, as we must ultimately select a subset of tests, $\mathcal{T}' \subseteq \mathcal{T}^*$. Setting the cut-off value θ depends on the acceptance criteria of a project's developers: For higher empirical safety[5], they will set the value of θ close to 0, whereas for low safety, but high time savings, it should be near 1. Therefore, to derive expected RTS cost-effectiveness trade-offs, we further measure the test run-time savings for three different empirical failure detection safety levels (90, 95, and 100%), i.e., three different cut-off values ($\theta_{90\%}$, $\theta_{95\%}$, and $\theta_{100\%}$). These levels are chosen based on the idea of empirical safety (100%), following Facebook's example (95%) [45], and using safety acceptance criteria expressed by our industry partner IVU (90%). Notably, this cut-off criterion is based on the idea of empirical safety levels [45] rather than cutting off tests based on *time constraints* [8, 71].

2.4.3 Model Training and Testing. The proposed ranking models need to be trained on a subset of D before evaluating the trained model on a *different hold-out subset* of D. We delineate this process of model training and testing in the following four steps: Defining training and test splits, model training, model testing, and randomness. Note that *testing* in this context means that the model is examined for its performance on a hold-out test dataset, i.e., on data that was not available during model training.

Training and Test Splits. There is a myriad of ways to split a dataset into training and test set. For instance, for unsafe RTS, Machalica et al. [45] use the most recent week of their three months dataset for testing. Philip et al. [62] train on one year and test on two months of data. There is no general rule, neither about how much data should be used for training and testing, nor about the ratio. Yet, in practice, we need to decide whether to use all available historical data for training or stick to more recent data, which might resemble the *current* failure behavior more accurately [3].

Fig. 5 shows the training-test-splits which we use to measure these possible influences: We divide a time-ordered dataset, D, into 5 equal-sized folds by CI runs. This is based on the widespread 5 fold split in ML research (e.g., [9]). One could also use *absolute* amounts of data for splitting, e.g., always test on one month of data, but there is no argumentation to pick one over another. Notably, we cannot perform cross-validation since there are temporal dependencies between results of CI runs: Training models on past data and testing on more recent data is realistic and most suitable in practice [62].

We obtain different training-test-splits as follows: First, to investigate the best *ratio of training to test folds*, we vary the amount of training folds while keeping the test set at the most *up-to-date* fold. The derived splits (S_{1-3}) with different ratios use 100, 75, and 50% of historical data for training. Recall that we regard the training-test-ratio, i.e., amount of training data, as the first (i) of three parameters of any CI-RTP/S approach that needs to be calibrated.

Second, to examine the sensitivity of a CI-RTP/S approach to data *timeliness*, we extend S_{1-3} by three additional training-test-splits. The intuition is that the *up-to-date* fold, which we use as a test set, might not be representative. However, if the CI-RTP/S approach works well across test sets with different timeliness, we

[5]Due to the lack of deterministic test execution traces or static dependencies, CI-RTP/S can only give *empirical*, i.e., statistical evidence-based, safety guarantees.

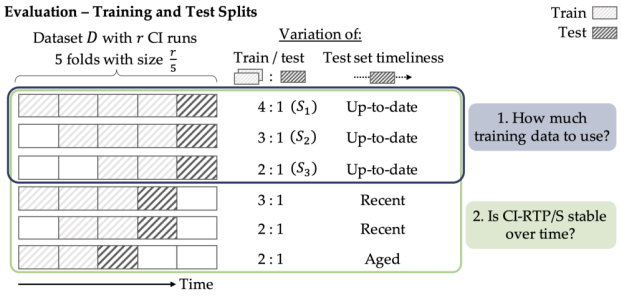


Figure 5: Splitting D into different training and test sets.

can provide better predictability of CI-RTP/S to practitioners, due to stable performance over time. Alternative test sets, i.e., with timeliness *recent* and *aged* (see Fig. 5), may also allow to derive a performance estimation in case there are no failures in the *up-to-date* fold, which disallows constructing the gain curve.

In total, this leaves us with 6 different training-test-splits which we need to evaluate. We refer to a training set as D^{train} and to a test set as D^{test} in the following.

Model Training. The heuristic ranking models, $M_{h,f_{j,k}}$, are created by fitting min-max-scalers on each of the 16 features in D^{train} as described in Sec. 2.3.1. These scalers are then used to predict the failure score for each test $T_{t,i}$ in D^{test} solely based on $f_{j,k}$ in D^{test}. All scores are clipped to be between 0 and 1 resulting in a score $\hat{y} \in \mathbb{R}_{[0,1]}$ for each test. Since a feature might follow an inverse scoring order, where a high feature value indicates a low failure score, we calculate $APFD_c$ values for $\hat{y}$ and $1 - \hat{y}$ on the training set and only use the better performing one for model testing.

Additionally, for each project, we train a ranking model for each ML algorithm and feature set as well as on the composition of all four feature sets. This results in $(4 + 1) * 5 = 25$ (feature set, model)-combinations to be evaluated per training-test-split.

Model Testing. Each created ranking model is used to predict the failure score of each test $T_{t,i}$, i.e., each row, in D^{test}. If the predicted failure scores of two tests are equal, the test with the shorter last execution duration is executed first, since the last test duration has proven to be a reasonable baseline [13, 60]. Thereby, we obtain a test ranking for each CI run in D^{test}. We follow prior research [11, 50, 60] by reporting our evaluation metrics averaged across all CI runs in D^{test} that contained failures: *Avg. $APFD_c$* (RTP) and *avg. test time savings* (RTS).

Randomness. Several ML algorithms involve randomization. We repeat the experiments with 30 different random seeds to reduce the impact of randomness [23, 71]. Results in our empirical study (see Sec. 3.3) report the mean of evaluation metrics.

3 EMPIRICAL STUDY

We perform an empirical study to evaluate CI-RTP/S approaches built and calibrated using our methodology; and to derive evidence-based guidelines and cost-effectiveness expectations for practitioners. Therefore, we strive to answer the following research questions (RQs):

- **RQ**$_1$: How sensitive is the cost-effectiveness of CI-RTP/S to different parameterizations regarding amount of training data, choice of features, and ranking model?
- **RQ**$_2$: How do CI-RTP/S approaches built with our methodology compare against baseline RTP and RTS techniques in terms of cost-effectiveness?

3.1 Study Subjects

Table 1 lists the selected 23 software projects from industry (3) and open-source development (20).

3.1.1 Industrial Projects. These are provided by our industry partner IVU Traffic Technologies, each counting several millions of source lines of code (SLOC). One project is primarily written in C/C++ (P_1), two in Java (P_{2-3}). Additionally, web-based and native graphical user interface (GUI) clients are part of these code bases which are programmed in different domain specific languages (DSLs) or other general purpose programming languages (GPLs) such as JavaScript. Their test suites, besides unit testing, involve integration- as well as system-level testing often performed across project boundaries. Developers commit their changes directly to the main VCS development line, where the company-internal Jenkins CI system collects commits and triggers a new *retest-all* CI run once the previous run has finished. Once a test run is finished, a Jenkins plugin aggregates all XUnit test results into a structured test report in XML or JSON format[6].

3.1.2 Open-source Projects. These are part of a recently published dataset for RTP, RTPTorrent (Mattis et al., Zenodo, CC BY 4.0[7]), that aims to deliver a representative sample of all Java projects on GitHub [50, 51]. We discovered that most of them (14/20) additionally use more GPLs other than Java (e.g., C++, Python) or DSLs (e.g., SQL, YAML). As RTPTorrent is yet missing some required links between CI runs and respective VCS commits, we used the underlying massive TravisTorrent [6] CI dataset to extend RTPTorrent. If there are multiple VCS branches that are tested in the CI system, we use historical data from *all* of these branches. The same applies for multiple sub-stages in a test stage, where different sub-stages, e.g., for different compiler versions, might report the same failures. In the worst case, this leads to an over- or undersampling of failures. We still keep these data to not waste potentially valuable information.

3.1.3 Datasets. We argue that this set of projects resembles reality, where RTP and RTS techniques have to cope with multi-language software of varying size as well as test-levels, i.e., unit-, integration, or system-level [12, 80]. More than 37,000 CI test logs and 76,000 VCS commits were analyzed. We publish the resulting 23 datasets, $D_{P_{1-23}}$, as part of our supplemental material.

3.2 Experimental Setup

Recall that we identify CI-RTP/S approaches as triples of the parameters (i) training data amount (i.e., training-test-ratio), (ii) features, and (iii) ranking model. All studied settings for these parameters described in Sec. 2 are summarized in Table 2. Due to the combinatorial explosion of assessed study subjects, random seeds, training-test-splits, features (or feature sets), and ranking models, the experiments were run on a highly parallelized cluster infrastructure. The measured total CPU time was more than 50,000 hours.

Since we aim to constitute a generic example of applying our methodology, we do not perform elaborate hyper-parameter grid search for ML algorithms. Instead, we follow Chen et al. [13] and stick to the default model hyper-parameters provided in the *scikit-learn* package [59], but use the *LightGBM* package for a more lightweight implementation of gradient boosting [35]. Notably, before model training each feature is normalized.

In our supplemental material, we provide the source code necessary to reproduce our results from the created 23 datasets, $D_{P_{1-23}}$.

3.3 Results

In the following, we discuss the empirical results and address the RQs. Detailed results are provided with the supplemental material.

3.3.1 RQ$_1$: Cost-Effectiveness Sensitivity Analysis. We aim to analyze how sensitive the cost-effectiveness of the CI-RTP/S approaches, built with our methodology, is to the parameters (i)-(iii) and find calibrations that are empirically superior to others (see Table 2). When comparing different parameter settings, we use the APFD$_c$ as calculated for each CI run in a project's test dataset D^{test} and average it over these runs to obtain the avg. APFD$_c$. This metric is then considered across projects for sensitivity analysis. We run a one-way analysis of variance (ANOVA) for each parameter to investigate its individual influence on the avg. APFD$_c$. Therefore, we vary its value, while having the other two parameters at their project-specific best setting. Our approach for sensitivity analysis of parameterization follows related work on RTS and RTP [8, 60].

(i) Training Data Amount. To check how much training data is beneficial for CI-RTP/S, we study whether there are significant differences in the means of the avg. APFD$_c$ across projects for S_{1-3}. To choose the appropriate statistical test for the ANOVA, we first perform the Shapiro-Wilk test with Bonferroni correction to check for normality which cannot be rejected with a minimal p-value of 0.045 (significance level $\alpha = 0.05$ corrected by $|S| = 3$ is $\alpha_{norm} = 0.017$). We then use Bartlett's test for homoscedasticity, which is also not rejected at a p-value of 0.977. Hence, with normal and homoscedastic data we can perform a repeated measures ANOVA. We fail to reject the null hypothesis (p-value = 0.717), indicating no significant difference between the mean values of S_1, S_2, and S_3. Among them, S_2 shows the highest mean avg. APFD$_c$ of 0.896.

Prior research has trained ranking models only on *faulty* CI runs [11], that is runs that contain at least one test failure, or by using *all* available CI runs [45]. To check if there are any significant differences, we repeated all experiments a second time, but this time we only trained on those CI runs in the training set that contained failures. Using the same procedure as before, we find that differences in means are insignificant (p-value = 0.284) by using a paired t-test, which is suitable as data is normally distributed (p-value = 0.059) and we have two populations, that is *only faulty* and *all* CI runs.

Overall, we can summarize that using less training data does not harm cost-effectiveness of CI-RTP/S. Even limiting ourselves to only using faulty CI runs did not negatively impact the avg. APFD$_c$.

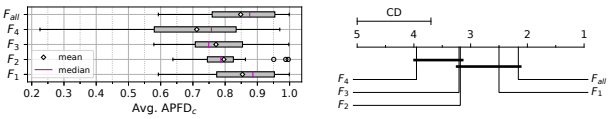
[6] Jenkins JUnit Plugin: https://plugins.jenkins.io/junit/
[7] CC Attribution 4.0 International: https://creativecommons.org/licenses/by/4.0/

Table 1: Study subject statistics

P_{ID}	Project	# SLOC	Time period [days]	# Commits	# CI runs	# Failing CI runs	# Test runs	# Tests	# Failures	Avg. test stage duration [sec]	Avg./median # failures per failing test stage
P_1	IVU_Cpp	≫ 1M	267	8,632	3,996	2,841	3,608.4K	1,240	25,973	5,454	9.1/3.0
P_2	IVU_Java_1	≫ 1M	313	7,747	943	876	178.7K	279	14,568	65,557	16.6/6.0
P_3	IVU_Java_2	≫ 1M	699	7,965	3,209	1,521	3,526.3K	1,278	7,603	5,330	5.0/2.0
P_4	jcabi-github	64K	872	1,050	809	205	398.1K	201	740	778	2.2/2.0
P_5	jade4j	10K	1,539	400	358	59	35.9K	46	1,323	4	13.8/19.0
P_6	optiq	243K	395	560	458	38	55.3K	63	110	2,168	1.6/1.0
P_7	buck	562K	307	2,517	846	339	586.1K	864	1,511	1,650	4.5/1.0
P_8	jetty.project	346K	63	237	192	174	63.9K	787	415	508	1.3/1.0
P_9	jsprit	59K	368	326	267	14	91.8K	107	123	23	2.4/1.0
P_{10}	LittleProxy	13K	1,580	353	271	62	11.0K	50	172	134	2.8/2.0
P_{11}	dynjs	57K	1,163	517	385	25	68.5K	83	496	15	12.1/1.0
P_{12}	sling	673K	213	13,376	1,403	812	268.1K	304	1,158	420	1.4/1.0
P_{13}	HikariCP	13K	661	1,787	1,575	125	44.0K	23	383	58	3.1/1.0
P_{14}	wicket-bootstrap	42K	1,245	1,150	904	342	41.4K	91	9,007	8	26.3/29.0
P_{15}	okhttp	69K	1,423	3,518	3,412	744	236.5K	266	939	108	1.2/1.0
P_{16}	titan	59K	747	621	384	157	43.3K	107	551	2,366	2.2/2.0
P_{17}	deeplearning4j	138K	727	1,071	982	566	14.6K	174	908	477	1.6/1.0
P_{18}	cloudify	132K	909	6,048	4,973	496	283.6K	116	602	92	1.2/1.0
P_{19}	graylog2-server	127K	1,381	5,414	3,891	165	798.5K	250	403	792	1.6/1.0
P_{20}	Achilles	54K	1,114	904	642	23	139.9K	627	162	139	6.0/2.0
P_{21}	DSpace	384K	1,043	2,489	1,929	82	122.1K	83	1,697	130	20.7/35.0
P_{22}	sonarqube	661K	532	7,899	4,286	488	6,696.0K	3,122	2,156	334	3.5/1.0
P_{23}	jOOQ	351K	961	1,525	1,318	403	81.5K	51	573	13	1.1/1.0

Table 2: Parameters of CI-RTP/S approaches: (i) Training data amount S, (ii) features F, (iii) ranking models M

S_1	100% of available historical data
S_2	75% of available historical data
S_3	50% of available historical data
F_1	Failure count ($f_{1,1}$), Last failure ($f_{1,2}$), Transition count ($f_{1,3}$) Last transition ($f_{1,4}$), Avg. test duration ($f_{1,5}$)
F_2	Max. (test, file)-failure freq. ($f_{2,1}$), Max. (test, file)-failure freq. (rel.) ($f_{2,2}$), Max. (test, file)-transition freq. ($f_{2,3}$), Max. (test, file)-transition freq. (rel.) ($f_{2,4}$),
F_3	Min. file path distance ($f_{3,1}$), Max. file path token similarity ($f_{3,2}$), Min. file name distance ($f_{3,3}$)
F_4	Distinct authors ($f_{4,1}$), Changeset cardinality ($f_{4,2}$), Amount of commits ($f_{4,3}$), Distinct file extensions ($f_{4,4}$)
M_1	Logistic regression
M_2	Multi-layer perceptron
M_3	Linear SVM
M_4	Random decision forest
M_5	Gradient boosted trees
$M_{h,f_{j,k}}$	Heuristic ranking models



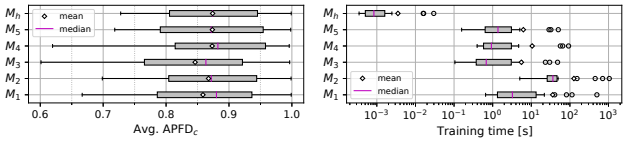
(a) Avg. APFD$_c$ across projects **(b) Pairwise comparison**

Figure 6: Sensitivity analysis of *features* F (ii)

(ii) Features. In Sec. 2.2 we described how features are grouped into feature sets F_{1-4} to increase comprehensiveness. This allows us to study their individual cost-effectiveness and empirical differences among them. Fig. 6a shows the distribution of the avg. APFD$_c$ for each feature set F_{1-4} and F_{all}. Again, we assume normality after conducting a Shapiro-Wilk test with Bonferroni correction which yields a minimal observed p-value of 0.023 ($\alpha_{norm} = 0.01$). Bartlett's test for homoscedasticity is further rejected at p-value 0.02, assuming heteroscedasticity. Thus, we use the non-parametric ANOVA Friedman test to check for differences in the means of avg. APFD$_c$. It is rejected at p-value 0.002 indicating that there are significant differences which we further explore with the *post hoc* Nemenyi test as proposed by Demšar [14]: It compares all feature sets pairwise based on the absolute differences of their avg. rankings. For α a *critical difference (CD)* is determined; if the avg. ranking difference is greater than CD, the null hypothesis that they have equal performance is rejected. Fig. 6b visualizes these pairwise differences in the avg. ranks through a CD diagram: If two feature sets are connected by a horizontal bar, they are not significantly different to each other [32]. The diagram indicates that F_4 is significantly worse than F_{all} (best in pairwise comparison) and F_1 (best among F_{1-4}), questioning the usefulness of features in F_4. Further, we find that even though differences in avg. ranks are not significant, F_1 and F_{all} have higher mean avg. APFD$_c$ than other feature sets by at least 0.05. This emphasizes the usefulness of features from test history (F_1).

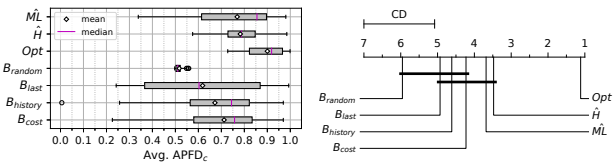
As heuristic ranking models do not rely on entire feature sets but only on single features, we conduct the sensitivity analysis for all single features $f_{j,k}$ as well, i.e., their respective heuristic ranking models $M_{h,f_{j,k}}$. The ANOVA shows that there are no significant differences between features (Shapiro-Wilk test rejected at p-value 0.001; Friedman test not rejected at p-value 0.063). However, from the five features with highest mean rank and median avg. APFD$_c$, three are from feature set F_1 ($f_{1,4}$, $f_{1,1}$, $f_{1,5}$), and two from F_2 ($f_{2,1}$, $f_{2,2}$), with the best one being $f_{2,1}$, i.e., max. (test,file)-failure frequency. Again, this emphasizes that features using test history correlate with better cost-effectiveness. There seems to be some combination of features that leads to superiority of F_1 over F_4. This observation motivates the use of more elaborate statistical feature selection techniques in the future (see Sec. 3.6).

(iii) Ranking Model. To investigate the cost-effectiveness of each ranking model M, we perform ANOVA twice, with the response variables avg. APFD$_c$ and *training time*, respectively. We consider the latter as a reasonable proxy for model efficiency as our experimental results show that model inference time is negligibly small (see experiment results in supplemental material). Fig. 7 shows the distributions of the avg. APFD$_c$ and the training times of ranking



(a) Avg. APFD$_c$ across projects **(b) Training time across projects**

Figure 7: Sensitivity analysis of *ranking models* M (iii)



(a) Avg. APFD$_c$ across projects **(b) Pairwise comparison**

Figure 8: Comparison of RTP cost-effectiveness

models across all projects. For the first ANOVA, we perform a repeated measures ANOVA, as the data is normal (p-value = 0.087) and homoscedastic (p-value = 0.953). Since the null hypothesis is not rejected at p-value 0.066, we assume that there are no statistically significant differences in the mean avg. APFD$_c$. M_5 (gradient boosted trees) and M_h have the highest mean avg. APFD$_c$ (both 0.874). The second ANOVA shows significant differences in training time: We reject the Friedman test at p-value <0.001 (non-parametric ANOVA due to non-normality at p-value <0.001). Without further inspection, it is obvious that M_h (as expected) is far more efficient than the ML algorithms as its training procedure is simply a mathematical transformation. Yet, despite its simplicity, the cost-effectiveness of M_h is still comparable.

While prior research also finds M_5 to be particularly effective [45], interestingly, these findings rather suggest focusing on existing simple heuristic ranking models (M_h) instead of investing the effort in training complex ML models from prior research.

> *RQ$_1$: We find that CI-RTP/S cost-effectiveness is sensitive to the choice of features, but is not significantly impacted by the amount of training data or the ranking model. We empirically determine that the best approaches contain features from test history and use heuristic ranking models.*

3.3.2 RQ$_2$: Comparative RTP and RTS Performance. We aim to provide estimations on the cost-effectiveness of CI-RTP/S approaches for RTP and unsafe RTS. We have motivated in Sec. 1 that it is of particular interest for practitioners to know how much cost-effectiveness is sacrificed if using only a semi-optimally calibrated approach. Thus, we compare the following CI-RTP/S approaches including four baselines for RTP and unsafe RTS.

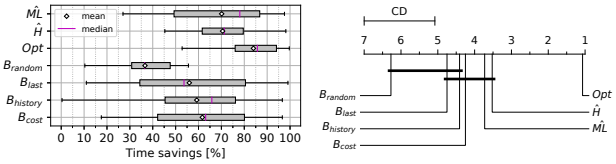
- $\hat{\boldsymbol{ML}}$: Empirically best ML ranking model from RQ$_1$, i.e., M_5 (gradient boosted trees) with F_{all} on S_2 (75% training data)
- $\hat{\boldsymbol{H}}$: Empirically best heuristic ranking model from RQ$_1$, i.e., $M_{h,f_{2,1}}$ (max. (test,file)-failure freq.) on S_2 (75% training data)
- ***Opt***: Always uses optimally calibrated approach from our methodology for each project; implies high effort in practice, as all combinations of parameter settings are computed.
- $\boldsymbol{B_{random}}$: Baseline ranking tests in random order [11, 17, 20]
- $\boldsymbol{B_{last}}$: Baseline ranking tests in ascending order by the time since the last failure (i.e., $f_{1,2}$) [21, 71]
- $\boldsymbol{B_{history}}$: Baseline ranking tests in descending order by the amount of historical failures (i.e., $f_{1,1}$) [3, 60]
- $\boldsymbol{B_{cost}}$: *Cost-only* baseline ranking tests in ascending order by their last execution time [13, 60]

The distribution of their avg. APFD$_c$ across projects is shown in Fig. 8a, reflecting the cost-effectiveness for RTP. To compare the median values, we perform the non-parametric ANOVA Friedman test assuming non-normality at minimal p-value of 0.001 (α_{norm} = 0.007). It is rejected at p-value <0.001 indicating that there are significant differences. Again, we use the *post hoc* Nemenyi test for pairwise comparison of avg. ranks and report the results in Fig. 8b. We can see that *Opt* significantly outperforms all other approaches, which is also reflected by the highest median avg. APFD$_c$ of 0.919 with the lowest median absolute deviation of 0.1. Although not statistically significant, the medians of the semi-optimally calibrated approaches $\hat{ML}$ (0.855) and $\hat{H}$ (0.787) are also better than all baselines. Notably, the spread of the avg. APFD$_c$ for $\hat{H}$ with respect to median absolute deviation is considerably smaller (0.177) than the one for $\hat{ML}$ (0.239). Similar to prior research [60], B_{cost} seems to be the best performing baseline (median: 0.757), yet not statistically significant across studied projects. Moreover, all approaches and baselines (partly significantly) outperform random ordering (B_{random}), which is also in line with previous results (see Fig. 8b) [11, 18, 19, 71]. Interestingly, we found that for 5 and 11 projects, respectively, the baselines $B_{history}$ and B_{last} performed worse than $M_{h,f_{1,1}}$ and $M_{h,f_{1,2}}$ which use the same features, but followed an inverse scoring order in these projects.

Regarding RTS, Fig. 9a shows the distribution of the avg. time savings for RTS across projects when setting the cut-off value to $\theta_{90\%}$, i.e., 90% empirical failure detection safety. The ANOVA, reported in Fig. 9b, has similar results (Shapiro-Wilk: p-value 0.029; Bartlett: p-value <0.001; Friedman: p-value <0.001), which is not surprising, as we generally expect good RTP to correlate with good RTS approaches. Though the project-specific best approach used for *Opt* is not necessarily the same for RTS and RTP: For $\theta_{90\%}$, 19 out of 23 projects have the same best project-specific approach.

Overall, using *Opt* we are able to save 84% of testing time on average across projects. However, even with the semi-optimally calibrated approaches, $\hat{H}$ and $\hat{ML}$, savings of on average >70% are achieved. While the baselines have relatively high average savings as well (up to 61.7%), they suffer a large spread across projects. For $\theta_{95\%}$ and $\theta_{100\%}$, we find 83.1 ± 13.8% and 82.8 ± 14.4% average test time savings with *Opt*, respectively. $\hat{H}$ and $\hat{ML}$ achieve average savings of 69.8±10.2% and 69.7±14.2% for $\theta_{95\%}$ and 69.4±10.5% and 69.4 ± 14.3% for $\theta_{100\%}$. The overall conclusions regarding relative performance of the seven compared approaches (including four baselines) remain similar. All numbers and figures for $\theta_{95\%}$ and $\theta_{100\%}$ are part of the provided supplemental material.

Finally, we investigate to what extent the ranking performance of *Opt*, $\hat{H}$, and $\hat{ML}$ (as they have been calibrated on the *up-to-date*



(a) Avg. savings across projects **(b) Pairwise comparison**

Figure 9: Comparison of RTS cost-effectiveness for $\theta_{90\%}$

Table 3: Time stability measured by avg. APFD$_c$ across 6 different training-test-splits ($\mu \pm \sigma$)

P_{ID}	*Opt*		$\hat{H}$	$\hat{ML}$
P_1	(F_{all}, M_4)	0.97 ± 0.00	0.74 ± 0.02	0.94 ± 0.02
P_2	$(f_{1,2}, M_h)$	0.90 ± 0.03	0.80 ± 0.02	0.88 ± 0.03
P_3	(F_{all}, M_4)	0.95 ± 0.01	0.77 ± 0.03	0.81 ± 0.06
P_4	(F_1, M_1)	0.84 ± 0.07	0.81 ± 0.08	0.64 ± 0.15
P_5	$(f_{3,3}, M_h)$	0.87 ± 0.16	0.72 ± 0.07	0.65 ± 0.22
P_6	$(f_{1,4}, M_h)$	0.64 ± 0.11	0.70 ± 0.11	0.65 ± 0.23
P_7	$(f_{1,5}, M_h)$	0.98 ± 0.01	0.91 ± 0.02	0.87 ± 0.10
P_8	(F_{all}, M_1)	0.86 ± 0.10	0.75 ± 0.12	0.81 ± 0.08
P_9	(F_{all}, M_1)	0.70 ± 0.30	0.81 ± 0.09	0.72 ± 0.25
P_{10}	(F_2, M_4)	0.63 ± 0.10	0.63 ± 0.10	0.42 ± 0.10
P_{11}	$(f_{3,3}, M_h)$	0.80 ± 0.21	0.83 ± 0.17	0.46 ± 0.30
P_{12}	(F_{all}, M_5)	0.86 ± 0.09	0.76 ± 0.12	0.86 ± 0.09
P_{13}	(F_4, M_5)	0.79 ± 0.05	0.69 ± 0.06	0.56 ± 0.08
P_{14}	$(f_{1,2}, M_h)$	0.79 ± 0.04	0.80 ± 0.02	0.72 ± 0.08
P_{15}	(F_1, M_2)	0.82 ± 0.05	0.81 ± 0.02	0.77 ± 0.09
P_{16}	(F_1, M_1)	0.69 ± 0.09	0.70 ± 0.06	0.68 ± 0.04
P_{17}	$(f_{1,1}, M_h)$	0.79 ± 0.11	0.80 ± 0.07	0.76 ± 0.10
P_{18}	(F_{all}, M_5)	0.85 ± 0.10	0.61 ± 0.10	0.85 ± 0.10
P_{19}	$(f_{3,3}, M_h)$	0.90 ± 0.17	0.90 ± 0.18	0.81 ± 0.15
P_{20}	$(f_{3,1}, M_h)$	0.90 ± 0.07	0.84 ± 0.08	0.67 ± 0.30
P_{21}	(F_1, M_5)	0.70 ± 0.14	0.75 ± 0.04	0.56 ± 0.14
P_{22}	(F_2, M_1)	0.67 ± 0.06	0.72 ± 0.03	0.53 ± 0.06
P_{23}	(F_{all}, M_4)	0.96 ± 0.00	0.86 ± 0.03	0.92 ± 0.00
Avg.		0.82 ± 0.07	0.77 ± 0.04	0.72 ± 0.08

test set) is stable over different test sets in time (see Fig. 5). Table 3 lists the mean and standard deviation of their avg. APFD$_c$ for each project, if applied across all 6 available training-test-splits. While *Opt* is always the best approach on the *up-to-date* test set, it is not necessarily the best one averaged across all training-test-splits. If there were two approaches performing equally well on the *up-to-date* test set in terms of their avg. APFD$_c$, hence being candidates for *Opt*, we decide in favor of the one with smaller standard deviation of the APFD$_c$. The cost-effectiveness oscillates considerably over time with an average σ of 0.07 for *Opt*, σ of 0.08 $\hat{ML}$, and 0.04 for $\hat{H}$. Hence, we conclude that re-adaptation intervals should be kept short, as optimal calibration of CI-RTP/S fluctuates over time.

> *RQ$_2$: We find that CI-RTP/S approaches outperform established baselines and save on average 84% of test run-time while retaining 90% of empirical failure detection safety. However, CI-RTP/S is unstable over time, thus requiring regular adaptation.*

3.4 Guidelines and Expectations for Practice

In summary, we derive the following practical implications from the findings of our empirical study:

(1) CI-RTP/S does not need large amounts of training data per se. It suffices to use the most recent or even only faulty CI runs. This speeds up re-adaptation and decreases required storage.
(2) Features from test history are frequently performing well, yet, in our experiments, adding VCS metadata can increase cost-effectiveness: F_{all} has been the best feature set.
(3) Rather naïve, inexpensive heuristic ranking models often outperform sophisticated ML algorithms.
(4) Calibrating the project-specific *optimal* (*Opt*) CI-RTP/S approach gives significant cost-effectiveness benefits over semi-optimally calibrated approaches or baseline models.
(5) CI-RTP/S approaches are not stable over time. Frequently re-adapting CI-RTP/S to more recent development is advisable.
(6) Unsafe RTS, even if solely based on metadata from CI and VCS, can achieve considerable test run-time savings (on average 84% while detecting 90% of failures across projects from our study).

3.5 Application in Industry

Besides the empirical results on the performance expectations of CI-RTP/S reported above, we share some initial experiences and challenges from deploying our methodology at IVU Traffic Technologies who supported and partially sponsored this research.

We implemented our methodology as a web service that is deployed in the company's infrastructure and integrated with their Jenkins CI system. As described in Sec. 3.1.1, the existing main Jenkins pipelines continuously execute all regression tests. Depending on the project this is either done in random order, to detect and prevent test order dependencies (see [39]), or by the alphabetical naming order of tests. In addition to the existing pipeline, we created a parallel RTS pipeline for project P_2 from our empirical study: This pipeline first queries the web service with the introduced changeset since the last CI run and a desired empirical safety level (the default is 90%) and then only executes the subset of tests retrieved from the web service. The reason why we choose this parallel setup for now, is to build up trust in the RTS mechanism among developers. They can directly compare results from the existing (safe) *retest-all* to the RTS pipeline, which makes test time savings transparent.

In this industry setting, we decided to *only* include *heuristic* ranking models ($M_{h,f_{j,k}}$) for the following reasons: First, as we have shown empirically, these heuristics often outperform complex ML algorithms and require low training effort in both time and computation resources. They are also non-randomized, which eliminates the need for costly repeated experiments. Second, they are easily interpreted by developers, who are not necessarily experts in predictive modeling, which might increase overall acceptance of the used models. As our guidelines from the empirical study suggest, the web service re-adapts all ranking models every night including the new data from the last day which are fetched from Jenkins. For the subsequent day, the web service will then only use the best performing model to rank tests. We follow the assumption by Facebook [45] that model performance on our test dataset is a good approximation of the model's general performance on unseen data. Still, to regularly check this assumption, we store trained models and inspect in hindsight how they performed on the following day, i.e., if the empirical safety level carried over from the test set.

Our parallel RTS pipeline has been in use in project P_2 for six weeks. This project contains a large fraction of relatively *long running* Java tests that operate across language boundaries (Java and C++) and often have long test setup times for database schemas.

During the considered time period, the RTS pipeline executed 366 CI runs, where 176 included at least one failure. Across those failing CI runs, the realized test time savings were on average 19.8% with 93.4% of failures being detected. We observe that the empirical safety level was only notably violated (below 90%) when the code base underwent major refactorings which were accompanied by a sudden increase in failures. However, the test time savings are considerably smaller than what we achieved for P_2 during the empirical study and for the open-source projects. There are (at least) two possible reasons for this observation: First, we found that the setup times often significantly impact the test execution time and created database schemas are cached and re-used for subsequent tests. Hence, even if certain tests are excluded, the considerable impact of the test setup time will still be prevalent if *any* of the tests requiring that setup is executed. Second, if multiple tests failed in a CI run, we observed an increase in the number of selected tests on the following days. We expect this to be a consequence of our decision to rely on heuristic ranking models only: $M_{h,f_{1,2}}$ has been the best for P_2 and thus, if multiple tests fail once, they will be selected for execution throughout the subsequent days which negatively affects the test time savings.

Even though these initially realized time savings are smaller than in the empirical study, IVU engineers are positive about the achieved results and will deploy parallel RTS pipelines for more projects. They expect test setup times to have significantly less impact in their other projects. In fact, the test setup caching mechanism is very project-specific and little prior research exists on test dependency aware RTP and RTS [39]. Thus, while our empirical study of open-source projects establishes comparability, context-specific practical challenges can impact CI-RTP/S cost-effectiveness and we will further investigate how to address them at IVU.

In the next step, we aim to reduce the frequency of *retest-all* cycles. Instead of parallel execution, the RTS pipeline will be the default CI pipeline. Re-executing all tests in certain intervals will still be required as the thereby obtained test outcome data are necessary for re-adapting ranking models.

3.6 Threats to Validity

3.6.1 External Validity. The main threats to external validity concern the representativeness of results. We address them by studying a heterogeneous set of real-world projects, but cannot, by nature of empirical studies, easily generalize our findings beyond our dataset. Since this is a known limitation of ML models as they always depend on the dataset quality, we followed established data mining and ML practices for splitting data, repeating randomized experiments, and training and testing models to mitigate these threats.

The investigated time periods of CI and VCS history might contain irregular development behavior, e.g., unusual maintenance activities. Therefore, we create multiple training-test-splits and investigate time stability of performances.

Since we rely on test execution time as reported in the CI logs, there is a threat from fluctuations within one CI test run due to irregular workload on the build machine. Tests are usually run in isolation inside CI environments to reduce such side-effects, but it might still affect the concrete values of reported evaluation metrics. Similar to prior work [60], we address this threat by our large dataset of CI runs and by conducting rigorous statistical analyses to ensure that findings are significant across projects.

Furthermore, we deliberately exclude an automated feature selection process or model hyper-parameter tuning. While this might limit the performance of ML algorithms compared to more sensible tuning, it enables us to perform fine-grained sensitivity analyses. We deem the investigation of automated feature selection techniques as an important future task to prune our current feature sets. In addition, while we focus on point-wise ranking models, there are recent studies on other approaches such as reinforcement learning (RL), which are beyond the scope of this work [8, 71].

As described in Sec. 2.4.1, we rely on a one-to-one failure-to-fault mapping similar to previous research [50, 60, 69]. While this assumption might distort results since faults often cause multiple failures, prior research on RTP shows that different mappings still lead to similar overall conclusions [60].

Finally, the presence of flaky tests may impact the effectiveness of CI-RTP/S. Existing research is not unequivocal regarding the expected effect of flaky tests: While at Facebook [45], the presence of flaky tests does not preclude the applicability of CI-RTP/S, Peng et al. [60] see substantial impact for some RTP techniques. We argue that flaky test detection requires special efforts and can be performed on top of our methodology. Due to resource constraints, we cannot re-run more than 37,000 CI test histories multiple times to *deflake* each dataset as proposed by prior work [45]. At IVU, flaky tests are currently not documented, yet developers are encouraged to fix such tests immediately when they behave non-deterministically.

3.6.2 Internal Validity. We identify the integrity of exploited data sources as well as the correctness of the implemented feature engineering and evaluation analysis as the main internal threat. To address this, we wrote run-time assertions and unit tests that discover invalid data and check feature computations. Furthermore, we manually checked results for their validity with IVU engineers.

4 RELATED WORK

Several RTP and unsafe RTS techniques have been proposed which incorporate other information than traditional white-box program analyses to predict test failures and rank tests. Throughout this paper, we have referenced existing research that we have consolidated into our methodology. While we focus on techniques that solely rely on CI and VCS metadata (CI-RTP/S), there is also significant related work which uses other additional information that is non-guaranteed in CI settings.

Studies on techniques that use such additional information *beyond* CI and VCS metadata have shown their effectiveness in specific contexts (i.e., single projects or organizations) [1, 2, 4, 11, 45, 47, 48, 54, 57, 62] as well as across multiple projects [5, 8, 13, 29, 33, 36, 40, 46, 49, 53, 56, 57, 60, 67]. We consider the following to be most relevant for our work: Machalica et al. [45] report a reduction of testing infrastructure cost by 50% and test executions by >66% at Facebook while retaining 95% empirical failure detection safety. They train a failure prediction ML model on features from CI and VCS metadata as well as static build dependencies and project identifiers. Busjaeger and Xie [11] train a linear SVM on black- and white-box (i.e., code coverage) features obtained from Salesforce's code repository and CI system. By ranking and selecting tests, they

achieve a trade-off of executing 3% of all tests to detect 75% of the failures. Chen et al. [13] train ML models to predict the effectiveness of RTP techniques by using features from test coverage and testing time. They find that there are no universally optimal RTP techniques across projects which supports findings in the related field of defect prediction [82]. Bertolino et al. [8] compare ten ML algorithms for ranking tests to provide guidelines on when to choose RL over supervised ML or vice versa. They use features from white-box code and dependency analysis as well as test history, and evaluate algorithms' performance on six open-source subjects with artificial faults. Henard et al. [29] provide an experimental comparison of 10 white-box and 10 black-box RTP techniques from prior research. In their study on five C programs from the software infrastructure repository [15] with seeded faults, they find that black-box techniques based on combinatorial interaction testing [10, 61] and test input diversity [24, 30, 31] perform comparably well to white-box approaches. While these black-box techniques are also applicable if there is no source code access, they require either a program's test inputs or a model thereof, which goes beyond CI and VCS metadata. Yoo and Harman [74, 75, 77] introduce the concept of Pareto efficient multi-objective regression test optimization to account for trade-offs between test criteria for different types of testing (e.g., structural and functional). Defect hypotheses and associated features in our methodology are drawn from their work (see Sec. 2.2), but we cannot directly apply their techniques due to the lack of required coverage information. Similarly, we cannot directly apply the unsafe RTS approach by Kim and Porter [36], who were among the first to create statistical ranking models for tests based on past-fault coverage, i.e., tests' history, and function coverage. Peng et al. [60] empirically study information retrieval (IR) techniques for RTP as first proposed by Saha et al. [67]. Their hybrid technique combines features from textual program changes and, similar to our work, test execution time and test failure history. It outperforms coverage-based RTP techniques and the baselines B_{cost} and $B_{history}$ (see Sec. 3.3.2) on a real-world dataset, and they argue for the "necessity [...] to better balance textual, cost, and historical information for more powerful test prioritization" [60]. We deem our approach with advanced feature engineering and predictive modeling to be one step in that direction, albeit excluding white-box textual analysis. Notably, analyzing textual code changes is possible, if CI and VCS metadata contain code *diffs* and tests' source code is accessible. However, going beyond the scope of our methodology, effective IR techniques further employ programming language-specific analysis (e.g., building an abstract syntax tree) or computationally expensive topic modeling [49, 60, 67].

Similarly, effectiveness for specific contexts [3, 16, 21, 38, 43, 68] and across projects [37, 44, 71, 81] has been studied for CI-RTP/S. However, none of the studies that investigate multiple projects uses VCS metadata; they exclusively rely on historical test execution information (i.e., CI logs). Using both kinds of data, as done in our study, indicates that features from CI logs alone are indeed powerful, but including VCS features can further increase cost-effectiveness. We are not aware of related work that uses *both* kinds of data and individually measures associated features' cost-effectiveness across projects. We consider the following papers to be most relevant for our work: Elbaum et al. [21] were the first to apply CI-RTP/S at industry scale: They used the simple heuristic B_{last} (see Sec. 3.3.2) to cost-effectively prioritize and select tests in Google's pre- and post-submit testing process. Spieker et al. [71] use RL for CI-RTP/S. On ABB's and Google's CI test history, their approach achieves competitive performance to simple RTP heuristics (e.g., B_{last} from Sec. 3.3.2). They formulate unsafe RTS as a time-constrained RTP problem, where the cut-off value is determined by the available test time. In contrast, we define cut-off values by empirical failure detection safety levels.

Finally, we have alluded to why sensitivity of cost-effectiveness to size, timeliness, and variety of data is important for CI-RTP/S. Prior work has investigated how much historical data can be beneficial [3, 8, 71]. Research conducted at Microsoft further showed fluctuating cost-effectiveness over time [33]. The impact of data variety has been studied by analyzing performance of predictive features [4, 8, 11, 45]. However, these studies either focus on specific industrial contexts or use more information than only CI and VCS metadata. In summary, we are not aware of prior work on CI-RTP/S approaches that studies how sensitive their cost-effectiveness is to associated parameters, performs *cost-aware* evaluation on real-world failures from industrial and open-source projects, and derives empirical guidelines for calibrating CI-RTP/S in practice.

5 CONCLUSION

Unsafe RTS and RTP techniques that exclusively rely on CI and VCS metadata (CI-RTP/S) are attractive alternatives to traditional, more intrusive techniques: They are inexpensive, language-agnostic, easy to transfer—no program or code access is necessary—, and have been shown to work well in different contexts. However, aspects of their design and evaluation are scattered across research, leaving practitioners to identify insights that apply to their context. Besides, adequately calibrating these techniques often requires high effort and experience with predictive modeling. Still, empirical calibration guidelines are not available. Instead of proposing new techniques, we consolidate existing RTP and unsafe RTS research into a methodology for building and evaluating CI-RTP/S approaches.

In our empirical study, we show that (1) limiting the training data to the most recent or even only faulty CI test runs often suffices, (2) features on test history work particularly well, and (3) naïve heuristics often outperform complex ML models from prior work. Practitioners can use these empirical guidelines to reduce the amount of effort for selecting and calibrating the best CI-RTP/S approaches for their project. Across studied projects, the approaches chosen by our methodology significantly outperform established RTP baselines. On average, practitioners can thereby expect to save 84% of the testing time while still detecting 90% of the failures when selecting tests. If CI and VCS metadata are available, the methodology is universally applicable, allowing practitioners to comfortably build and calibrate cost-effective RTP and RTS approaches.

ACKNOWLEDGMENTS

We thank Maria Graber, René Dammer, Markus Schnappinger, and the anonymous reviewers who provided helpful feedback to improve this paper. This work was partially funded by IVU Traffic Technologies and the German Federal Ministry of Education and Research (BMBF), grant "SOFIE, 01IS18012A".

REFERENCES

[1] Khaled Walid Al-Sabbagh, Miroslaw Staron, Regina Hebig, and Wilhelm Meding. 2019. Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns. In *Joint Proceedings of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, Vol. 2476. 138–153.

[2] Khaled Walid Al-Sabbagh, Miroslaw Staron, Miroslaw Ochodek, Regina Hebig, and Wilhelm Meding. 2020. Selective Regression Testing based on Big Data: Comparing Feature Extraction Techniques. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*. 322–329. https://doi.org/10.1109/ICSTW50294.2020.00058

[3] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2014. Improving the effectiveness of test suite through mining historical data. In *Processings of the Working Conference on Mining Software Repositories*. 142–151. https://doi.org/10.1145/2597073.2597084

[4] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2015. Striving for Failure: An Industrial Case Study about Test Failure Prediction. In *Proceedings of the International Conference on Software Engineering*. 49–58. https://doi.org/10.1109/ICSE.2015.134

[5] Maral Azizi and Hyunsook Do. 2018. ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development. In *Proceedings of the International Symposium on Software Reliability Engineering*. 144–154. https://doi.org/10.1109/issre.2018.00025

[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the International Conference on Mining Software Repositories*. 447–450. https://doi.org/10.1109/msr.2017.24

[7] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, 1 (2012), 281–305.

[8] Antonia Bertolino, Antonio Guerriero, Roberto Pietrantuono, Stefano Russo, Breno Miranda, and Roberto Pietran-Tuono. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *Proceedings of the International Conference on Software Engineering*. 1–12. https://doi.org/10.1145/3377811.3380369

[9] Leo Breiman and Philip Spector. 1992. Submodel Selection and Evaluation in Regression. The X-Random Case. *International Statistical Review / Revue Internationale de Statistique* 60, 3 (1992), 291–319. https://doi.org/10.2307/1403680

[10] Renée C. Bryce and Charles J. Colbourn. 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology* 48, 10 (2006), 960–970. https://doi.org/10.1016/j.infsof.2006.03.004

[11] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: An industrial case study. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 975–980. https://doi.org/10.1145/2950290.2983954

[12] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression test selection across JVM boundaries. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 809–820. https://doi.org/10.1145/3106237.3106297

[13] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 656–667. https://doi.org/10.1145/3236024.3236053

[14] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research* 7 (2006), 1–30.

[15] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[16] Edward Dunn Ekelund and Emelie Engstrom. 2015. Efficient regression testing based on test history: An industrial evaluation. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 449–457. https://doi.org/10.1109/icsm.2015.7332496

[17] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering*. 329–338. https://doi.org/10.1109/icse.2001.919106

[18] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*. 101–112. https://doi.org/10.1145/347324.348910

[19] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering* 28, 2 (2002), 159–182. https://doi.org/10.1109/32.988497

[20] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. 2004. Selecting a cost-effective test case prioritization technique. *Software Quality Journal* 12, 3 (2004), 185–210. https://doi.org/10.1023/b:sqjo.0000034708.84524.22

[21] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 235–245. https://doi.org/10.1145/2635868.2635910

[22] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Supplemental Material for: "Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration". https://doi.org/10.6084/m9.figshare.13656443

[23] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K. Burke. 2015. Empirical evaluation of Pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the International Symposium on Software Testing and Analysis*. 234–245. https://doi.org/10.1145/2771783.2771788

[24] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 223–233. https://doi.org/10.1109/ICST.2016.33

[25] Kurt F. Fischer. 1977. A test case selection method for the validation of software maintenance modifications. In *Proceedings of International Computer Software and Applications Conference*. 421–426.

[26] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the International Conference on Software Engineering*. 713–716. https://doi.org/10.1109/icse.2015.230

[27] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*. 211–222. https://doi.org/10.1145/2771783.2771784

[28] Mark Harman. 2011. Making the case for MORTO: Multi objective regression test optimization. In *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops*. 111–114.

[29] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *Proceedings of the International Conference on Software Engineering*. 523–534. https://doi.org/10.1145/2884781.2884791

[30] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering* 40, 7 (2014), 650–670. https://doi.org/10.1109/TSE.2014.2327020

[31] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*. 188–197. https://doi.org/10.1109/ICSTW.2013.30

[32] Steffen Herbold. 2020. Autorank: A Python package for automated ranking of classifiers. *Journal of Open Source Software* 5, 48 (2020), 2173–2173. https://doi.org/10.21105/joss.02173

[33] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *Proceedings of the International Conference on Software Engineering*. 483–493. https://doi.org/10.1109/icse.2015.66

[34] Xianhao Jin and Francisco Servant. 2020. A cost-efficient approach to building in continuous integration. In *Proceedings of the International Conference on Software Engineering*. 13–25. https://doi.org/10.1145/3377811.3380437

[35] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie Yan Liu. 2017. LightGBM: A highly efficient gradient boosting decision tree. In *Proceedings of the International Conference on Neural Information Processing Systems*. 3149–3157.

[36] Jung Min Kim and Adam Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering*. 119–129. https://doi.org/10.1145/581339.581357

[37] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Soder, Agneta Nilsson, and Magnus Castell. 2015. Supporting Continuous Integration by Code-Churn Based Test Selection. In *Proceedings of the International Workshop on Rapid Continuous Software Engineering*. 19–25. https://doi.org/10.1109/rcose.2015.11

[38] Jung Hyun Kwon and In Young Ko. 2018. Cost-Effective Regression Testing Using Bloom Filters in Continuous Integration Development Environments. In *Proceedings of the Asia-Pacific Software Engineering Conference*. 160–168. https://doi.org/10.1109/apsec.2017.22

[39] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*. 298–311. https://doi.org/10.1145/3395363.3397364

[40] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. 2012. Prioritizing test cases with string distances. In *Automated Software Engineering*, Vol. 19. 65–95. https://doi.org/10.1007/s10515-011-0093-0

[41] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 583–594. https://doi.org/10.1145/2950290.2950361

[42] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *Proceedings of the International Conference on Automated Software Engineering*. 949–954. https://doi.org/10.1109/ase.2017.8115710

[43] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing Transition-Based Test Selection Algorithms at Google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 101–110. https://doi.org/10.1109/icse-seip.2019.00019

[44] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: Continuous prioritization for continuous integration. In *Proceedings of the International Conference on Software Engineering*. 688–698. https://doi.org/10.1145/3180155.3180213

[45] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[46] Dusica Marijan, Arnaud Gotlieb, and Abhijeet Sapkota. 2020. Neural Network Classification for Improving Continuous Regression Testing. In *Proceedings of the International Conference On Artificial Intelligence Testing*. 123–124. https://doi.org/10.1109/AITEST49225.2020.00025

[47] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for continuous regression testing: An industrial case study. In *Proceedings of the International Conference on Software Maintenance*. 540–543. https://doi.org/10.1109/icsm.2013.91

[48] Dusica Marijan and Marius Liaaen. 2018. Practical selective regression testing with effective redundancy in interleaved tests. In *Proceedings of the International Conference on Software Engineering*. 153–162. https://doi.org/10.1145/3183519.3183532

[49] Toni Mattis and Robert Hirschfeld. 2020. Lightweight Lexical Test Prioritization for Immediate Feedback. *The Art, Science, and Engineering of Programming* 4, 3 (2020), 12:1–12:32. https://doi.org/10.22152/programming-journal.org/2020/4/12

[50] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In *Proceedings of the Conference on Mining Software Repositories*. 385–396. https://doi.org/10.1145/3379597.3387458

[51] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. https://doi.org/10.5281/zenodo.3610998

[52] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 233–242. https://doi.org/10.1109/icse-seip.2017.16

[53] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. FAST approaches to scalable similarity-based test case prioritization. In *Proceedings of the International Conference on Software Engineering*. 222–232. https://doi.org/10.1145/3180155.3180210

[54] Armin Najafi, Weiyi Shang, and Peter C. Rigby. 2019. Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 213–222. https://doi.org/10.1109/icse-seip.2019.00031

[55] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. 2011. Regression testing in the presence of non-code changes. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*. 21–30. https://doi.org/10.1109/icst.2011.60

[56] Tanzeem Bin Noor and Hadi Hemmati. 2016. A similarity-based approach for test case prioritization using historical failure data. In *Proceedings of the International Symposium on Software Reliability Engineering*. 58–68. https://doi.org/10.1109/issre.2015.7381799

[57] Tanzeem Bin Noor and Hadi Hemmati. 2017. Studying test case failure prediction for test case prioritization. In *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering*. 2–11. https://doi.org/10.1145/3127005.3127006

[58] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 241–251. https://doi.org/10.1145/1029894.1029928

[59] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[60] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*. 324–336. https://doi.org/10.1145/3395363.3397383

[61] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. 2013. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 26–36. https://doi.org/10.1145/2491411.2491436

[62] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagppan. 2019. FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services. In *Proceedings of the International Conference on Software Engineering*. 408–418. https://doi.org/10.1109/icse.2019.00054

[63] John Platt. 1999. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers* 10, 3 (1999), 61–74.

[64] Gregg Rothermel and Mary Jean Harrold. 1994. A Framework for Evaluating Regression Test Selection Techniques. In *Proceedings of the International Conference on Software Engineering*. 201–210. https://doi.org/10.1109/ICSE.1994.296779

[65] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262

[66] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. 2000. Regression test selection for C++ software. *Software Testing, Verification and Reliability* 10, 2 (2000), 77–109.

[67] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the International Conference on Software Engineering*. 268–279. https://doi.org/10.1109/icse.2015.47

[68] Mark Sherriff, Mike Lake, and Laurie Williams. 2007. Prioritization of regression tests using singular value decomposition with empirical change records. In *Proceedings of the International Symposium on Software Reliability Engineering*. 81–90. https://doi.org/10.1109/issre.2007.25

[69] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *Proceedings of the International Symposium on Software Testing and Analysis*. 84–94. https://doi.org/10.1145/3213846.3213875

[70] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *Proceedings of the International Symposium on Software Reliability Engineering*. 228–238. https://doi.org/10.1109/issre.2019.00031

[71] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 12–22. https://doi.org/10.1145/3092703.3092709

[72] Robert White, Jens Krinke, and Raymond Tan. 2020. Establishing Multilevel Test-to-Code Traceability Links. In *Proceedings of the International Conference on Software Engineering*. 861–872. https://doi.org/10.1145/3377811.3380921

[73] Rüdiger Wirth and Jochen Hipp. 2000. CRISP-DM : Towards a Standard Process Model for Data Mining. In *Proceedings of the International Conference on the Practical Application of Knowledge Discovery and Data Mining*. 29–39.

[74] Shin Yoo and Mark Harman. 2007. Pareto efficient multi-objective test case selection. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM Press, 140–150. https://doi.org/10.1145/1273463.1273483

[75] Shin Yoo and Mark Harman. 2010. Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software* 83, 4 (2010), 689–701. https://doi.org/10.1016/j.jss.2009.11.706

[76] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

[77] Shin Yoo, Robert Nilsson, and Mark Harman. 2011. Faster Fault Finding at Google Using Multi Objective Regression Test Optimisation. In *Proceedings of the International Symposium on the Foundations of Software Engineering*.

[78] Tingting Yu and Ting Wang. 2018. A Study of Regression Test Selection in Continuous Integration Environments. In *Proceedings of the International Symposium on Software Reliability Engineering*. 135–143. https://doi.org/10.1109/ISSRE.2018.00024

[79] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the International Conference on Software Engineering*. 199–209. https://doi.org/10.1145/3180155.3180198

[80] Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. 2019. TestSage: Regression test selection for large-scale web service testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 430–440. https://doi.org/10.1109/icst.2019.00052

[81] Yuecai Zhu, Emad Shihab, and Peter C. Rigby. 2018. Test re-prioritization in continuous testing environments. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 69–79. https://doi.org/10.1109/icsme.2018.00016

[82] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 91–100. https://doi.org/10.1145/1595696.1595713

A.3.4. Challenges in Regression Test Selection for End-to-End Testing of Microservice-based Software Systems

 Daniel Elsner, Daniel Bertagnolli, Alexander Pretschner, Rudi Klaus, Challenges in Regression Test Selection for End-to-End Testing of Microservice-Based Software Systems, AST '22: Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test, pages 1–5, July 2022.

In the following, the complete paper is included in its published form in accordance with the ACM author rights, DOI: 10.1145/3524481.3527217.

Challenges in Regression Test Selection for End-to-End Testing of Microservice-based Software Systems

Daniel Elsner
Daniel Bertagnolli
Alexander Pretschner
firstname.lastname@tum.de
Technical University of Munich
Munich, Germany

Rudi Klaus
rudi.klaus@t-systems.com
T-Systems International
Munich, Germany

ABSTRACT

Dynamic regression test selection (RTS) techniques aim to minimize testing efforts by selecting tests using per-test execution traces. However, most existing RTS techniques are not applicable to microservice-based, or, more generally, distributed systems, as the dynamic program analysis is typically limited to a single system. In this paper, we describe our distributed RTS approach, `microRTS`, which targets automated and manual end-to-end testing in microservice-based software systems. We employ `microRTS` in a case study on a set of 20 manual end-to-end test cases across 12 versions of the German COVID-19 contact tracing application, a modern microservice-based software system. The results indicate that initially `microRTS` selects all manual test cases for each version. Yet, through semi-automated filtering of test traces, we are able to effectively reduce the testing effort by 10–50%. In contrast with prior results on automated unit tests, we find method-level granularity of per-test execution traces to be more suitable than class-level for manual end-to-end testing.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Software testing, regression test selection, microservice architectures, end-to-end testing, manual testing

ACM Reference Format:
Daniel Elsner, Daniel Bertagnolli, Alexander Pretschner, and Rudi Klaus. 2022. Challenges in Regression Test Selection for End-to-End Testing of Microservice-based Software Systems. In *IEEE/ACM 3rd International Conference on Automation of Software Test (AST '22), May 17–18, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524481.3527217

AST '22, May 17–18, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9286-0/22/05…$15.00
https://doi.org/10.1145/3524481.3527217

1 INTRODUCTION

Regression testing is a software testing activity that is regularly performed to ascertain that changes have not inadvertently altered previous system behavior [15]. Yet, with increasingly large test suites and shorter software (delivery) life-cycles, running all tests after each change is often too costly [4, 30]. To reduce the costs of regression testing, regression test selection (RTS) [7, 13, 21–23, 31, 33] techniques have been extensively studied since the 1970s [5].

Traditional RTS techniques identify a subset of tests by comparing new code changes with per-test code dependencies. Collecting these per-test code dependencies can either be achieved through *static* or *dynamic* program analysis at the level of basic-blocks [11, 21], functions/methods [6, 17, 32, 33], classes/files [7, 8, 12, 13], modules [24], or combinations thereof [25, 28, 31]. In dynamic RTS techniques, these per-test code dependencies can also be interpreted as per-test execution traces. Since most techniques are incapable of collecting dependencies across system boundaries, they mainly target unit testing. Yet, in microservice-based systems, or, more generally, highly distributed systems, checking for functional correctness is no longer limited to individual units or modules, but especially requires anticipation of interface and interaction bugs [16, 33]. Hence, existing RTS techniques are not directly applicable, as these systems need to be tested through integration or end-to-end tests that operate across service or system boundaries.

While there are a few studies on RTS for web applications and services [17, 19, 33], they have several limitations in the context of microservices: First, existing RTS techniques targeting web applications only instrument server code for tracing tests [17, 19]. However, excluding client code is problematic in microservice-based systems, where (rich) web or mobile client applications often contain large parts of the business logic and orchestrate calls to different microservices. Second, even though protocol- and language-agnostic distributed tracing approaches are provided by observability frameworks such as `OpenTelemetry`[1], existing RTS techniques are implemented for monolingual systems [33] and specific communication protocols, such as Hypertext Transfer Protocol (HTTP) [17, 19]. Third, high instrumentation overhead of RTS techniques may preclude their applicability in practice [2, 3, 18], but existing studies lack analyses on the introduced instrumentation overhead during service startup and test execution. Finally, due to the complexity of automating end-to-end tests, these tests are often performed manually [10, 19]. Yet, no prior RTS study investigates RTS for manual end-to-end tests in microservice-based systems.

[1] `OpenTelemetry`: https://opentelemetry.io/

In this paper, we propose microRTS, a distributed RTS technique suitable for manual or automated end-to-end testing in microservice-based software systems. microRTS is implemented on top of well-established distributed tracing infrastructure and Java bytecode manipulation libraries, to enable automated instrumentation of arbitrary Java microservices at runtime. We evaluate the influence of several implementation aspects on the instrumentation overhead in the open-source microservice benchmark application TeaStore [29].

We further present a case study that was conducted together with our industry partner T-Systems[2] on a subset of the manual end-to-end test suite on 12 software versions of the German COVID-19 contact tracing application, Corona-Warn-App (CWA), a modern microservice-based software system. Since the CWA accesses backend microservices through a rich mobile client (we only consider Android), we design microRTS to collect per-test execution traces from mobile clients as well as microservices, leading to more complete traces. In contrast with prior results on automated unit tests [7, 12], we find class-level granularity of test traces to be too coarse grained for RTS of manual end-to-end tests in this context, essentially leading to *retest-all*. With traces at method-level granularity, microRTS initially still selects all test cases across the 12 studied versions. However, after closer inspection of the reasons behind a test being selected, we find that the manual tests are commonly imprecisely specified. For instance, while all manual tests cover the CWA start screen, most of them do not test any start screen functionality. Consequently, by semi-automatically filtering out irrelevant parts of the test traces, microRTS can exclude 10–50% of tests.

To foster more research on RTS for integration or end-to-end testing in microservice-based systems, we discuss challenges and elaborate on experiences when implementing and applying microRTS in a real-world context.

2 RELATED WORK & STATE-OF-PRACTICE

Among the many existing RTS studies already referenced, we consider the following to be most relevant for the context of this work:

Nakagawa et al. [19] propose a method-level RTS technique for manual end-to-end tests for Java web applications. By sending a custom header with each HTTP request to the server, a tester's browser can be mapped to accessed methods, assuming a one-to-one mapping of HTTP requests and Java Virtual Machine (JVM) threads. The results of the industrial case study on two web applications indicate that it is likely that all tests need to be executed for large modifications or changes to common code parts. As the proposed RTS technique is not able to trace more than one web server, it is yet unsuitable in a microservice context.

Long et al. [17] propose the RTS tool WebRTS, which supports collecting file-level per-test execution traces across multiple instances of a web server. It is designed for Java web applications with server-side page rendering and is limited to communication using the HTTP protocol which confines the applicability to a small subset of microservice-based systems. Unfortunately, although publicly available, WebRTS lacks an adequate user documentation and further relies on JVM bytecode instrumentation from an external library that lacks English documentation[3]. For this reason, we were neither able to fully comprehend, nor to apply their tool in preliminary experiments on Java microservices.

Zhong et al. [33] propose the RTS technique TestSage that targets web service testing at Google. TestSage supports C++ services and performs function-level instrumentation using XRay[4]. While TestSage reduces the testing time by 34%, it does not support parallel test tracing and is limited to homogeneous C++ web services.

In summary, we are not aware of any prior work that investigates the potential and challenges of RTS in the context of manual end-to-end testing for microservice-based systems.

3 DISTRIBUTED TEST SELECTION

When a microservice-based software system is tested in an end-to-end fashion, restarting the system under test (SUT) between each test case is not always feasible. This is because such systems often involve tens or even hundreds of services. Hence the startup process is expensive and time-consuming [33]. Consequently, when collecting per-test execution traces, we need to take into account that multiple tests are executed on the same deployed service instance, either sequentially or in parallel. We thus require segmentation of collected traces according to the tests' execution time frame and link covered code parts to the test that executed them [17]. In the following, we describe how microRTS collects precise distributed per-test execution traces for end-to-end tests and performs change-based test selection.

While microRTS currently supports microservices written in languages that target the JVM (e.g., Java, Kotlin) and instrumentation of Android mobile clients, the concepts are agnostic to the actually used programming language or platform. We chose Java as our case study subject is written in Java and Kotlin (see Sec. 5).

3.1 Distributed Tracing

The core principle behind *distributed tracing* is *context propagation*: A *context* contains (at least) a unique identifier that identifies a *trace* and is transferred in and across services in a distributed system [27]. The trace thereby encapsulates all requests related to an individual transaction. To enable context propagation, clients and services are instrumented to be able to create, transfer, and access context information embedded into requests. Consider Fig. 1 that depicts an instrumented service, where context information is extracted from inbound requests and injected into outbound requests. Furthermore, *trace points* can be inserted into the instrumented service that define actions such as attaching metadata to the context.

The implementation of the required code instrumentation for the middleware (e.g., HTTP client libraries) can be performed using well-established, polyglot distributed tracing and observability frameworks, such as OpenTracing[5] or OpenTelemetry. microRTS uses OpenTelemetry to automatically instrument Java microservices by attaching a Java Agent [1] that performs Java bytecode instrumentation at runtime. Thereby, the instrumented microservice will extract the context from inbound requests and we can link the context with custom code instrumentation as described next.

[2] T-Systems is one of the largest European information technology service providers: https://www.t-systems.com/

[3] JVM-SANDBOX: https://github.com/alibaba/jvm-sandbox

[4] LLVM XRay: https://llvm.org/docs/XRay.html

[5] OpenTracing: https://opentracing.io/

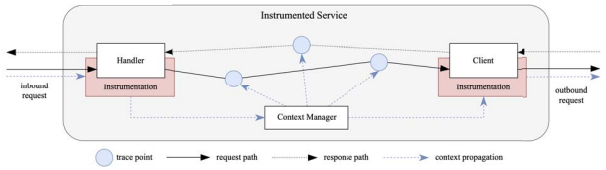


Figure 1: Context propagation in an instrumented service (inspired by Shkuro [26])

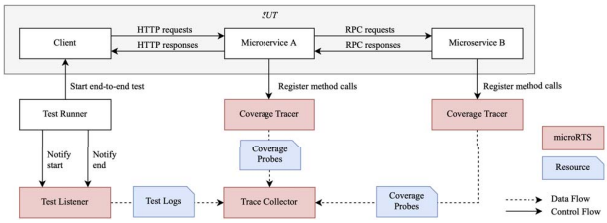


Figure 2: Overview of the `microRTS` test tracing architecture

3.2 Code Instrumentation

We have described why the SUT typically cannot be restarted after each test in microservice-based systems. Therefore, if several tests are executed one after another, we need a code instrumentation that also takes into account already created objects from previously executed tests [17]. Thus, similar to pre-existing RTS techniques [19, 33], `microRTS` instruments microservices (and clients) at method-level granularity rather than class-level, as only instrumenting JVM class loading or object creation would miss if tests call methods of already existing objects. However, since we aim to investigate benefits of test trace granularity more closely (see Sec. 5), `microRTS` offers to control if test traces are stored (and aggregated) at method- or class-level granularity. Furthermore, existing approaches for collecting test traces differ regarding the strategy to export information about covered methods (i.e., *coverage probes*) during runtime [14, 20]. `microRTS` offers to export coverage probes directly after they fire or in batches, and supports writing coverage probes into a file or sending them via Transmission Control Protocol (TCP) sockets to a central *trace collector*.

We implement the method-level code instrumentation using a Java Agent and the `ByteBuddy` bytecode manipulation library[6]. Thereby, whenever a method is entered, the instrumentation stores a coverage probe in the *coverage tracer*. A coverage probe contains the method's signature, the name of the surrounding class, and the current context's trace identifier. Depending on how `microRTS` is configured, the coverage probes are written into a file or sent via TCP to the trace collector, either one-by-one or in batches. Fig. 2 illustrates how coverage information is collected from instrumented microservices. Additionally, the client is connected to a *test listener* that is responsible for maintaining the context for test cases in *test logs*, to later on link coverage probes to test cases. In Sec. 5, we describe how we implemented a test listener for the Android client of the CWA. `microRTS` further implements compile-time instrumentation of Android mobile clients using the `AndroidBuddy` library[7], as in contrast to the JVM, Android does not allow runtime instrumentation.

3.3 Test Selection

For change-based test selection, `microRTS` uses the changeset since the last time a test suite was executed from the version control system (VCS), together with the collected method-level test traces from the last test execution. `microRTS` then parses the `.java` and `.kt` files from the changeset and computes (1) the set of changed classes and (2) the set of changed methods by comparing checksums of each method's source code, similar to existing RTS techniques [31]. Using these two sets and the test traces, `microRTS` then determines affected tests through class- or method-level selection. In Sec. 5, we describe the effects of using class- and method-level selection in a manual end-to-end testing context.

[6] `ByteBuddy`: https://bytebuddy.net/
[7] `AndroidBuddy`: https://github.com/LikeTheSalad/android-buddy

4 EFFICIENCY EVALUATION

In order to analyze the efficiency of `microRTS`'s instrumentation, we conduct experiments on the TeaStore microservice benchmark application. We thereby strive to answer the following research questions (RQs):

- **RQ**$_1$: How much instrumentation overhead does `microRTS` introduce at system startup and during testing?
- **RQ**$_2$: How does the granularity of coverage probes and their export strategy affect instrumentation overhead?

4.1 Experimental Setup

TeaStore is an open-source microservice reference application used by researchers to analyze and test novel techniques for microservice-based systems [9, 29]. We conduct our experiments on version *v1.4.0* of the TeaStore, consisting of 6 Java microservices that we orchestrate using `Docker-Compose`. Since TeaStore currently only contains unit tests, we implemented a set of 23 automated end-to-end tests using the testing framework `Cypress`. Our test suite covers each feature of the application by at least one test case and was merged by the project's maintainers into the main code base[8].

To execute our experiments, we instrument each microservice and the `Cypress` test runner with `microRTS` and run all 23 test cases with different (1) coverage probe collection granularity (method- or class-level) and (2) coverage probe export strategies (in-memory, file, or socket export). We repeat the experiments 30 times to account for variations in the runtimes and measure the average system startup and testing runtime for all configurations of `microRTS` and compare them to executions without any instrumentation (`NoInst`).

4.2 Discussion of Results

RQ$_1$: Startup and Runtime Instrumentation Overhead. The results of the comparative analysis between `microRTS` and `NoInst` show that the performance impact of `microRTS` is more significant during services' startup (+67.5%) than during testing (+18%). By re-running the tests using *only* `OpenTelemetry`'s instrumentation, we see that most of the overhead stems from `OpenTelemetry`, which already adds 40% to the startup and 10.5% to the testing runtime.

[8] TeaStore Pull Request: https://github.com/DescartesResearch/TeaStore/pull/203

We further observe that the overhead caused by microRTS is significantly higher during the first test compared to the mean overhead. The reason is inherent to dynamic bytecode instrumentation, which transforms Java bytecode files on the fly when they are first loaded by the JVM ClassLoader.

RQ_2: Granularity and Export Strategies of Coverage Probes. The granularity at which microRTS is configured to export coverage probes does not significantly affect the instrumentation overhead: method-level granularity adds roughly 1.2% overhead compared to class-level granularity in total test suite execution time. The reason why storing and exporting coarser-grained class-level coverage probes is not far more efficient is that microRTS still needs to instrument all methods as explained in Sec. 3.2.

Regarding the chosen coverage probe export strategy, we find that in-memory is the fastest strategy because it does not export probes until service shutdown. Perhaps surprisingly, the file export strategy only adds around 1% of runtime overhead when compared to in-memory, despite the I/O overhead. Finally, although used in prior studies [20, 33], the socket strategy has a comparatively high runtime overhead of 6.7% compared to in-memory.

5 CASE STUDY: CORONA-WARN-APP

To evaluate microRTS in a real-world microservice-based system, we conduct a case study on the manual end-to-end test suite of the Corona-Warn-App (CWA) to answer **RQ_3**: How much manual end-to-end testing effort reduction can be achieved using microRTS?

5.1 Experimental Setup

The CWA is the official German Covid-19 contact tracing application, based on a decentralized, microservices architecture. The source code of the microservices and mobile clients is open-sourced and available on Github[9], easing reproducibility and extension of our case study. Service providers such as our industry partner T-Systems are responsible for end-to-end regression testing of new versions and releases of the CWA. Therefore, they use a manual regression test suite that is not publicly available. As currently the test cases for release testing are selected manually, automated and systematic tool-support through microRTS can be beneficial.

For our experiments with microRTS, we prepare a suitable testing environment: We (1) instrument seven CWA microservices, (2) instrument the mobile client (only Android), (3) patch or mock requests to external services such as Google's Exposure Notification System, as they can exclusively be used by authorized official health agencies, and (4) orchestrate all instrumented services with Docker-Compose, as neither the staging, nor the production environment configuration are publicly available. We then execute 20 manual end-to-end test cases provided by T-Systems for version *2.5.1* of the CWA. These tests are still executable without limitations in our experimental setup and we only instrument the seven services required for the provided test cases. We implement a small Android sidecar application, where we can start and stop a manual test case, which internally initializes and closes a tracing context.

To evaluate the potential of microRTS, we determine the set of selected tests on all (12) CWA release candidate versions between *2.5.1* and *2.6.1*. We include these release candidates to gain insights on how shorter testing cycles affect RTS results. Furthermore, we compare class- and method-level RTS as their effectiveness is not unequivocal in existing RTS literature [7, 17, 19].

[9]Corona-Warn-App (CWA): https://github.com/corona-warn-app

5.2 Discussion of Results

RQ_3: Test Effort Reduction. The initial results show that RTS at method- and class-level already selects all 20 tests for the first version, namely between *2.5.1* and *2.6.0-RC0*. As a result, RTS between *2.5.1* and all other subsequent versions has the same outcome.

To understand the underlying reasons, we investigate the causes for selection: First, 100% of the test selections for all versions have been caused by changes in the Android client, both using class- and method-level RTS. This highlights the importance of instrumenting client code as well. Second, all test cases include various shared covered methods, which originate primarily from the home screen where all tests start or end according to the test case specifications. Yet, surprisingly, only 9 out of the 20 test cases effectively verify functionality of the home screen. To determine if the other 11 test cases would have been selected even if their traces started on their respective sub-page, we proceed by refining the test traces. During this refinement step, we remove all covered methods that are associated to the home screen for the 11 test cases. Using class-level RTS nothing changes: all tests are selected. However, when using method-level RTS the selected tests for *2.6.0-RC0* are reduced by 50%, only 10 out of 20 tests are selected; for the subsequent versions and the next release *2.6.1*, up to 18 out of 20 tests are selected (90%).

Hence, we can conclude that the effectiveness of our RTS approach for manual end-to-end testing is highly dependent on the precision of test specifications. Through semi-automated pruning of test traces using domain knowledge, we are able to exclude up to 50% of tests, but only when using fine-grained method-level RTS. Our results confirm findings from prior RTS research on manual testing, where RTS effectiveness was limited with shared covered code parts in test traces or with large changesets [19].

6 CONCLUSION

In this paper, we introduce microRTS, a dynamic RTS technique for microservice-based systems. By combining established distributed tracing infrastructure with code instrumentation, microRTS collects per-test execution traces at method- or class-level across services and clients, thereby enabling test selection for automated or manual end-to-end tests. We further present a case study on RTS for manual end-to-end tests in the CWA, a real-world microservice-based system. Our initial results show that if manual tests are specified rather coarse-grained, microRTS can not provide any benefits over retest-all. However, when pruning per-test execution traces using domain knowledge, we are able to exclude up to 50% of tests. These findings confirm prior research on manual testing and show that manual end-to-end testing of microservice-based systems is particularly intricate to optimize.

ACKNOWLEDGMENTS

This work was funded by the German Federal Ministry of Education and Research (BMBF), grant SOFIE 01IS18012B. The responsibility for this article lies with the authors.

REFERENCES

[1] 2017. Java Agent API. https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/package-summary.html

[2] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. 235–245. https://doi.org/10.1145/2635868.2635910

[3] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *Proceedings of the International Symposium on Software Testing and Analysis*. 491–504. https://doi.org/10.1145/3460319.3464834

[4] Kurt Fischer, Farzad Raji, and Andrew Chruscicki. 1981. A Methodology for Retesting Modified Software. In *Proceedings of the National Telecommunications Conference*. 1–6.

[5] Kurt F. Fischer. 1977. A test case selection method for the validation of software maintenance modifications. In *Proceedings of International Computer Software and Applications Conference*. 421–426.

[6] Ben Fu, Sasa Misailovic, and Milos Gligoric. 2019. Resurgence of Regression Test Selection for C++. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 323–334. https://doi.org/10.1109/ICST.2019.00039

[7] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the International Conference on Software Engineering*. 713–716. https://doi.org/10.1109/icse.2015.230

[8] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*. 211–222. https://doi.org/10.1145/2771783.2771784

[9] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. 2021. SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications. In *Proceedings of the International Conference on Performance Engineering*. 165–176. https://doi.org/10.1145/3427921.3450248

[10] Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. 2021. How can manual testing processes be optimized? Developer survey, optimization guidelines, and case studies. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1281–1291. https://doi.org/10.1145/3468264.3473922

[11] Mary Jean Harrold, Alessandro Orso, James A. Jones, Tongyu Li, Maikel Pennings, Saurabh Sinha, Ashish Gujarathi, Donglin Liang, and S. Alexander Spoon. 2001. Regression test selection for Java software. *ACM SIGPLAN Notices* 36, 11 (2001), 312–326. https://doi.org/10.1145/504311.504305

[12] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 583–594. https://doi.org/10.1145/2950290.2950361

[13] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *Proceedings of the International Conference on Automated Software Engineering*. 949–954. https://doi.org/10.1109/ase.2017.8115710

[14] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing Transition-Based Test Selection Algorithms at Google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 101–110. https://doi.org/10.1109/icse-seip.2019.00019

[15] Hareton K.N. Leung and Lee White. 1989. Insights into regression testing. In *Proceedings of the International Conference on Software Maintenance*. 60–69.

[16] Hareton K.N. Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Press, Silver Spring, MD, 290–301. https://doi.org/10.1109/icsm.1990.131377

[17] Zhenyue Long, Zeliu Ao, Guoquan Wu, Wei Chen, and Jun Wei. 2020. WebRTS: A Dynamic Regression Test Selection Tool for Java Web Applications. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 822–825. https://doi.org/10.1109/ICSME46990.2020.00102

[18] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[19] Takao Nakagawa, Kazuki Munakata, and Koji Yamamoto. 2019. 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*. 1–6. https://doi.org/10.1109/ISSREW.2019.00033

[20] Raphael Noemmer and Roman Haas. 2020. An Evaluation of Test Suite Minimization Techniques. In *Software Quality: Quality Intelligence in Software and Systems Engineering*, D Winkler, S Biffl, D Mendez, and J Bergsmann (Eds.). Vol. 371. Springer,, 51–66. https://doi.org/10.1007/978-3-030-35510-4_4

[21] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 241–251. https://doi.org/10.1145/1029894.1029928

[22] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262

[23] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. 2000. Regression test selection for C++ software. *Software Testing, Verification and Reliability* 10, 2 (2000), 77–109. https://doi.org/10.1002/1099-1689(200006)10:2<77::AID-STVR197>3.0.CO;2-E

[24] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-Level Regression Testing. In *Proceedings of the International Conference on Software Engineering*. 689–699. https://doi.org/10.1109/ICSE.2017.69

[25] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *Proceedings of the International Symposium on Software Reliability Engineering*. 228–238. https://doi.org/10.1109/issre.2019.00031

[26] Yuri Shkuro. 2019. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd.

[27] Benjamin H Sigelman, Luiz Andr, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. https://doi.org/dapper-2010-1

[28] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-level vs. module-level regression test selection for .NET. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 848–853. https://doi.org/10.1145/3106237.3117763

[29] Joakim Von Kistowski, Simon Eismann, Norbert Schmitt, Andre Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A micro-service reference application for benchmarking, modeling and resource management research. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 223–236. https://doi.org/10.1109/MASCOTS.2018.00030

[30] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

[31] Lingming Zhang. 2018. Hybrid regression test selection. In *Proceedings of the International Conference on Software Engineering*. 199–209. https://doi.org/10.1145/3180155.3180198

[32] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2013. FaultTracer: A spectrum-based approach to localizing failure-inducing program edits. *Journal of Software: Evolution and Process* 25, 12 (2013), 1357–1383. https://doi.org/10.1002/smr.1634

[33] Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. 2019. TestSage: Regression test selection for large-scale web service testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. 430–440. https://doi.org/10.1109/icst.2019.00052

A.3.5. How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies

 Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, Sven Apel, How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies, Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1281–1291, August 2021.

In the following, the complete paper is included in its published form in accordance with the ACM author rights, DOI: 10.1145/3468264.3473922.

How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies

Roman Haas*
Saarbrücken Graduate School of Computer Science, CQSE
Saarland, Munich, Germany

Daniel Elsner*
Technical University of Munich
Munich, Germany

Elmar Juergens
CQSE
Munich, Germany

Alexander Pretschner
Technical University of Munich
Munich, Germany

Sven Apel
Saarland University,
Saarland Informatics Campus
Saarland, Germany

ABSTRACT

Manual software testing is tedious and costly as it involves significant human effort. Yet, it is still widely applied in industry and will be in the foreseeable future. Although there is arguably a great need for optimization of manual testing processes, research focuses mostly on optimization techniques for automated tests. Accordingly, there is no precise understanding of the practices and processes of manual testing in industry nor about pitfalls and optimization potential that is untapped. To shed light on this issue, we conducted a survey among 38 testing professionals from 16 companies, to investigate their manual testing processes and to identify potential for optimization. We synthesize guidelines when optimization techniques from automated testing can be implemented for manual testing. By means of case studies on two industrial software projects, we show that fault detection likelihood, test feedback time and test creation efforts can be improved when following our guidelines.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Software testing, manual testing, test optimization

ACM Reference Format:
Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. 2021. How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3473922

*Both authors contributed equally to this research.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473922

1 INTRODUCTION

Manual software testing is tedious, costly, and involves significant human effort. Yet, according to a recent survey, it is still widely applied in industry [1]. Despite the availability of advanced test automation techniques, previous research reports that manual software testing often complements automated testing [10, 26]. In fact, manual testing strategies can arguably detect other software faults than automated strategies [4]. Depending on the project's context, automation might be too costly [37], too complex [36], or even impossible [35], 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 [18, 38]. There has been significant research effort on optimizing automated testing, for example, on regression test optimization [6–8, 12, 14, 15, 25, 28, 31]. Still, only few research efforts attempt to transfer techniques such as regression test selection [5, 30], regression test prioritization [17, 22], or failure prediction [18] to *manual* software testing. Transfer is hindered, among other things, by missing required data (e.g., unavailability of code coverage information [17]): 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 [17, 18].

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 manual software testing process model*, 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 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 9 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.

The survey results, analyses, and the optimization guidelines are publicly available in our supplemental repository[1].

2 RELATED WORK

Studies from 2011 and 2013 on the state of software testing practice report that more than 90% of survey participants test their software manually [9, 11]. 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 [35, 36] or deliberately employed [10, 26].

Test optimization is widespread in automated testing [6–8, 12, 14, 15, 25, 28, 31], but techniques are often not transferable to manual testing due to missing data (e.g., code coverage information) [18] or unsuitable testing processes (e.g., only black-box access during testing) [17]. Despite these difficulties, several researchers have applied techniques to optimize different aspects of manual testing [2, 5, 17, 18, 21, 22, 30]. For example, Hemmati et al. [17] 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 1. The optimization techniques are later consolidated with findings from our empirical study in Sec. 3.4 to provide a holistic view on manual test optimization.

Table 1: Prerequisites and caveats of existing techniques to optimize manual testing

Ref.	Prerequisites	Caveats
	1. Test Prioritization	
[17]	Textual test descriptions, test failure history	Less effective in traditional development approaches
[22]	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 Selection	
[21]	Test traces, familiarity of testers with code base	Under-specification of tests leads to unstable traces
[5]	Textual test descriptions, static code analysis tool, program profiler	Accuracy of static analysis low (90%)
[30]	Test traces, adjustment of system to separate traces for parallel testing	Unsuitable in the case of large or frequent changes
	3. Test Gap Analysis	
[3]	Test traces, version control data	Up-to-date test traces are costly, data granularity is critical
	4. Test Case Reduction	
[18]	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. Refactoring	
[2]	Textual test descriptions, individual test steps, expert labels for test suites	Varying effectiveness depending on the test suite
	6. Test Quality Monitoring	
[16]	Textual test descriptions	Parameterization requires experience

Test Prioritization. Hemmati et al. [17] were the first to study regression test prioritization 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. [22] 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 Selection. Juergens et al. [21] report on an industrial case study that demonstrates challenges of applying test selection to manual system tests based on method-level test traces. They suggest to use a semi-automated process in practice, 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. [5] propose an approach for regression test 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

[1] https://github.com/manual-testing-study/manual-testing-esec-fse-21/

limits its transferability. In a case study on manual end-to-end testing of legacy Web applications, Nakagawa et al. [30] show that their simple test 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. [3] describe a semi-automated approach for manual regression test selection. Although their selection technique reveals deficiencies in effectiveness, it provides practical benefits for test gap analysis, 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. [18] 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, 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 [38].

Refactoring. Bernard et al. [2] aim at improving tool support for refactoring manual tests to 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. [16] study how 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 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 [11].

3 DEVELOPER SURVEY AND GUIDELINES

In this section, we provide details of our semi-structured online interview, following the suggestions of Jedlitschka et al. [20], and we derive a set of guidelines for optimization of manual testing processes that synthesize our findings.

3.1 Research Areas and Questions

With our interviews, we target several research questions (RQs) from three research areas (RAs): 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 [27]. 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 2, 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 Participants

In August 2020, we contacted 115 test engineers, testers, developers, test architects, test leads, and test managers of industry partners. $N = 38$ 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.

3.3 Questionnaire and Conduct

We designed a questionnaire to address the above research questions. In Table 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.

Table 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?	📄

3.4 Results

To analyze the answers of the survey, we used an open card-sorting technique [19]. 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? Fig. 1.1 and 1.2 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 under-specification 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.

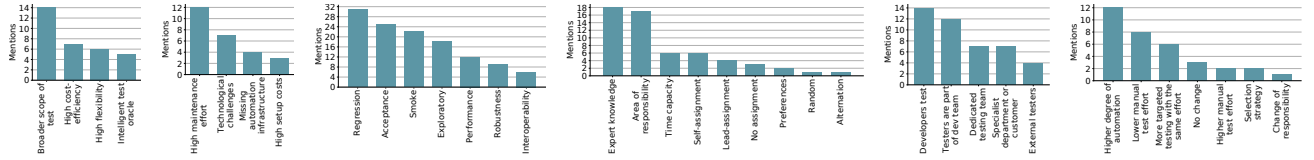
Regarding technological and organizational challenges that hinder test automation, we find the following obstacles to be prevalent (number of mentions in parentheses): Lack of time (8), lack of budget (6), limited know-how (6), limited technology (6), for example unstable testing tools or tedious creation 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? Fig. 1.3 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? Fig. 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



1.1: Advantages of manual testing 1.2: Drawbacks of automated tests 1.3: Test activities 1.4: Assignment criteria for test cases 1.5: Manual testing organization 1.6: Expected evolution of the manual testing process

Figure 1: Charts for survey answers

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.

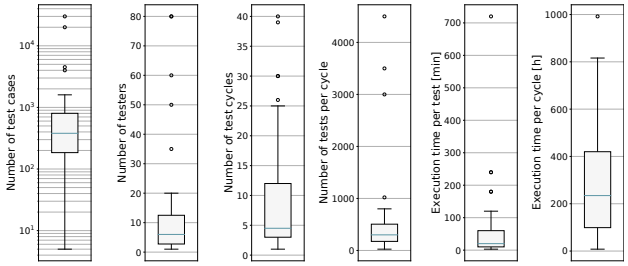


Figure 2: Distributions of manual test process characteristics

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.

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

RQ$_{2.3}$: 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).

Fig. 1.4 shows how test cases are assigned to testers, where tester experience (18) and areas of responsibility (17) dominate other assignment criteria.

RQ$_{2.4}$: 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 (HIL) (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 concurrently are used by multiple testers. Frequently, there are issues because of interactions of the system under test (SUT) with other systems or applications (15), and remote test environments (12). Furthermore, different hardware combinations (4), legacy technologies (2), several test environments (2), HIL tests (1) and network latency (1) were highlighted as technology-related challenges.

Fig. 1.5 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 [23, 27]. They are commonly first detected if a previously passing test, that is clearly unrelated to code changes introduced to the system, suddenly fails [23, 24]. 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)—5 participants are not sure whether there are flaky tests in their test suite. Only 5 participants answered the question about how they deal with flaky tests: 2 participants re-run tests that are deemed flaky. 3 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

$RQ_{3.1}$: 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. Fig. 1.6 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 SUT 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 3 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 2, 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 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 SUT (14), creating and loading test data (10), and setting up the SUT (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 *re-using 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 [29], the most straight-forward 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 lists the optimization levers that we identify, 3 derived optimization techniques with associated prerequisites and caveats. Together with Table 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.*

3.5 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 1 and 3).

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

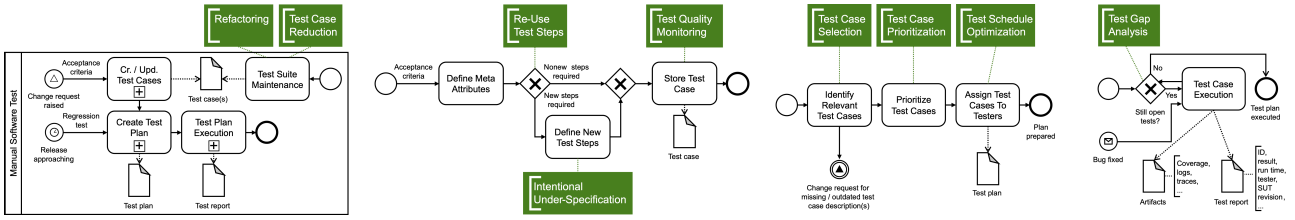
Table 3: Prerequisites and caveats of derived optimization techniques based on identified existing levers in practice (extension of Table 1)

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

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 current bottlenecks in their process. Using Tables 1 and 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 Fig. 3.1, this triggers the sub-process *Create Test Plan*, which is unrolled in Fig. 3.3, 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 1 collects prerequisites and caveats of three test 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 Fig. 3.1 labelled with ⊞ are sub-processes, which are explained in more detail in the following paragraphs and figures. The optimization techniques *Refactoring* and *Test Case Reduction* can be applied most easily during test suite maintenance.

Test Case Creation. Fig. 3.2 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



3.1: Manual software test process 3.2: Test case creation 3.3: Test plan creation 3.4: Test plan execution

Figure 3: Optimization potentials (green) in the empirical manual testing process

applied. That is, the test can be defined generically such that several cases are covered. For example, if there are mutliple 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. Fig. 3.3 shows the *Test Plan Creation* sub-process. Initially, the set of relevant 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. Fig. 3.4 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.

4 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 3.5 to identify applicable optimization guidelines. 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 4.

4.1 Case Study 1: User Acceptance Testing

Background Information. Our first study subject is owned by Munich Re[2], 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 5 testers did manual user acceptance testing (UAT). There are approximately 7 releases per year, each release has a pre-defined 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[3] is used to manage test cases and results.

Applicability of Optimization Techniques Following our process model in Fig. 3.1, we were able to suggest 5 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. 3.2) 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[4].

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.

Test Plan Execution. The optimization technique during test plan execution, a test gap analysis, is already used by the team[5] 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 *refactoring* for this study subject. Some tests appear to be partially redundant, so *test case reduction* is promising.

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 SUT 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

[2] CQSE is a contracting partner of Munich Re
[3] Azure Test Plans: https://azure.microsoft.com/de-de/services/devops/test-plans/
[4] Scout: https://www.qualicen.de/scout/
[5] Teamscale: https://teamscale.com/, see also Haas et al. [13]

Table 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			×
Refactoring			🕒
Test quality monitoring			🕒
Test plan optimization			🕒

it as inspiration for additional testing activities. She considers this helpful because it "lowers the risk of missing relevant test cases" and increases the likelihood of detecting faults before deploying the SUT 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."

4.2 Case Study 2: Regression Testing

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 13 additional testers provided targeted testing support for releases. The test management software in use is TestLink[6], an open-source tool that is modified to suit the company's needs.

Applicability of Optimization Techniques. Again, following our process model of Section 3.5, 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 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 [8] and manual testing [17, 18, 22]. In addition, the textual descriptions could further be leveraged using natural language analyses [18, 22].

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 [29].

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

Test Suite Maintenance. Since textual test descriptions and test failure history are available, the optimization techniques *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 [2].

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 *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 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

[6] TestLink: http://testlink.org/

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)."

5 DISCUSSION

5.1 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 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 helps 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.

5.2 Threats to Validity

Internal Validity. A threat to internal validity arises from the Rosenthal effect [32]: 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 for more information and replication.

The set of guidelines presented in Section 3.1 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 [34]. 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 [33] 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.

6 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.

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 9 optimization techniques for manual testing. We discuss prerequisites and caveats for each technique, as they have been described in the literature or reported by practitioners during our study. We further demonstrate 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.

As manual testing will be applied in industry in the foreseeable future without the intention of full automation, as is confirmed by our developer survey, we deem optimization of manual testing to be of significant relevance in practice. Hence, future work shall investigate the identified and proposed optimization techniques in varying settings, especially since existing studies on manual testing are still rare and not unequivocal in their results.

ACKNOWLEDGMENTS

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "SOFIE, 01IS18012A", and the German Research Foundation (DFG), grant "AP 206/14-1". The responsibility for this article lies with the authors.

REFERENCES

[1] Y. Baron and K. Yitmen. 2018. *ISTQB® Worldwide Software Testing Practices*. ISTQB. Retrieved July 8, 2021 from https://www.istqb.org/documents/ISTQB2017-18_Revised.pdf

[2] E. Bernard, J. Botella, F. Ambert, B. Legeard, and M. Utting. 2020. Tool Support for Refactoring Manual Tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 332–342. https://doi.org/10.1109/icst46399.2020.00041

[3] G. Buchgeher, C. Ernstbrunner, R. Ramler, and M. Lusser. 2013. 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, 74–79. https://doi.org/10.1109/ICSTW.2013.16

[4] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner. 2008. Finding Faults: Manual Testing vs. Random+ Testing vs. User Reports. In *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE, 157–166. https://doi.org/10.1109/ISSRE.2008.18

[5] S. Eder, B. Hauptmann, M. Junker, R. Vaas, and K.-H. Prommer. 2014. 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, 29–35. https://doi.org/10.1145/2593501.2593506

[6] S. Elbaum, A. Malishevsky, and G. Rothermel. 2000. Prioritizing Test Cases for Regression Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*. IEEE, 101–112. https://doi.org/10.1145/347324.348910

[7] S. Elbaum, A. Malishevsky, and G. Rothermel. 2001. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. In *Proceedings of the International Conference on Software Engineering*. IEEE, 329–338. https://doi.org/10.1109/icse.2001.919106

[8] S. Elbaum, G. Rothermel, and J. Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM, 235–245. https://doi.org/10.1145/2635868.2635910

[9] E. Engström, P. Runeson, and A. Ljung. 2011. 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, 367–376. https://doi.org/10.1109/ICST.2011.27

[10] E. Engström, P. Runeson, and M. Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology* 52, 1 (2010), 14–30. https://doi.org/10.1016/j.infsof.2009.07.001

[11] V. Garousi and J. Zhi. 2013. A Survey of Software Testing Practices in Canada. *Journal of Systems and Software* 86, 5 (2013), 1354–1376. https://doi.org/10.1016/j.jss.2012.12.051

[12] M. Gligoric, L. Eloussi, and D. Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the International Conference on Software Engineering*. IEEE, 713–716. https://doi.org/10.1109/icse.2015.230

[13] R. Haas, R. Niedermayr, and E. Juergens. 2019. Teamscale: Tackle Technical Debt and Control the Quality of Your Software. In *International Conference on Technical Debt*. IEEE, 55–56. https://doi.org/10.1109/TechDebt.2019.00016

[14] M. Harrold, R. Gupta, and M. Soffa. 1993. A Methodology for Controlling the Size of a Test Suite. *Transactions on Software Engineering and Methodology* 2, 3 (1993), 270–285. https://doi.org/10.1145/152388.152391

[15] M. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. 2001. Empirical Studies of a Prediction Model for Regression Test Selection. *Transactions on Software Engineering* 27, 3 (2001), 248–263. https://doi.org/10.1109/32.910860

[16] B. Hauptmann, L. Heinemann, R. Vaas, and P. Braun. 2013. Hunting for Smells in Natural Language Tests. In *Proceedings of the International Conference on Software Engineering*. IEEE, 1217–1220. https://doi.org/10.1109/ICSE.2013.6606682

[17] H. Hemmati, Z. Fang, and M. Mäntylä. 2015. Prioritizing Manual Test Cases in Traditional and Rapid Release Environments. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 1–10. https://doi.org/10.1109/ICST.2015.7102602

[18] H. Hemmati and F. Sharifi. 2018. Investigating NLP-Based Approaches for Predicting Manual Test Case Failure. In *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE, 309–319. https://doi.org/10.1109/ICST.2018.00038

[19] W. Hudson. 2012. Card Sorting. In *Encyclopedia of Human-Computer Interaction*. Interaction Design Foundation.

[20] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. 2008. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*. Springer, 201–228. https://doi.org/10.1007/978-1-84800-044-5_8

[21] E. Juergens, B. Hummel, F. Deissenboeck, M. Feilkas, C. Schlögel, and A. Wübbeke. 2011. Regression Test Selection of Manual System Tests in Practice. In *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE, 309–312. https://doi.org/10.1109/CSMR.2011.44

[22] R. Lachmann, M. Nieke, C. Seidl, I. Schaefer, and S. Schulze. 2016. System-Level Test Case Prioritization using Machine Learning. In *Proceedings of the International Conference on Machine Learning and Applications*. IEEE, 361–368. https://doi.org/10.1109/ICMLA.2016.0065

[23] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 101–111. https://doi.org/10.1145/3293882.3330570

[24] W. Lam, K. Muslu, H. Sajnani, and S. Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *Proceedings of the International Conference of Software Engineering*. ACM, 1471–1482. https://doi.org/10.1145/3377811.3381749

[25] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 583–594. https://doi.org/10.1145/2950290.2950361

[26] A. Leitner, H. Ciupa, B. Meyer, and M. Howard. 2007. Reconciling Manual and Automated Testing: The AutoTest Experience. In *Proceedings of the Annual Hawaii International Conference on System Sciences*. IEEE, 261a–261a. https://doi.org/10.1109/HICSS.2007.462

[27] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the Symposium on the Foundations of Software Engineering*. ACM, 643–653. https://doi.org/10.1145/2635868.2635920

[28] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. 2019. Predictive Test Selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. IEEE, 91–100. https://doi.org/10.1109/ICSE-SEIP.2019.00018

[29] M. Mossige, A. Gotlieb, H. Spieker, H. Meling, and M. Carlsson. 2017. Time-Aware Test Case Execution Scheduling for Cyber-Physical Systems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*. Springer, 387–404. https://doi.org/10.1007/978-3-319-66158-2_25

[30] T. Nakagawa, K. Munakata, and K. Yamamoto. 2019. 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, 1–6. https://doi.org/10.1109/ISSREW.2019.00033

[31] A. Philip, R. Bhagwan, R. Kumar, C. Maddila, and N. Nagppan. 2019. FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services. In *Proceedings of the International Conference on Software Engineering*. IEEE, 408–418. https://doi.org/10.1109/icse.2019.00054

[32] R. Rosenthal and K. Fode. 1963. The Effect of Experimenter Bias on the Performance of the Albino Rat. *Behavioral Science* 8, 3 (1963), 183–189. https://doi.org/10.1002/bs.3830080302

[33] P. Runeson, M. Host, A. Rainer, and B. Regnell. 2012. *Case Study Research in Software Engineering: Guidelines and Examples* (1st ed.). Wiley. https://doi.org/10.5555/2361717

[34] J. Siegmund, N. Siegmund, and S. Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of the International Conference on Software Engineering*. IEEE, 9–19. https://doi.org/10.5555/2818754.2818759

[35] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander. 2011. Trade-Off Between Automated and Manual Software Testing. *International Journal of Systems Assurance Engineering and Management* 2, 2 (2011), 114–125. https://doi.org/10.1007/s13198-011-0065-6

[36] S. van der Burg and E. Dolstra. 2010. Automating System Tests using Declarative Virtual Machines. In *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE, 181–190. https://doi.org/10.1109/ISSRE.2010.34

[37] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist. 2017. Impediments for Software Test Automation: A Systematic Literature Review. *Software Testing Verification and Reliability* 27, 8 (2017), e1639. https://doi.org/10.1002/stvr.1639

[38] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

Bibliography

[1] Nauman bin Ali, Emelie Engström, Masoumeh Taromirad, Mohammad Reza Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. "On the search for industry-relevant regression testing research". In: *Empirical Software Engineering* 24.4 (2019), pp. 2020–2055.

[2] Jeff Anderson, Saeed Salem, and Hyunsook Do. "Improving the Effectiveness of Test Suite through Mining Historical Data". In: *Proceedings of the Working Conference on Mining Software Repositories*. 2014, pp. 142–151.

[3] Jeff Anderson, Saeed Salem, and Hyunsook Do. "Striving for Failure: An Industrial Case Study about Test Failure Prediction". In: *Proceedings of the International Conference on Software Engineering*. 2015, pp. 49–58.

[4] Maral Azizi and Hyunsook Do. "ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development". In: *Proceedings of the International Symposium on Software Reliability Engineering*. 2018, pp. 144–154.

[5] Victor R. Basili and Richard W. Selby. "Comparing the Effectiveness of Software Testing Strategies". In: *IEEE Transactions on Software Engineering* SE-13.12 (1987), pp. 1278–1296.

[6] Kent Beck. "Embracing Change with Extreme Programming". In: *Computer* 32.10 (1999), pp. 70–77.

[7] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. "DeFlaker: Automatically Detecting Flaky Tests". In: *Proceedings of the International Conference on Software Engineering* (2018), pp. 433–444.

[8] Moritz Beller, Georgios Gousios, and Andy Zaidman. "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration". In: *Proceedings of the International Conference on Mining Software Repositories*. 2017, pp. 447–450.

[9] Antonia Bertolino, Antonio Guerriero, Roberto Pietrantuono, Stefano Russo, Breno Miranda, and Roberto Pietran-Tuono. "Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration". In: *Proceedings of the International Conference on Software Engineering*. 2020, pp. 1–12.

[10] Luca Bigliardi, Michele Lanza, Alberto Bacchelli, Marco Dambros, and Andrea Mocci. "Quantitatively Exploring Non-code Software Artifacts". In: *Proceedings of the International Conference on Quality Software*. 2014, pp. 286–295.

[11] Nick Bilton. *Nest Thermostat Glitch Leaves Users in the Cold*. The New York Times. 2016. URL: `https://www.nytimes.com/2016/01/14/fashion/nest-thermostat-glitch-battery-dies-software-freeze.html` (visited on 2023-04-20).

[12] Derek Bruening. "Efficient, Transparent, and Comprehensive Runtime Code Manipulation". PhD thesis. MIT, Sept. 2004.

[13] Sander Van Der Burg and Eelco Dolstra. "Automating System Tests Using Declarative Virtual Machines". In: *Proceedings of the International Symposium on Software Reliability Engineering*. 2010, pp. 181–190.

[14] Benjamin Busjaeger and Tao Xie. "Learning for Test Prioritization: An Industrial Case Study". In: *Proceedings of the International Symposium on Foundations of Software Engineering*. 2016, pp. 975–980.

[15] Ahmet Celik, Young Chul Lee, and Milos Gligoric. "Regression Test Selection for TizenRT". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 845–850.

[16] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. "Regression Test Selection Across JVM Boundaries". In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 2017, pp. 809–820.

[17] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. "BUILDFAST: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration". In: *Proceedings of the International Conference on Automated Software Engineering*. 2020, pp. 42–53.

[18] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. "Optimizing Test Prioritization via Test Distribution Analysis". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 656–667.

[19] T. Y. Chen, H. Leung, and I. K. Mak. "Adaptive Random Testing". In: *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*. Ed. by Michael J. Maher. Springer Berlin Heidelberg, 2005, pp. 320–329.

[20] Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. "Test-Case Prioritization for Configuration Testing". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2021, pp. 452–465.

[21] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. "Finding Faults: Manual Testing vs. Random Testing+ vs. User Reports". In: *Proceedings of the International Symposium on Software Reliability Engineering*. 2008, pp. 157–166.

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

[23] Hyunsook Do. "Recent Advances in Regression Testing Techniques". In: *Advances in Computers* 103 (2016), pp. 53–77.

[24] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact". In: *Empirical Software Engineering* 10.4 (2005), pp. 405–435.

[25] Hyunsook Do and Gregg Rothermel. "A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults". In: *Proceedings of the International Conference on Software Maintenance*. 2005, pp. 411–420.

[26] Zakir Durumeric et al. “The Matter of Heartbleed”. In: *Proceedings of the Internet Measurement Conference*. 2014, pp. 475–488.

[27] Sebastian Eder, Benedikt Hauptmann, Maximilian Junker, Rudolf Vaas, and Karl Heinz 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*. 2014, pp. 29–35.

[28] Edward Dunn Ekelund and Emelie Engstrom. “Efficient Regression Testing Based on Test History: An Industrial Evaluation”. In: *Proceedings of the International Conference on Software Maintenance and Evolution*. 2015, pp. 449–457.

[29] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. “Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization”. In: *Proceedings of the International Conference on Software Engineering*. 2001, pp. 329–338.

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

[31] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. “Test Case Prioritization: A Family of Empirical Studies”. In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 159–182.

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

[33] Daniel Elsner, Daniel Bertagnolli, Alexander Pretschner, and Rudi Klaus. “Challenges in Regression Test Selection for End-to-End Testing of Microservice-based Software Systems”. In: *Proceedings of the International Conference on Automation of Software Test*. 2022, pp. 1–5.

[34] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. “Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration”. In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2021, pp. 491–504.

[35] Daniel Elsner, Severin Kacianka, Stephan Lipp, Alexander Pretschner, Axel Habermann, Maria Graber, and Silke Reimer. “BinaryRTS: Cross-language Regression Test Selection for C++ Binaries in CI”. In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. 2023, pp. 327–338.

[36] Daniel Elsner, Roland Wuersching, Markus Schnappinger, and Alexander Pretschner. “Probe-based Syscall Tracing for Efficient and Practical File-level Test Traces”. In: *Proceedings of the International Conference on Automation of Software Test*. 2022, pp. 126–137.

[37] Daniel Elsner, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. “Build System Aware Multi-language Regression Test Selection in Continuous Integration”. In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 2022, pp. 87–96.

[38] Emelie Engström, Per Runeson, and Mats Skoglund. "A Systematic Review on Regression Test Selection Techniques". In: *Information and Software Technology* 52.1 (2010), pp. 14–30.

[39] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K. Burke. "Empirical Evaluation of Pareto Efficient Multi-objective Regression Test Case Prioritisation". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2015, pp. 234–245.

[40] Emad Fallahzadeh and Peter C. Rigby. "The Impact of Flaky Tests on Historical Test Prioritization on Chrome". In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 2022, pp. 273–282.

[41] Michael Felderer and Elizabeta Fourneret. "A systematic classification of security regression testing approaches". In: *International Journal on Software Tools for Technology Transfer* 17.3 (2015), pp. 305–319.

[42] Kurt F. Fischer. "A test case selection method for the validation of software maintenance modifications". In: *Proceedings of International Computer Software and Applications Conference*. 1977, pp. 421–426.

[43] Ben Fu, Sasa Misailovic, and Milos Gligoric. "Resurgence of Regression Test Selection for C++". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. 2019, pp. 323–334.

[44] *gitflow-incremental-builder (GIB)*. URL: `https://github.com/gitflow-incremental-builder/gitflow-incremental-builder` (visited on 2023-04-16).

[45] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. "Ekstazi: Lightweight Test Selection". In: *Proceedings of the International Conference on Software Engineering*. 2015, pp. 713–716.

[46] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. "Practical Regression Test Selection with Dynamic File Dependencies". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2015, pp. 211–222.

[47] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[48] Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. "How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1281–1291.

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

[50] Ramzi A. Haraty, Nashat Mansour, and Bassel A. Daou. "Regression Testing of Database Applications". In: *Journal of Database Management* 13.2 (2002), pp. 31–43.

[51] Mark Harman. "Making the Case for MORTO: Multi Objective Regression Test Optimization". In: *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops*. 2011, pp. 111–114.

[52] Mary Jean Harrold, Alessandro Orso, James A. Jones, Tongyu Li, Maikel Pennings, Saurabh Sinha, Ashish Gujarathi, Donglin Liang, and S. Alexander Spoon. "Regression Test Selection for Java Software". In: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Vol. 36. 11. 2001, pp. 312–326.

[53] Hadi Hemmati, Zhihan Fang, and Mika V. Mäntylä. "Prioritizing Manual Test Cases in Traditional and Rapid Release Environments". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. 2015, pp. 1–10.

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

[55] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. "Comparing White-box and Black-box Test Prioritization". In: *Proceedings of the International Conference on Software Engineering*. 2016, pp. 523–534.

[56] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. "The Art of Testing Less without Sacrificing Quality". In: *Proceedings of the International Conference on Software Engineering*. 2015, pp. 483–493.

[57] Dominik Holling. "Defect-based Quality Assurance with Defect Models". PhD thesis. Technical University of Munich, 2016.

[58] Simon Hundsdorfer, Daniel Elsner, and Alexander Pretschner. "DIRTS: Dependency Injection Aware Regression Test Selection". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. 2023, pp. 422–432.

[59] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84.

[60] Y. K. Jang, M. Munro, and Y. R. Kwon. "An improved method of selecting regression tests for C++ programs". In: *Journal of Software Maintenance and Evolution: Research and Practice* 13 (2001), pp. 331–350.

[61] Bo Jiang, Zhenyu Zhang, W. K. Chan, and T. H. Tse. "Adaptive Random Test Case Prioritization". In: *Proceedings of the International Conference on Automated Software Engineering*. 2009, pp. 233–244.

[62] Xianhao Jin and Francisco Servant. "HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration". In: *ACM Trans. Softw. Eng. Methodol.* (2022). Just Accepted.

[63] Xianhao Jin and Francisco Servant. "What Helped, and What Did Not? An Evaluation of the Strategies to Improve Continuous Integration". In: *Proceedings of the International Conference on Software Engineering*. 2021, pp. 213–225.

[64] Claudius Jordan, Philipp Foth, Alexander Pretschner, and Matthias Fruth. "Unreliable Test Infrastructures in Automotive Testing Setups". In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 2022, pp. 307–308.

[65] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. 4th ed. Auerbach Publications, Oct. 2013.

[66] Elmar Juergens, Benjamin Hummel, Florian Deissenboeck, Martin Feilkas, Christian Schlögel, and Andreas Wübbeke. "Regression Test Selection of Manual System Tests in Practice". In: *Proceedings of the European Conference on Software Maintenance and Reengineering*. 2011, pp. 309–312.

[67] Eero Kauhanen, Jukka K. Nurminen, Tommi Mikkonen, and Matvei Pashkovskiy. "Regression Test Selection Tool for Python in Continuous Integration Process". In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. 2021, pp. 618–621.

[68] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. "Effective Regression Test Case Selection: A Systematic Literature Review". In: *ACM Computing Surveys* 50.2 (June 2017), pp. 1–32.

[69] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. "Efficient Binary-Level Coverage Analysis". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1153–1164.

[70] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N. A. Jawawi, and Rooster Tumeng. "Test case prioritization approaches in regression testing: A systematic literature review". In: *Information and Software Technology* 93 (2018), pp. 74–93.

[71] Jung Min Kim and Adam Porter. "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments". In: *Proceedings of the International Conference on Software Engineering*. 2002, pp. 119–129.

[72] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Soder, Agneta Nilsson, and Magnus Castell. "Supporting Continuous Integration by Code-Churn Based Test Selection". In: *Proceedings of the International Workshop on Rapid Continuous Software Engineering*. 2015, pp. 19–25.

[73] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. "A Large Scale Study of Multiple Programming Languages and Code Quality". In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. 2016, pp. 563–573.

[74] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. "Modeling and Ranking Flaky Tests at Apple". In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 2020, pp. 110–119.

[75] Herb Krasner. *The Cost of Poor Software Quality in the US: A 2020 Report*. Consortium for IT Software Quality (CISQ), 2021.

[76] Herb Krasner. *The Cost of Poor Software Quality in the US: A 2022 Report*. Consortium for IT Software Quality (CISQ), 2022.

[77] David C. Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. "Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs". In: *Journal of Object-Oriented Programming* 8.2 (1995), pp. 51–65.

[78] Jung Hyun Kwon and In Young Ko. "Cost-Effective Regression Testing Using Bloom Filters in Continuous Integration Development Environments". In: *Proceedings of the Asia-Pacific Software Engineering Conference*. 2018, pp. 160–168.

[79] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. "Measuring the Cost of Regression Testing in Practice: A Study of Java Projects using Continuous Integration". In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 2017, pp. 821–830.

[80] Remo Lachmann, Manuel Nieke, Christoph Seidl, Ina Schaefer, and Sandro Schulze. "System-Level Test Case Prioritization Using Machine Learning". In: *Proceedings of the International Conference on Machine Learning and Applications*. 2016, pp. 361–368.

[81] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. "Root Causing Flaky Tests in a Large-Scale Industrial Setting". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2019, pp. 101–111.

[82] Wing Lam, Kivanc Muslu, Hitesh Sajnani, and Suresh Thummalapenta. "A Study on the Lifecycle of Flaky Tests". In: *Proceedings of the International Conference of Software Engineering*. 2020, pp. 1471–1482.

[83] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. "Dependent-Test-Aware Regression Testing Techniques". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2020, pp. 298–311.

[84] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. "When Life Gives You Oranges:Detecting and Diagnosing Intermittent Job Failures at Mozilla". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1381–1392.

[85] J. C. Laprie. "Dependability: Basic Concepts and Terminology". In: *Dependable Computing and Fault-Tolerant Systems*. Ed. by J. C. Laprie. Vol. 5. Springer Vienna, 1992.

[86] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. "Prioritizing test cases with string distances". In: *Automated Software Engineering* 19.1 (2012), pp. 65–95.

[87] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. "An Extensive Study of Static Regression Test Selection in Modern Software Evolution". In: *Proceedings of the International Symposium on Foundations of Software Engineering*. 2016, pp. 583–594.

[88] Owolabi Legunsen, August Shi, and Darko Marinov. "STARTS: STAtic Regression Test Selection". In: *Proceedings of the International Conference on Automated Software Engineering*. 2017, pp. 949–954.

[89] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. "Assessing Transition-Based Test Selection Algorithms at Google". In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 2019, pp. 101–110.

[90] Hareton K. N. Leung and Lee White. "A Study of Integration Testing and Software Regression at the Integration Level". In: *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Press, 1990, pp. 290–301.

[91] Hareton K. N. Leung and Lee White. "Insights into Regression Testing". In: *Proceedings of the International Conference on Software Maintenance*. 1989, pp. 60–69.

[92] Nancy G. Leveson and Clark S. Turner. "An Investigation of the Therac-25 Accidents". In: *Computer* 26.7 (1993), pp. 18–41.

[93] Wen Li, Li Li, and Haipeng Cai. "On the Vulnerability Proneness of Multilingual Code". In: *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 847–859.

[94] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. "PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023.

[95] Zheng Li, Mark Harman, and Robert M. Hierons. "Search Algorithms for Regression Test Case Prioritization". In: *IEEE Transactions on Software Engineering* 33.4 (2007), pp. 225–237.

[96] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. "Redefining Prioritization: Continuous Prioritization for Continuous Integration". In: *Proceedings of the International Conference on Software Engineering*. 2018, pp. 688–698.

[97] Panagiotis K. Linos. "PolyCARE: A Tool for Re-engineering Multi-language Program Integrations". In: *Proceedings of the International Conference on Engineering of Complex Computer Systems*. 1995, pp. 338–341.

[98] Jacques-Louis Lions. *Ariane 5 Flight 501 Failure*. European Space Agency, The Inquiry Board, 1996.

[99] LLVM. *LLVM Compiler Infrastructure*. URL: `https://llvm.org` (visited on 2023-04-20).

[100] LLVM. *LLVM XRay Function Call Tracing*. URL: `https://llvm.org/docs/XRay.html` (visited on 2023-04-20).

[101] Zhenyue Long, Zeliu Ao, Guoquan Wu, Wei Chen, and Jun Wei. "WebRTS: A Dynamic Regression Test Selection Tool for Java Web Applications". In: *Proceedings of the International Conference on Software Maintenance and Evolution*. 2020, pp. 822–825.

[102] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. "A Survey on Regression Test-Case Prioritization". In: *Advances in Computers* 113.1 (2019), pp. 1–46.

[103] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. "How Does Regression Test Prioritization Perform in Real-World Software Evolution?" In: *Proceedings of the International Conference on Software Engineering*. 2016, pp. 535–546.

[104] Qi Luo, Kevin Moran, and Denys Poshyvanyk. "A Large-Scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques". In: *Proceedings of the International Symposium on Foundations of Software Engineering*. 2016, pp. 559–570.

[105] Qi Luo, Kevin Moran, Lingming Zhang, and Denys Poshyvanyk. "How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects". In: *IEEE Transactions on Software Engineering* 45.11 (2019), pp. 1054–1080.

[106] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. "An Empirical Analysis of Flaky Tests". In: *Proceedings of the International Symposium on Foundations of Software Engineering*. 2014, pp. 643–653.

[107] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. "Predictive Test Selection". In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 2019, pp. 91–100.

[108] Dusica Marijan, Arnaud Gotlieb, and Abhijeet Sapkota. "Neural Network Classification for Improving Continuous Regression Testing". In: *Proceedings of the International Conference On Artificial Intelligence Testing*. 2020, pp. 123–124.

[109] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study". In: *Proceedings of the International Conference on Software Maintenance*. 2013, pp. 540–543.

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

[111] Dusica Marijan, Marius Liaaen, Arnaud Gotlieb, Sagar Sen, and Carlo Ieva. "TITAN: Test Suite Optimization for Highly Configurable Software". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. 2017, pp. 524–531.

[112] Toni Mattis, Falco Dürsch, and Robert Hirschfeld. "Faster Feedback Through Lexical Test Prioritization". In: *Companion of the International Conference on Art, Science, and Engineering of Programming*. 2019, pp. 1–10.

[113] Toni Mattis and Robert Hirschfeld. "Lightweight Lexical Test Prioritization for Immediate Feedback". In: *The Art, Science, and Engineering of Programming* 4.3 (2020), pp. 1–32.

[114] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. "RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization". In: *Proceedings of the Conference on Mining Software Repositories*. 2020, pp. 385–396.

[115] Philip Mayer and Alexander Bauer. "An Empirical Analysis of the Utilization of Multiple Programming Languages in Open Source Projects". In: *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. 2015, pp. 1–10.

[116] Philip Mayer, Michael Kirsch, and Minh Anh Le. "On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers". In: *Journal of Software Engineering Research and Development* 5.1 (2017).

[117] Philip Mayer and Andreas Schroeder. "Automated Multi-Language Artifact Binding and Rename Refactoring between Java and DSLs Used by Java Frameworks". In: *Proceedings of the European Conference on Object-Oriented Programming*. 2014, pp. 437–462.

[118] Sonu Mehta, Farima Farmahinifarahani, Ranjita Bhagwan, Suraj Guptha, Sina Jafari, Rahul Kumar, Vaibhav Saini, and Anirudh Santhiar. "Data-driven Test Selection at Scale". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1225–1235.

[119] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. "A Static Approach to Prioritizing JUnit Test Cases". In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1258–1275.

[120] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. "FAST Approaches to Scalable Similarity-based Test Case Prioritization". In: *Proceedings of the International Conference on Software Engineering*. 2018, pp. 222–232.

[121] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. "PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2017, pp. 23–34.

[122] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The Art of Software Testing*. John Wiley and Sons, Inc., Jan. 2012.

[123] Armin Najafi, Weiyi Shang, and Peter C. Rigby. "Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report". In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 2019, pp. 213–222.

[124] Takao Nakagawa, Kazuki Munakata, and Koji 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*. 2019, pp. 1–6.

[125] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. "Regression Testing in the Presence of Non-code Changes". In: *Proceedings of the International Conference on Software Testing, Verification, and Validation*. 2011, pp. 21–30.

[126] Sathish Natarajan and Dhiraj Sinha. *World Quality Report 2020-21*. Capgemini, 2020.

[127] Nicholas Nethercote. "Dynamic Binary Analysis and Instrumentation or Building Tools is Easy". PhD thesis. University of Cambridge, Nov. 2004.

[128] Tanzeem Bin Noor and Hadi Hemmati. "A similarity-based approach for test case prioritization using historical failure data". In: *Proceedings of the International Symposium on Software Reliability Engineering*. 2016, pp. 58–68.

[129] Tanzeem Bin Noor and Hadi Hemmati. "Studying Test Case Failure Prediction for Test Case Prioritization". In: *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering*. 2017, pp. 2–11.

[130] Jesper Öqvist, Görel Hedin, and Boris Magnusson. "Extraction-Based Regression Test Selection". In: *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2016, pp. 1–10.

[131] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. "Scaling Regression Testing to Large Software Systems". In: *Proceedings of the International Symposium on Foundations of Software Engineering*. 2004, pp. 241–251.

[132] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. "Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review". In: *Empirical Software Engineering* 27.2 (2022), pp. 1–34.

[133] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. "A Survey of Flaky Tests". In: *ACM Trans. Softw. Eng. Methodol.* 31.1 (Oct. 2021).

[134] Qianyang Peng, August Shi, and Lingming Zhang. "Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2020, pp. 324–336.

[135] Adithya Abraham Philip, Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Nachiappan Nagppan. "FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services". In: *Proceedings of the International Conference on Software Engineering*. 2019, pp. 408–418.

[136] Alexander Pretschner. "Defect-Based Testing". In: *Dependable Software Systems Engineering*. Ed. by Maximilian Irlbeck, Doron Peled, and Alexander Pretschner. Vol. 40. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2015, pp. 224–245.

[137] Xiao Qu, Mithun Acharya, and Brian Robinson. "Configuration Selection Using Code Change Impact Analysis for Regression Testing". In: *Proceedings of the International Conference on Software Maintenance*. 2012, pp. 129–138.

[138] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. "Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2008, pp. 75–86.

[139] Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. "Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization". In: *Proceedings of the International Conference on Software Maintenance*. 2007, pp. 255–264.

[140] Maaz Hafeez Ur Rehman and Peter C. Rigby. "Quantifying No-Fault-Found Test Failures to Prioritize Inspection of Flaky Tests at Ericsson". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1371–1380.

[141] Raúl H. Rosero, Omar S. Gómez, and Glen Rodríguez. "15 Years of Software Regression Testing Techniques - A Survey". In: *International Journal of Software Engineering and Knowledge Engineering* 26.5 (2016), pp. 675–689.

[142] Gregg Rothermel. "Efficient, Effective Regression Testing Using Safe Test Selection Techniques". PhD thesis. Clemson University, May 1996.

[143] Gregg Rothermel and Mary Jean Harrold. "A Framework for Evaluating Regression Test Selection Techniques". In: *Proceedings of the International Conference on Software Engineering*. 1994, pp. 201–210.

[144] Gregg Rothermel and Mary Jean Harrold. "A Safe, Efficient Regression Test Selection Technique". In: *ACM Transactions on Software Engineering and Methodology* 6.2 (1997), pp. 173–210.

[145] Gregg Rothermel and Mary Jean Harrold. "Analyzing Regression Test Selection Techniques". In: *IEEE Transactions on Software Engineering* 22.8 (1996), pp. 529–551.

[146] Gregg Rothermel and Mary Jean Harrold. "Empirical Studies of a Safe Regression Test Selection Technique". In: *IEEE Transactions on Software Engineering* 24.6 (1998), pp. 401–419.

[147] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. "Regression Test Selection for C++ Software". In: *Software Testing, Verification and Reliability* 10.2 (2000), pp. 77–109.

[148] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. "Prioritizing Test Cases for Regression Testing". In: *IEEE Transactions on Software Engineering* 27.10 (2001), pp. 929–948.

[149] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. "Test Case Prioritization: An Empirical Study". In: *Proceedings of the International Conference on Software Maintenance*. 1999, pp. 179–188.

[150] Khaled Walid Al-Sabbagh, Miroslaw Staron, Regina Hebig, and Wilhelm Meding. "Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns". In: *Joint Proceedings of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*. Vol. 2476. 2019, pp. 138–153.

[151] Khaled Walid Al-Sabbagh, Miroslaw Staron, Miroslaw Ochodek, Regina Hebig, and Wilhelm Meding. "Selective Regression Testing based on Big Data: Comparing Feature Extraction Techniques". In: *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*. 2020, pp. 322–329.

[152] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. "An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes". In: *Proceedings of the International Conference on Software Engineering*. 2015, pp. 268–279.

[153] Mark Sherriff, Mike Lake, and Laurie Williams. "Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records". In: *Proceedings of the International Symposium on Software Reliability Engineering*. 2007, pp. 81–90.

[154] August Shi, Jonathan Bell, and Darko Marinov. "Mitigating the Effects of Flaky Tests on Mutation Testing". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2019, pp. 112–122.

[155] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. "Evaluating Test-Suite Reduction in Real Software Evolution". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2018, pp. 84–94.

[156] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. "Optimizing Test Placement for Module-Level Regression Testing". In: *Proceedings of the International Conference on Software Engineering*. 2017, pp. 689–699.

[157] August Shi, Peiyuan Zhao, and Darko Marinov. "Understanding and Improving Regression Test Selection in Continuous Integration". In: *Proceedings of the International Symposium on Software Reliability Engineering*. 2019, pp. 228–238.

[158] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. "Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2017, pp. 12–22.

[159] *Stack Overflow Annual Developer Survey 2022*. URL: `https://survey.stackoverflow.co/2022/#technology-most-popular-technologies` (visited on 2023-04-20).

[160] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. "Testing Configuration Changes in Context to Prevent Production Failures". In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 2020, pp. 735–751.

[161] Stephen W. Thomas, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. "Static test case prioritization using topic models". In: *Empirical Software Engineering* 19.1 (2014), pp. 182–212.

[162] Fabian Trautsch, Steffen Herbold, and Jens Grabowski. "Are Unit and Integration Test Definitions Still Valid for Modern Java Projects? An Empirical Study on Open-Source Projects". In: *Journal of Systems and Software* 159:110421 (2020).

[163] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. "File-Level vs. Module-Level Regression Test Selection for .NET". In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. 2017, pp. 848–853.

[164] Shuai Wang, Shaukat Ali, Tao Yue, Oyvind Bakkeli, and Marius Liaaen. "Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-objective Search". In: *Proceedings of the International Conference on Software Engineering Companion*. 2016, pp. 182–191.

[165] Lee White, Khaled Jaber, Brian Robinson, and Václav Rajlich. "Extended firewall for regression testing: An experience report". In: *Journal of Software Maintenance and Evolution* 20.6 (2008), pp. 419–433.

[166] Kristian Wiklund, Sigrid Eldh, Daniel Sundmark, and Kristina Lundqvist. "Impediments for software test automation: A systematic literature review". In: *Software Testing Verification and Reliability* 27.8 (2017), e1639.

[167] David Willmor and Suzanne M. Embury. "A safe regression test selection technique for database-driven applications". In: *Proceedings of the International Conference on Software Maintenance*. 2005, pp. 421–430.

[168] W. Eric Wong, J. R. Horgan, Saul London, and Hira Agrawal. "A Study of Effective Regression Testing in Practice". In: *Proceedings of the International Symposium on Software Reliability Engineering*. 1997, pp. 264–274.

[169] Roland Wuersching, Daniel Elsner, Fabian Leinen, Alexander Pretschner, Georg Grueneissl, Thomas Neumeyr, and Tobias Vosseler. "Severity-Aware Prioritization of System-Level Regression Tests in Automotive Software". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. 2023, pp. 398–409.

[170] Ahmadreza Saboor Yaraghi, Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. "Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts". In: *IEEE Transactions on Software Engineering* 49.4 (Apr. 1, 2023), pp. 1615–1639.

[171] Pu Yi, Hao Wang, Tao Xie, Darko Marinov, and Wing Lam. "A Theoretical Analysis of Random Regression Test Prioritization". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13244 LNCS. Springer International Publishing, 2022, pp. 217–235.

[172] Shin Yoo and Mark Harman. "Regression testing minimization, selection and prioritization: A survey". In: *Software Testing Verification and Reliability* 22.2 (2012), pp. 67–120.

[173] Tingting Yu and Ting Wang. "A Study of Regression Test Selection in Continuous Integration Environments". In: *Proceedings of the International Symposium on Software Reliability Engineering*. 2018, pp. 135–143.

[174] Lingming Zhang. "Hybrid Regression Test Selection". In: *Proceedings of the International Conference on Software Engineering*. 2018, pp. 199–209.

[175] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. "FaultTracer: A spectrum-based approach to localizing failure-inducing program edits". In: *Journal of Software: Evolution and Process* 25.12 (2013), pp. 1357–1383.

[176] Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, and Hong Mei. "Prioritizing JUnit Test Cases in Absence of Coverage Information". In: *Proceedings of the International Conference on Software Maintenance*. 2009, pp. 19–28.

[177] Wei Zheng, Guoliang Liu, Manqing Zhang, Xiang Chen, and Wenqiao Zhao. "Research Progress of Flaky Tests". In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. 2021, pp. 639–646.

[178] Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. "TestSage: Regression Test Selection for Large-scale Web Service Testing". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. 2019, pp. 430–440.

[179] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. "A Framework for Checking Regression Test Selection Tools". In: *Proceedings of the International Conference on Software Engineering*. 2019, pp. 430–441.

[180] Yuecai Zhu, Emad Shihab, and Peter C. Rigby. "Test Re-prioritization in Continuous Testing Environments". In: *Proceedings of the International Conference on Software Maintenance and Evolution*. 2018, pp. 69–79.

[181] Celal Ziftci and Jim Reardon. "Who Broke the Build? Automatically Identifying Changes That Induce Test Failures In Continuous Integration at Google Scale". In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 2017, pp. 113–122.

[182] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. "Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process". In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. 2009, pp. 91–100.

List of Acronyms

APFD Average Percentage of Faults Detected

APFD$_C$ Average Percentage of Faults Detected per Cost

API Application Programming Interface

CFG Control-Flow Graph

CI Continuous Integration

DI Dependency Injection

DSL Domain-Specific Language

DLL Dynamic-link Library

GPL General-Purpose Language

IR Information Retrieval

JAR Java Archive

JNI Java Native Interface

JVM Java Virtual Machine

ML Machine Learning

RL Reinforcement Learning

RTO Regression Test Optimization

RTP Regression Test Prioritization

RTS Regression Test Selection

SUT System Under Test

TSM Test Suite Minimization

VCS Version Control System

List of Figures